Spring Framework Documentation

Version 6.0.1

Chapter 1. Spring Framework Overview

Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment, with support for Groovy and Kotlin as alternative languages on the JVM, and with the flexibility to create many kinds of architectures depending on an application's needs. As of Spring Framework 6.0, Spring requires Java 17+.

Spring supports a wide range of application scenarios. In a large enterprise, applications often exist for a long time and have to run on a JDK and application server whose upgrade cycle is beyond developer control. Others may run as a single jar with the server embedded, possibly in a cloud environment. Yet others may be standalone applications (such as batch or integration workloads) that do not need a server.

Spring is open source. It has a large and active community that provides continuous feedback based on a diverse range of real-world use cases. This has helped Spring to successfully evolve over a very long time.

1.1. What We Mean by "Spring"

The term "Spring" means different things in different contexts. It can be used to refer to the Spring Framework project itself, which is where it all started. Over time, other Spring projects have been built on top of the Spring Framework. Most often, when people say "Spring", they mean the entire family of projects. This reference documentation focuses on the foundation: the Spring Framework itself.

The Spring Framework is divided into modules. Applications can choose which modules they need. At the heart are the modules of the core container, including a configuration model and a dependency injection mechanism. Beyond that, the Spring Framework provides foundational support for different application architectures, including messaging, transactional data and persistence, and web. It also includes the Servlet-based Spring MVC web framework and, in parallel, the Spring WebFlux reactive web framework.

A note about modules: Spring's framework jars allow for deployment to JDK 9's module path ("Jigsaw"). For use in Jigsaw-enabled applications, the Spring Framework 5 jars come with "Automatic-Module-Name" manifest entries which define stable language-level module names ("spring.core", "spring.context", etc.) independent from jar artifact names (the jars follow the same naming pattern with "-" instead of ".", e.g. "spring-core" and "spring-context"). Of course, Spring's framework jars keep working fine on the classpath on both JDK 8 and 9+.

1.2. History of Spring and the Spring Framework

Spring came into being in 2003 as a response to the complexity of the early J2EE specifications. While some consider Java EE and its modern-day successor Jakarta EE to be in competition with Spring, they are in fact complementary. The Spring programming model does not embrace the Jakarta EE platform specification; rather, it integrates with carefully selected individual specifications from the traditional EE umbrella:

- Servlet API (JSR 340)

- WebSocket API (JSR 356)
- Concurrency Utilities (JSR 236)
- JSON Binding API (JSR 367)
- Bean Validation (JSR 303)
- JPA (JSR 338)
- JMS (JSR 914)
- as well as JTA/JCA setups for transaction coordination, if necessary.

The Spring Framework also supports the Dependency Injection (JSR 330) and Common Annotations (JSR 250) specifications, which application developers may choose to use instead of the Spring-specific mechanisms provided by the Spring Framework. Originally, those were based on common `javax` packages.

As of Spring Framework 6.0, Spring has been upgraded to the Jakarta EE 9 level (e.g. Servlet 5.0+, JPA 3.0+), based on the `jakarta` namespace instead of the traditional `javax` packages. With EE 9 as the minimum and EE 10 supported already, Spring is prepared to provide out-of-the-box support for the further evolution of the Jakarta EE APIs. Spring Framework 6.0 is fully compatible with Tomcat 10.1, Jetty 11 and Undertow 2.3 as web servers, and also with Hibernate ORM 6.1.

Over time, the role of Java/Jakarta EE in application development has evolved. In the early days of J2EE and Spring, applications were created to be deployed to an application server. Today, with the help of Spring Boot, applications are created in a devops- and cloud-friendly way, with the Servlet container embedded and trivial to change. As of Spring Framework 5, a WebFlux application does not even use the Servlet API directly and can run on servers (such as Netty) that are not Servlet containers.

Spring continues to innovate and to evolve. Beyond the Spring Framework, there are other projects, such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others. It's important to remember that each project has its own source code repository, issue tracker, and release cadence. See spring.io/projects for the complete list of Spring projects.

1.3. Design Philosophy

When you learn about a framework, it's important to know not only what it does but what principles it follows. Here are the guiding principles of the Spring Framework:

- Provide choice at every level. Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs.
- Accommodate diverse perspectives. Spring embraces flexibility and is not opinionated about how things should be done. It supports a wide range of application needs with different perspectives.
- Maintain strong backward compatibility. Spring's evolution has been carefully managed to force few breaking changes between versions. Spring supports a carefully chosen range of JDK

versions and third-party libraries to facilitate maintenance of applications and libraries that depend on Spring.

- Care about API design. The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.
- Set high standards for code quality. The Spring Framework puts a strong emphasis on meaningful, current, and accurate javadoc. It is one of very few projects that can claim clean code structure with no circular dependencies between packages.

1.4. Feedback and Contributions

For how-to questions or diagnosing or debugging issues, we suggest using Stack Overflow. Click here for a list of the suggested tags to use on Stack Overflow. If you're fairly certain that there is a problem in the Spring Framework or would like to suggest a feature, please use the GitHub Issues.

If you have a solution in mind or a suggested fix, you can submit a pull request on Github. However, please keep in mind that, for all but the most trivial issues, we expect a ticket to be filed in the issue tracker, where discussions take place and leave a record for future reference.

For more details see the guidelines at the CONTRIBUTING, top-level project page.

1.5. Getting Started

If you are just getting started with Spring, you may want to begin using the Spring Framework by creating a Spring Boot-based application. Spring Boot provides a quick (and opinionated) way to create a production-ready Spring-based application. It is based on the Spring Framework, favors convention over configuration, and is designed to get you up and running as quickly as possible.

You can use start.spring.io to generate a basic project or follow one of the "Getting Started" guides, such as Getting Started Building a RESTful Web Service. As well as being easier to digest, these guides are very task focused, and most of them are based on Spring Boot. They also cover other projects from the Spring portfolio that you might want to consider when solving a particular problem.

Chapter 2. Core Technologies

This part of the reference documentation covers all the technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container. A thorough treatment of the Spring Framework's IoC container is closely followed by comprehensive coverage of Spring's Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand and which successfully addresses the 80% sweet spot of AOP requirements in Java enterprise programming.

Coverage of Spring's integration with AspectJ (currently the richest — in terms of features — and certainly most mature AOP implementation in the Java enterprise space) is also provided.

AOT processing can be used to optimize your application ahead-of-time. It is typically used for native image deployment using GraalVM.

2.1. The IoC Container

This chapter covers Spring's Inversion of Control (IoC) container.

2.1.1. Introduction to the Spring IoC Container and Beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the Service Locator pattern.

The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. The `BeanFactory` interface provides an advanced configuration mechanism capable of managing any type of object. `ApplicationContext` is a sub-interface of `BeanFactory`. It adds:

- Easier integration with Spring's AOP features
- Message resource handling (for use in internationalization)
- Event publication
- Application-layer specific contexts such as the `WebApplicationContext` for use in web applications.

In short, the `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality. The `ApplicationContext` is a complete superset of the `BeanFactory` and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the `BeanFactory` instead of the

`ApplicationContext`, see the section covering the `BeanFactory` API.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

2.1.2. Container Overview

The `org.springframework.context.ApplicationContext` interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.

Several implementations of the `ApplicationContext` interface are supplied with Spring. In stand-alone applications, it is common to create an instance of `ClassPathXmlApplicationContext` or `FileSystemXmlApplicationContext`. While XML has been the traditional format for defining configuration metadata, you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the `web.xml` file of the application typically suffices (see Convenient ApplicationContext Instantiation for Web Applications). If you use the Spring Tools for Eclipse (an Eclipse-powered development environment), you can easily create this boilerplate configuration with a few mouse clicks or keystrokes.

The following diagram shows a high-level view of how Spring works. Your application classes are combined with configuration metadata so that, after the `ApplicationContext` is created and initialized, you have a fully configured and executable system or application.

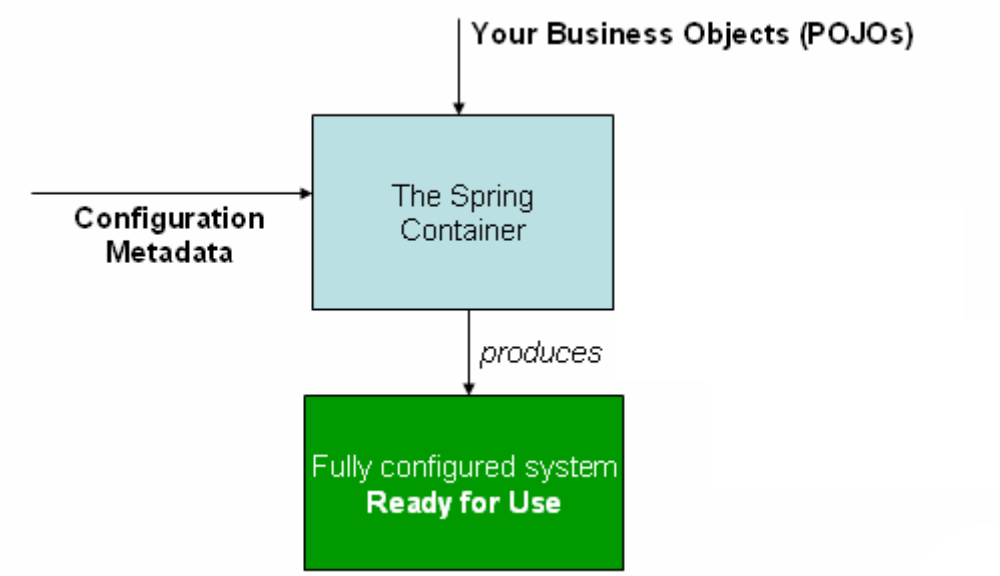


Figure 1. The Spring IoC container

Configuration Metadata

As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose Java-based configuration for their Spring applications.

For information about using other forms of metadata with the Spring container, see:

- Annotation-based configuration: define beans using annotation-based configuration metadata.
- Java-based configuration: define beans external to your application classes by using Java rather than XML files. To use these features, see the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata configures these beans as `<bean/>` elements inside a top-level `<beans/>` element. Java configuration typically uses `@Bean`-annotated methods within a `@Configuration` class.

These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, persistence layer objects such as repositories or data access objects (DAOs), presentation objects such as Web controllers, infrastructure objects such as a JPA `EntityManagerFactory`, JMS queues, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of repositories and business logic to create and load domain objects.

The following example shows the basic structure of XML-based configuration metadata:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="..." class="..."> ① ②
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

① The `id` attribute is a string that identifies the individual bean definition.

② The `class` attribute defines the type of the bean and uses the fully qualified class name.

The value of the `id` attribute can be used to refer to collaborating objects. The XML for referring to collaborating objects is not shown in this example. See Dependencies for more information.

Instantiating a Container

The location path or paths supplied to an `ApplicationContext` constructor are resource strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java `CLASSPATH`, and so on.

Java

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",
"daos.xml");
```

Kotlin

```
val context = ClassPathXmlApplicationContext("services.xml", "daos.xml")
```

After you learn about Spring's IoC container, you may want to know more about Spring's `Resource` abstraction (as described in Resources), which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, `Resource` paths are used to construct applications contexts, as described in Application Contexts and Resource Paths.

The following example shows the service layer objects `(services.xml)` configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- services -->

    <bean id="petStore"
class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for services go here -->

</beans>
```

The following example shows the data access objects `daos.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao"
class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

In the preceding example, the service layer consists of the `PetStoreServiceImpl` class and two data access objects of the types `JpaAccountDao` and `JpaItemDao` (based on the JPA Object-Relational Mapping standard). The `property name` element refers to the name of the JavaBean property, and the `ref` element refers to the name of another bean definition. This linkage between `id` and `ref` elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see Dependencies.

Composing XML-based Configuration Metadata

It can be useful to have bean definitions span multiple XML files. Often, each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple `Resource` locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<import/>` element to load bean definitions from another file or files. The following example shows how to do so:

```
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

In the preceding example, external bean definitions are loaded from three files: `services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are relative to the definition file doing the importing, so `services.xml` must be in the same directory or classpath location as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is ignored. However, given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level `<beans/>` element, must be valid XML bean definitions, according to the Spring Schema.

It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for `classpath:` URLs (for example, `classpath:../services.xml`), where the runtime resolution process chooses the “nearest” classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.

You can always use fully qualified resource locations instead of relative paths: for example, `file:C:/config/services.xml` or `classpath:/config/services.xml`. However, be aware that you are coupling your application’s configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations — for example, through "${...}" placeholders that are resolved against JVM system properties at runtime.

The namespace itself provides the import directive feature. Further configuration features beyond plain bean definitions are available in a selection of XML namespaces provided by Spring — for example, the `context` and `util` namespaces.

The Groovy Bean Definition DSL

As a further example for externalized configuration metadata, bean definitions can also be expressed in Spring's Groovy Bean Definition DSL, as known from the Grails framework. Typically, such configuration live in a ".groovy" file with the structure shown in the following example:

```
beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

This configuration style is largely equivalent to XML bean definitions and even supports Spring's XML configuration namespaces. It also allows for importing XML bean definition files through an `importBeans` directive.

Using the Container

The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. By using the method `T getBean(String name, Class<T> requiredType)`, you can retrieve instances of your beans.

The `ApplicationContext` lets you read bean definitions and access them, as the following example shows:

Java

```
// create and configure beans
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",
"daos.xml");

// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);

// use configured instance
List<String> userList = service.getUsernameList();
```

Kotlin

```
import org.springframework.beans.factory.getBean

// create and configure beans
val context = ClassPathXmlApplicationContext("services.xml", "daos.xml")

// retrieve configured instance
val service = context.getBean<PetStoreService>("petStore")

// use configured instance
var userList = service.getUsernameList()
```

With Groovy configuration, bootstrapping looks very similar. It has a different context implementation class which is Groovy-aware (but also understands XML bean definitions). The following example shows Groovy configuration:

Java

```
ApplicationContext context = new GenericGroovyApplicationContext("services.groovy",
"daos.groovy");
```

Kotlin

```
val context = GenericGroovyApplicationContext("services.groovy", "daos.groovy")
```

The most flexible variant is `GenericApplicationContext` in combination with reader delegates — for example, with `XmlBeanDefinitionReader` for XML files, as the following example shows:

Java

```
GenericApplicationContext context = new GenericApplicationContext();
new XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");
context.refresh();
```

Kotlin

```
val context = GenericApplicationContext()
XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml")
context.refresh()
```

You can also use the `GroovyBeanDefinitionReader` for Groovy files, as the following example shows:

Java

```
GenericApplicationContext context = new GenericApplicationContext();
new GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",
"daos.groovy");
context.refresh();
```

Kotlin

```
val context = GenericApplicationContext()
GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",
"daos.groovy")
context.refresh()
```

You can mix and match such reader delegates on the same `ApplicationContext`, reading bean definitions from diverse configuration sources.

You can then use `getBean` to retrieve instances of your beans. The `ApplicationContext` interface has a few other methods for retrieving beans, but, ideally, your application code should never use them. Indeed, your application code should have no calls to the `getBean()` method at all and thus have no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides dependency injection for various web framework components such as controllers and JSF-managed beans, letting you declare a dependency on a specific bean through metadata (such as an autowiring annotation).

2.1.3. Bean Overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML `<bean/>` definitions).

Within the container itself, these bean definitions are represented as `BeanDefinition` objects, which contain (among other information) the following metadata:

- A package-qualified class name: typically, the actual implementation class of the bean being defined.
- Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
- References to other beans that are needed for the bean to do its work. These references are also called collaborators or dependencies.
- Other configuration settings to set in the newly created object — for example, the size limit of the pool or the number of connections to use in a bean that manages a connection pool.

This metadata translates to a set of properties that make up each bean definition. The following table describes these properties:

Table 1. The bean definition

Property	Explained in...
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

In addition to bean definitions that contain information on how to create a specific bean, the `ApplicationContext` implementations also permit the registration of existing objects that are created outside the container (by users). This is done by accessing the ApplicationContext's `BeanFactory` through the `getBeanFactory()` method, which returns the `DefaultListableBeanFactory` implementation. `DefaultListableBeanFactory` supports this registration through the `registerSingleton(..)` and `registerBeanDefinition(..)` methods. However, typical applications work solely with beans defined through regular bean definition metadata.

Bean metadata and manually supplied singleton instances need to be registered as early as possible, in order for the container to properly reason about them during autowiring and other introspection steps. While overriding existing metadata and existing singleton instances is supported to some degree, the registration of new beans at runtime (concurrently with live access to the factory) is not officially supported and may lead to concurrent access exceptions, inconsistent state in the bean container, or both.

Naming Beans

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier. However, if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the `id` attribute, the `name` attribute, or both to specify the bean identifiers. The `id` attribute lets you specify exactly one id. Conventionally, these names are alphanumeric ('myBean', 'someService', etc.), but they can contain special characters as well. If you want to introduce other aliases for the bean, you can also specify them in the `name` attribute, separated by a comma (`,`), semicolon (`;`), or white space. As a historical note, in versions prior to Spring 3.1, the `id` attribute was defined as an `xsd:ID` type, which constrained possible characters. As of 3.1, it is defined as an `xsd:string` type. Note that bean `id` uniqueness is still enforced by the container, though no longer by XML parsers.

You are not required to supply a `name` or an `id` for a bean. If you do not supply a `name` or `id` explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the `ref` element or a Service Locator style lookup, you must provide a

name. Motivations for not supplying a name are related to using inner beans and autowiring collaborators.

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include `accountManager`, `accountService`, `userDao`, `loginController`, and so forth.

Naming beans consistently makes your configuration easier to read and understand. Also, if you use Spring AOP, it helps a lot when applying advice to a set of beans related by name.

With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules described earlier: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by `java.beans.Introspector.decapitalize` (which Spring uses here).

Aliasing a Bean outside the Bean Definition

In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the `id` attribute and any number of other names in the `name` attribute. These names can be equivalent aliases to the same bean and are useful for some situations, such as letting each component in an application refer to a common dependency by using a bean name that is specific to that component itself.

Specifying all aliases where the bean is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, with each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the `<alias/>` element to accomplish this. The following example shows how to do so:

```
<alias name="fromName" alias="toName"/>
```

In this case, a bean (in the same container) named `fromName` may also, after the use of this alias definition, be referred to as `toName`.

For example, the configuration metadata for subsystem A may refer to a DataSource by the name of `subsystemA-dataSource`. The configuration metadata for subsystem B may refer to a DataSource by the name of `subsystemB-dataSource`. When composing the main application that uses both these subsystems, the main application refers to the DataSource by the name of `myApp-dataSource`. To have all three names refer to the same object, you can add the following alias definitions to the configuration metadata:

```
<alias name="myApp-dataSource" alias="subsystemA-dataSource"/>
<alias name="myApp-dataSource" alias="subsystemB-dataSource"/>
```

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

Java-configuration

If you use Javaconfiguration, the `@Bean` annotation can be used to provide aliases. See Using the `@Bean` Annotation for details.

Instantiating Beans

A bean definition is essentially a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the `class` attribute of the `<bean/>` element. This `class` attribute (which, internally, is a `Class` property on a `BeanDefinition` instance) is usually mandatory. (For exceptions, see Instantiation by Using an Instance Factory Method and Bean Definition Inheritance.) You can use the `Class` property in one of two ways:

- Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code with the `new` operator.
- To specify the actual class containing the `static` factory method that is invoked to create the object, in the less common case where the container invokes a `static` factory method on a class to create the bean. The object type returned from the invocation of the `static` factory method may be the same class or another class entirely.

Nested class names

If you want to configure a bean definition for a nested class, you may use either the binary name or the source name of the nested class.

For example, if you have a class called `SomeThing` in the `com.example` package, and this `SomeThing` class has a `static` nested class called `OtherThing`, they can be separated by a dollar sign (`$`) or a dot (`.`). So the value of the `class` attribute in a bean definition would be `com.example.SomeThing$OtherThing` or `com.example.SomeThing.OtherThing`.

Instantiation with a Constructor

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific

interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually any class you want it to manage. It is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container. You can also have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

With XML-based configuration metadata you can specify your bean class as follows:

```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

For details about the mechanism for supplying arguments to the constructor (if required) and setting object instance properties after the object is constructed, see Injecting Dependencies.

Instantiation with a Static Factory Method

When defining a bean that you create with a static factory method, use the `class` attribute to specify the class that contains the `static` factory method and an attribute named `factory-method` to specify the name of the factory method itself. You should be able to call this method (with optional arguments, as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call `static` factories in legacy code.

The following bean definition specifies that the bean will be created by calling a factory method. The definition does not specify the type (class) of the returned object, but rather the class containing the factory method. In this example, the `createInstance()` method must be a `static` method. The following example shows how to specify a factory method:

```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

The following example shows a class that would work with the preceding bean definition:

Java

```
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

Kotlin

```
class ClientService private constructor() {
    companion object {
        private val clientService = ClientService()
        @JvmStatic
        fun createInstance() = clientService
    }
}
```

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after the object is returned from the factory, see Dependencies and Configuration in Detail.

Instantiation by Using an Instance Factory Method

Similar to instantiation through a static factory method, instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the `class` attribute empty and, in the `factory-bean` attribute, specify the name of a bean in the current (or parent or ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the `factory-method` attribute. The following example shows how to configure such a bean:

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

The following example shows the corresponding class:

Java

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

Kotlin

```
class DefaultServiceLocator {
    companion object {
        private val clientService = ClientServiceImpl()
    }
    fun createClientServiceInstance(): ClientService {
        return clientService
    }
}
```

One factory class can also hold more than one factory method, as the following example shows:

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

The following example shows the corresponding class:

Java

```
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    private static AccountService accountService = new AccountServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

Kotlin

```
class DefaultServiceLocator {
    companion object {
        private val clientService = ClientServiceImpl()
        private val accountService = AccountServiceImpl()
    }

    fun createClientServiceInstance(): ClientService {
        return clientService
    }

    fun createAccountServiceInstance(): AccountService {
        return accountService
    }
}
```

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See Dependencies and Configuration in Detail.

In Spring documentation, "factory bean" refers to a bean that is configured in the Spring container and that creates objects through an instance or static factory method. By contrast, `FactoryBean` (notice the capitalization) refers to a Spring-specific `FactoryBean` implementation class.

Determining a Bean's Runtime Type

The runtime type of a specific bean is non-trivial to determine. A specified class in the bean metadata definition is just an initial class reference, potentially combined with a declared factory method or being a `FactoryBean` class which may lead to a different runtime type of the bean, or not being set at all in case of an instance-level factory method (which is resolved via the specified `factory-bean` name instead). Additionally, AOP proxying may wrap a bean instance with an

interface-based proxy with limited exposure of the target bean's actual type (just its implemented interfaces).

The recommended way to find out about the actual runtime type of a particular bean is a `BeanFactory.getType` call for the specified bean name. This takes all of the above cases into account and returns the type of object that a `BeanFactory.getBean` call is going to return for the same bean name.

2.1.4. Dependencies

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand alone to a fully realized application where objects collaborate to achieve a goal.

Dependency Injection

Dependency injection (DI) is a process whereby objects define their dependencies (that is, the other objects with which they work) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes or the Service Locator pattern.

Code is cleaner with the DI principle, and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies. As a result, your classes become easier to test, particularly when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants: Constructor-based dependency injection and Setter-based dependency injection.

Constructor-based Dependency Injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a `static` factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a `static` factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection:

Java

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on a MovieFinder
    private final MovieFinder movieFinder;

    // a constructor so that the Spring container can inject a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Kotlin

```
// a constructor so that the Spring container can inject a MovieFinder
class SimpleMovieLister(private val movieFinder: MovieFinder) {
    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Notice that there is nothing special about this class. It is a POJO that has no dependencies on container specific interfaces, base classes, or annotations.

Constructor Argument Resolution

Constructor argument resolution matching occurs by using the argument's type. If no potential ambiguity exists in the constructor arguments of a bean definition, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider the following class:

Java

```
package x.y;

public class ThingOne {

    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {
        // ...
    }
}
```

Kotlin

```
package x.y

class ThingOne(thingTwo: ThingTwo, thingThree: ThingThree)
```

Assuming that the `ThingTwo` and `ThingThree` classes are not related by inheritance, no potential ambiguity exists. Thus, the following configuration works fine, and you do not need to specify the constructor argument indexes or types explicitly in the `<constructor-arg/>` element.

```
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg ref="beanTwo"/>
        <constructor-arg ref="beanThree"/>
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as `<value>true</value>`, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

Java

```
package examples;

public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private final int years;

    // The Answer to Life, the Universe, and Everything
    private final String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Kotlin

```
package examples

class ExampleBean(
    private val years: Int, // Number of years to calculate the Ultimate Answer
    private val ultimateAnswer: String // The Answer to Life, the Universe, and
Everything
)
```

Constructor argument type matching

In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument by using the `type` attribute, as the following example shows:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

Constructor argument index

You can use the `index` attribute to specify explicitly the index of constructor arguments, as the following example shows:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type.

The index is 0-based.

Constructor argument name

You can also use the constructor parameter name for value disambiguation, as the following example shows:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

Keep in mind that, to make this work out of the box, your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you cannot or

do not want to compile your code with the debug flag, you can use the @ConstructorProperties JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

Java

```
package examples;

public class ExampleBean {

    // Fields omitted

    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Kotlin

```
package examples

class ExampleBean
@ConstructorProperties("years", "ultimateAnswer")
constructor(val years: Int, val ultimateAnswer: String)
```

Setter-based Dependency Injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument `static` factory method to instantiate your bean.

The following example shows a class that can only be dependency-injected by using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes, or annotations.

Java

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can inject a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Kotlin

```
class SimpleMovieLister {

    // a late-initialized property so that the Spring container can inject a
MovieFinder
    lateinit var movieFinder: MovieFinder

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

The `ApplicationContext` supports constructor-based and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies have already been injected through the constructor approach. You configure the dependencies in the form of a `BeanDefinition`, which you use in conjunction with `PropertyEditor` instances to convert properties from one format to another. However, most Spring users do not work with these classes directly (that is, programmatically) but rather with XML `bean` definitions, annotated components (that is, classes annotated with `@Component`, `@Controller`, and so forth), or `@Bean` methods in Java-based `@Configuration` classes. These sources are then converted internally into instances of `BeanDefinition` and used to load an entire Spring IoC container instance.

Constructor-based or setter-based DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the @Autowired annotation on a setter method can be used to make the property be a required dependency; however, constructor injection with programmatic validation of arguments is preferable.

The Spring team generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not `null`. Furthermore, constructor-injected components are always returned to the client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

Dependency Resolution Process

The container performs bean dependency resolution as follows:

- The `ApplicationContext` is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified by XML, Java code, or annotations.
- For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method (if you use that instead of a normal constructor). These dependencies are provided to the bean, when the bean is actually created.
- Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
- Each property or constructor argument that is a value is converted from its specified format to the actual type of that property or constructor argument. By default, Spring can convert a value supplied in string format to all built-in types, such as `int`, `long`, `String`, `boolean`, and so forth.

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in Bean Scopes. Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean's dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that

resolution mismatches among those dependencies may show up late — that is, on first creation of the affected bean.

Circular dependencies

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a `BeanCurrentlyInCreationException`.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken-and-egg scenario).

You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container that has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies — for example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why `ApplicationContext` implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the `ApplicationContext` is created, not later. You can still override this default behavior so that singleton beans initialize lazily, rather than being eagerly pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being injected into the dependent bean. This means that, if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if it is not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a configured init method or the InitializingBean callback method) are invoked.

Examples of Dependency Injection

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions as follows:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>

    <!-- setter injection using the neater ref attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

The following example shows the corresponding `ExampleBean` class:

Java

```
public class ExampleBean {

    private AnotherBean beanOne;

    private YetAnotherBean beanTwo;

    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

Kotlin

```
class ExampleBean {
    lateinit var beanOne: AnotherBean
    lateinit var beanTwo: YetAnotherBean
    var i: Int = 0
}
```

In the preceding example, setters are declared to match against the properties specified in the XML

file. The following example uses constructor-based DI:

```
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

    <!-- constructor injection using the neater ref attribute -->
    <constructor-arg ref="yetAnotherBean"/>

    <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

The following example shows the corresponding `ExampleBean` class:

Java

```
public class ExampleBean {

    private AnotherBean beanOne;

    private YetAnotherBean beanTwo;

    private int i;

    public ExampleBean(
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

Kotlin

```
class ExampleBean(
        private val beanOne: AnotherBean,
        private val beanTwo: YetAnotherBean,
        private val i: Int)
```

The constructor arguments specified in the bean definition are used as arguments to the constructor of the `ExampleBean`.

Now consider a variant of this example, where, instead of using a constructor, Spring is told to call a `static` factory method to return an instance of the object:

```
<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">
    <constructor-arg ref="anotherExampleBean"/>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

The following example shows the corresponding `ExampleBean` class:

Java

```
public class ExampleBean {

    // a private constructor
    private ExampleBean(...) {
        ...
    }

    // a static factory method; the arguments to this method can be
    // considered the dependencies of the bean that is returned,
    // regardless of how those arguments are actually used.
    public static ExampleBean createInstance (
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

        ExampleBean eb = new ExampleBean (...);
        // some other operations...
        return eb;
    }
}
```

Kotlin

```
class ExampleBean private constructor() {
    companion object {
        // a static factory method; the arguments to this method can be
        // considered the dependencies of the bean that is returned,
        // regardless of how those arguments are actually used.
        @JvmStatic
        fun createInstance(anotherBean: AnotherBean, yetAnotherBean: YetAnotherBean,
i: Int): ExampleBean {
            val eb = ExampleBean (...)
            // some other operations...
            return eb
        }
    }
}
```

Arguments to the `static` factory method are supplied by `<constructor-arg/>` elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the `static` factory method (although, in this example, it is). An instance (non-static) factory method can be used in an essentially identical fashion (aside from the use of the `factory-bean` attribute instead of the `class` attribute), so we do not discuss those details here.

Dependencies and Configuration in Detail

As mentioned in the previous section, you can define bean properties and constructor arguments as references to other managed beans (collaborators) or as values defined inline. Spring's XML-based configuration metadata supports sub-element types within its `<property/>` and `<constructor-arg/>` elements for this purpose.

Straight Values (Primitives, Strings, and so on)

The `value` attribute of the `<property/>` element specifies a property or constructor argument as a human-readable string representation. Spring's conversion service is used to convert these values from a `String` to the actual type of the property or argument. The following example shows various values being set:

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="misterkaoli"/>
</bean>
```

The following example uses the p-namespace for even more succinct XML configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/mydb"
        p:username="root"
        p:password="misterkaoli"/>

</beans>
```

The preceding XML is more succinct. However, typos are discovered at runtime rather than design

time, unless you use an IDE (such as IntelliJ IDEA or the Spring Tools for Eclipse) that supports automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

You can also configure a `java.util.Properties` instance, as follows:

```
<bean id="mappings"
    class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

    <!-- typed as a java.util.Properties -->
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver
            jdbc.url=jdbc:mysql://localhost:3306/mydb
        </value>
    </property>
</bean>
```

The Spring container converts the text inside the `<value/>` element into a `java.util.Properties` instance by using the JavaBeans `PropertyEditor` mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested `<value/>` element over the `value` attribute style.

The `idref` element

The `idref` element is simply an error-proof way to pass the `id` (a string value - not a reference) of another bean in the container to a `<constructor-arg/>` or `<property/>` element. The following example shows how to use it:

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean"/>
    </property>
</bean>
```

The preceding bean definition snippet is exactly equivalent (at runtime) to the following snippet:

```
<bean id="theTargetBean" class="..." />

<bean id="client" class="...">
    <property name="targetName" value="theTargetBean"/>
</bean>
```

The first form is preferable to the second, because using the `idref` tag lets the container validate at deployment time that the referenced, named bean actually exists. In the second variation, no

validation is performed on the value that is passed to the `targetName` property of the `client` bean. Typos are only discovered (with most likely fatal results) when the `client` bean is actually instantiated. If the `client` bean is a prototype bean, this typo and the resulting exception may only be discovered long after the container is deployed.

The `local` attribute on the `idref` element is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular `bean` reference any more. Change your existing `idref local` references to `idref bean` when upgrading to the 4.0 schema.

A common place (at least in versions earlier than Spring 2.0) where the `<idref/>` element brings value is in the configuration of AOP interceptors in a `ProxyFactoryBean` bean definition. Using `<idref/>` elements when you specify the interceptor names prevents you from misspelling an interceptor ID.

References to Other Beans (Collaborators)

The `ref` element is the final element inside a `<constructor-arg/>` or `<property/>` definition element. Here, you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property is to be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may already be initialized by the container.) All references are ultimately a reference to another object. Scoping and validation depend on whether you specify the ID or name of the other object through the `bean` or `parent` attribute.

Specifying the target bean through the `bean` attribute of the `<ref/>` tag is the most general form and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. The value of the `bean` attribute may be the same as the `id` attribute of the target bean or be the same as one of the values in the `name` attribute of the target bean. The following example shows how to use a `ref` element:

```
<ref bean="someBean"/>
```

Specifying the target bean through the `parent` attribute creates a reference to a bean that is in a parent container of the current container. The value of the `parent` attribute may be the same as either the `id` attribute of the target bean or one of the values in the `name` attribute of the target bean. The target bean must be in a parent container of the current one. You should use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that has the same name as the parent bean. The following pair of listings shows how to use the `parent` attribute:

```
<!-- in the parent context -->
<bean id="accountService" class="com.something.SimpleAccountService">
    <!-- insert dependencies as required here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService" <!-- bean name is the same as the parent bean -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/> <!-- notice how we refer to the parent bean -->
    </property>
    <!-- insert other configuration and dependencies as required here -->
</bean>
```

The `local` attribute on the `ref` element is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular `bean` reference any more. Change your existing `ref local` references to `ref bean` when upgrading to the 4.0 schema.

Inner Beans

A `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements defines an inner bean, as the following example shows:

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target bean
inline -->
    <property name="target">
        <bean class="com.example.Person"> <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

An inner bean definition does not require a defined ID or name. If specified, the container does not use such a value as an identifier. The container also ignores the `scope` flag on creation, because inner beans are always anonymous and are always created with the outer bean. It is not possible to access inner beans independently or to inject them into collaborating beans other than into the enclosing bean.

As a corner case, it is possible to receive destruction callbacks from a custom scope — for example, for a request-scoped inner bean contained within a singleton bean. The creation of the inner bean instance is tied to its containing bean, but destruction callbacks let it participate in the request scope's lifecycle. This is not a common scenario. Inner beans typically simply share their containing bean's scope.

Collections

The `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements set the properties and arguments of the Java `Collection` types `List`, `Set`, `Map`, and `Properties`, respectively. The following example shows how to use them:

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

The value of a map key or value, or a set value, can also be any of the following elements:

```
bean | ref | idref | list | set | map | props | value | null
```

Collection Merging

The Spring container also supports merging collections. An application developer can define a parent `<list/>`, `<map/>`, `<set/>` or `<props/>` element and have child `<list/>`, `<map/>`, `<set/>` or `<props/>` elements inherit and override values from the parent collection. That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the relevant section before continuing.

The following example demonstrates collection merging:

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
            <props>
                <prop key="administrator">administrator@example.com</prop>
                <prop key="support">support@example.com</prop>
            </props>
        </property>
    </bean>
    <bean id="child" parent="parent">
        <property name="adminEmails">
            <!-- the merge is specified on the child collection definition -->
            <props merge="true">
                <prop key="sales">sales@example.com</prop>
                <prop key="support">support@example.co.uk</prop>
            </props>
        </property>
    </bean>
<beans>
```

Notice the use of the `merge=true` attribute on the `<props/>` element of the `adminEmails` property of the `child` bean definition. When the `child` bean is resolved and instantiated by the container, the resulting instance has an `adminEmails` `Properties` collection that contains the result of merging the child's `adminEmails` collection with the parent's `adminEmails` collection. The following listing shows the result:

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

The child `Properties` collection's value set inherits all property elements from the parent `<props/>`, and the child's value for the `support` value overrides the value in the parent collection.

This merging behavior applies similarly to the `<list/>`, `<map/>`, and `<set/>` collection types. In the specific case of the `<list/>` element, the semantics associated with the `List` collection type (that is, the notion of an `ordered` collection of values) is maintained. The parent's values precede all of the child list's values. In the case of the `Map`, `Set`, and `Properties` collection types, no ordering exists. Hence, no ordering semantics are in effect for the collection types that underlie the associated `Map`, `Set`, and `Properties` implementation types that the container uses internally.

Limitations of Collection Merging

You cannot merge different collection types (such as a `Map` and a `List`). If you do attempt to do so, an appropriate `Exception` is thrown. The `merge` attribute must be specified on the lower, inherited, child definition. Specifying the `merge` attribute on a parent collection definition is redundant and does not result in the desired merging.

Strongly-typed collection

Thanks to Java's support for generic types, you can use strongly typed collections. That is, it is possible to declare a `Collection` type such that it can only contain (for example) `String` elements. If you use Spring to dependency-inject a strongly-typed `Collection` into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed `Collection` instances are converted to the appropriate type prior to being added to the `Collection`. The following Java class and bean definition show how to do so:

Java

```
public class SomeClass {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
```

Kotlin

```
class SomeClass {
    lateinit var accounts: Map<String, Float>
}
```

```
<beans>
    <bean id="something" class="x.y.SomeClass">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

When the `accounts` property of the `something` bean is prepared for injection, the generics information about the element type of the strongly-typed `Map<String, Float>` is available by reflection. Thus, Spring's type conversion infrastructure recognizes the various value elements as being of type `Float`, and the string values (`9.99`, `2.75`, and `3.99`) are converted into an actual `Float` type.

Null and Empty String Values

Spring treats empty arguments for properties and the like as empty `Strings`. The following XML-based configuration metadata snippet sets the `email` property to the empty `String` value ("").

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

The preceding example is equivalent to the following Java code:

Java

```
exampleBean.setEmail("");
```

Kotlin

```
exampleBean.email = ""
```

The `<null/>` element handles `null` values. The following listing shows an example:

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

The preceding configuration is equivalent to the following Java code:

Java

```
exampleBean.setEmail(null);
```

Kotlin

```
exampleBean.email = null
```

XML Shortcut with the p-namespace

The p-namespace lets you use the `bean` element's attributes (instead of nested `<property/>` elements) to describe your property values collaborating beans, or both.

Spring supports extensible configuration formats with namespaces, which are based on an XML Schema definition. The `beans` configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets (the first uses standard XML format and the second uses the p-namespace) that resolve to the same result:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="someone@somewhere.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="someone@somewhere.com"/>
</beans>
```

The example shows an attribute in the p-namespace called `email` in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

This example includes not only a property value using the p-namespace but also uses a special format to declare property references. Whereas the first bean definition uses `<property name="spouse" ref="jane"/>` to create a reference from bean `john` to bean `jane`, the second bean definition uses `p:spouse-ref="jane"` as an attribute to do the exact same thing. In this case, `spouse` is the property name, whereas the `-ref` part indicates that this is not a straight value but rather a reference to another bean.

The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in `Ref`, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members to avoid producing XML documents that use all three approaches at the same time.

XML Shortcut with the c-namespace

Similar to the XML Shortcut with the p-namespace, the c-namespace, introduced in Spring 3.1, allows inlined attributes for configuring the constructor arguments rather then nested `constructor-arg` elements.

The following example uses the `c:` namespace to do the same thing as the from Constructor-based Dependency Injection:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="beanTwo" class="x.y.ThingTwo"/>
    <bean id="beanThree" class="x.y.ThingThree"/>

    <!-- traditional declaration with optional argument names -->
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg name="thingTwo" ref="beanTwo"/>
        <constructor-arg name="thingThree" ref="beanThree"/>
        <constructor-arg name="email" value="something@somewhere.com"/>
    </bean>

    <!-- c-namespace declaration with argument names -->
    <bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"
        c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>
```

The `c:` namespace uses the same conventions as the `p:` one (a trailing `-ref` for bean references) for setting the constructor arguments by their names. Similarly, it needs to be declared in the XML file even though it is not defined in an XSD schema (it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), you can use fallback to the argument indexes, as follows:

```
<!-- c-namespace index declaration -->
<bean id="beanOne" class="x.y.ThingOne" c:_0-ref="beanTwo" c:_1-ref="beanThree"
    c:_2="something@somewhere.com"/>
```

Due to the XML grammar, the index notation requires the presence of the leading `_`, as XML attribute names cannot start with a number (even though some IDEs allow it). A corresponding index notation is also available for `<constructor-arg>` elements but not commonly used since the plain order of declaration is usually sufficient there.

In practice, the constructor resolution mechanism is quite efficient in matching arguments, so unless you really need to, we recommend using the name notation throughout your configuration.

Compound Property Names

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not `null`. Consider the following bean definition:

```
<bean id="something" class="things.ThingOne">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

The `something` bean has a `fred` property, which has a `bob` property, which has a `sammy` property, and that final `sammy` property is being set to a value of `123`. In order for this to work, the `fred` property of `something` and the `bob` property of `fred` must not be `null` after the bean is constructed. Otherwise, a `NullPointerException` is thrown.

Using `depends-on`

If a bean is a dependency of another bean, that usually means that one bean is set as a property of another. Typically you accomplish this with the `<ref/>` element in XML-based configuration metadata. However, sometimes dependencies between beans are less direct. An example is when a static initializer in a class needs to be triggered, such as for database driver registration. The `depends-on` attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the `depends-on` attribute to express a dependency on a single bean:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

To express a dependency on multiple beans, supply a list of bean names as the value of the `depends-on` attribute (commas, whitespace, and semicolons are valid delimiters):

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
    <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

The depends-on attribute can specify both an initialization-time dependency and, in the case of singleton beans only, a corresponding destruction-time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus, depends-on can also control shutdown order.

Lazy-initialized Beans

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as being lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element, as the following example shows:

```
<bean id="lazy" class="com.something.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.something.AnotherBean"/>
```

When the preceding configuration is consumed by an ApplicationContext, the lazy bean is not eagerly pre-instantiated when the ApplicationContext starts, whereas the not.lazy bean is eagerly pre-instantiated.

However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the singleton's dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element, as the following example shows:

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

Autowiring Collaborators

The Spring container can autowire relationships between collaborating beans. You can let Spring resolve collaborators (other beans) automatically for your bean by inspecting the contents of the `ApplicationContext`. Autowiring has the following advantages:

- Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this regard.)
- Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata (see Dependency Injection), you can specify the autowire mode for a bean definition with the `autowire` attribute of the `<bean/>` element. The autowiring functionality has four modes. You specify autowiring per bean and can thus choose which ones to autowire. The following table describes the four autowiring modes:

Table 2. Autowiring modes

Mode	Explanation
`no`	(Default) No autowiring. Bean references must be defined by `ref` elements. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
`byName`	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name and it contains a `master` property (that is, it has a `setMaster(..)` method), Spring looks for a bean definition named `master` and uses it to set the property.
`byType`	Lets a property be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use `byType` autowiring for that bean. If there are no matching beans, nothing happens (the property is not set).
`constructor`	Analogous to `byType` but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

With `byType` or `constructor` autowiring mode, you can wire arrays and typed collections. In such cases, all autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed `Map` instances if the expected key type is `String`. An autowired `Map` instance's values consist of all bean instances that match the expected type, and the `Map` instance's keys contain the corresponding bean names.

Limitations and Disadvantages of Autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

- Explicit dependencies in `property` and `constructor-arg` settings always override autowiring. You cannot autowire simple properties such as primitives, `Strings`, and `Classes` (and arrays of such simple properties). This limitation is by-design.
- Autowiring is less exact than explicit wiring. Although, as noted in the earlier table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results. The relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or `Map` instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

- Abandon autowiring in favor of explicit wiring.
- Avoid autowiring for a bean definition by setting its `autowire-candidate` attributes to `false`, as described in the next section.
- Designate a single bean definition as the primary candidate by setting the `primary` attribute of its `<bean/>` element to `true`.
- Implement the more fine-grained control available with annotation-based configuration, as described in Annotation-based Container Configuration.

Excluding a Bean from Autowiring

On a per-bean basis, you can exclude a bean from autowiring. In Spring's XML format, set the `autowire-candidate` attribute of the `<bean/>` element to `false`. The container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as `@Autowired`).

The `autowire-candidate` attribute is designed to only affect type-based autowiring. It does not affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate. As a consequence, autowiring by name nevertheless injects a bean if the name matches.

You can also limit autowire candidates based on pattern-matching against bean names. The top-level `<beans/>` element accepts one or more patterns within its `default-autowire-candidates` attribute. For example, to limit autowire candidate status to any bean whose name ends with `Repository`, provide a value of `*Repository`. To provide multiple patterns, define them in a comma-separated list. An explicit value of `true` or `false` for a bean definition's `autowire-candidate` attribute

always takes precedence. For such beans, the pattern matching rules do not apply.

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured by using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

Method Injection

In most application scenarios, most beans in the container are singletons. When a singleton bean needs to collaborate with another singleton bean or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container creates the singleton bean A only once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can make bean A aware of the container by implementing the `ApplicationContextAware` interface, and by making a `getBean("B")` call to the container ask for (a typically new) bean B instance every time bean A needs it. The following example shows this approach:

Java

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Object process(Map commandState) {
        // grab a new instance of the appropriate Command
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    protected Command createCommand() {
        // notice the Spring API dependency!
        return this.applicationContext.getBean("command", Command.class);
    }

    public void setApplicationContext(
            ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

Kotlin

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple

// Spring-API imports
import org.springframework.context.ApplicationContext
import org.springframework.context.ApplicationContextAware

class CommandManager : ApplicationContextAware {

    private lateinit var applicationContext: ApplicationContext

    fun process(commandState: Map<*, *>): Any {
        // grab a new instance of the appropriate Command
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.state = commandState
        return command.execute()
    }

    // notice the Spring API dependency!
    protected fun createCommand() =
            applicationContext.getBean("command", Command::class.java)

    override fun setApplicationContext(applicationContext: ApplicationContext) {
        this.applicationContext = applicationContext
    }
}
```

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, lets you handle this use case cleanly.

> You can read more about the motivation for Method Injection in this blog entry.

Lookup Method Injection

Lookup method injection is the ability of the container to override methods on container-managed beans and return the lookup result for another named bean in the container. The lookup typically involves a prototype bean, as in the scenario described in the preceding section. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to dynamically generate a subclass that overrides the method.

- For this dynamic subclassing to work, the class that the Spring bean container subclasses cannot be `final`, and the method to be overridden cannot be `final`, either.
- Unit-testing a class that has an `abstract` method requires you to subclass the class yourself and to supply a stub implementation of the `abstract` method.
- Concrete methods are also necessary for component scanning, which requires concrete classes to pick up.
- A further key limitation is that lookup methods do not work with factory methods and in particular not with `@Bean` methods in configuration classes, since, in that case, the container is not in charge of creating the instance and therefore cannot create a runtime-generated subclass on the fly.

In the case of the `CommandManager` class in the previous code snippet, the Spring container dynamically overrides the implementation of the `createCommand()` method. The `CommandManager` class does not have any Spring dependencies, as the reworked example shows:

Java

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

Kotlin

```
package fiona.apple

// no more Spring imports!

abstract class CommandManager {

    fun process(commandState: Any): Any {
        // grab a new instance of the appropriate Command interface
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.state = commandState
        return command.execute()
    }

    // okay... but where is the implementation of this method?
    protected abstract fun createCommand(): Command
}
```

In the client class that contains the method to be injected (the `CommandManager` in this case), the method to be injected requires a signature of the following form:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

If the method is `abstract`, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. Consider the following example:

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">
    <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="myCommand"/>
</bean>
```

The bean identified as `commandManager` calls its own `createCommand()` method whenever it needs a new instance of the `myCommand` bean. You must be careful to deploy the `myCommand` bean as a prototype if that is actually what is needed. If it is a singleton, the same instance of the `myCommand` bean is returned each time.

Alternatively, within the annotation-based component model, you can declare a lookup method through the `@Lookup` annotation, as the following example shows:

Java

```
public abstract class CommandManager {

    public Object process(Object commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup("myCommand")
    protected abstract Command createCommand();
}
```

Kotlin

```
abstract class CommandManager {

    fun process(commandState: Any): Any {
        val command = createCommand()
        command.state = commandState
        return command.execute()
    }

    @Lookup("myCommand")
    protected abstract fun createCommand(): Command
}
```

Or, more idiomatically, you can rely on the target bean getting resolved against the declared return type of the lookup method:

Java

```
public abstract class CommandManager {

    public Object process(Object commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup
    protected abstract Command createCommand();
}
```

Kotlin

```
abstract class CommandManager {

    fun process(commandState: Any): Any {
        val command = createCommand()
        command.state = commandState
        return command.execute()
    }

    @Lookup
    protected abstract fun createCommand(): Command
}
```

Note that you should typically declare such annotated lookup methods with a concrete stub implementation, in order for them to be compatible with Spring's component scanning rules where abstract classes get ignored by default. This limitation does not apply to explicitly registered or explicitly imported bean classes.

Another way of accessing differently scoped target beans is an `ObjectFactory`/`Provider` injection point. See Scoped Beans as Dependencies.

You may also find the `ServiceLocatorFactoryBean` (in the `org.springframework.beans.factory.config` package) to be useful.

Arbitrary Method Replacement

A less useful form of method injection than lookup method injection is the ability to replace arbitrary methods in a managed bean with another method implementation. You can safely skip the rest of this section until you actually need this functionality.

With XML-based configuration metadata, you can use the `replaced-method` element to replace an existing method implementation with another, for a deployed bean. Consider the following class, which has a method called `computeValue` that we want to override:

Java

```
public class MyValueCalculator {

    public String computeValue(String input) {
        // some real code...
    }

    // some other methods...
}
```

Kotlin

```
class MyValueCalculator {

    fun computeValue(input: String): String {
        // some real code...
    }

    // some other methods...
}
```

A class that implements the `org.springframework.beans.factory.support.MethodReplacer` interface provides the new method definition, as the following example shows:

Java

```
/**
 * meant to be used to override the existing computeValue(String)
 * implementation in MyValueCalculator
 */
public class ReplacementComputeValue implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

Kotlin

```
/**
 * meant to be used to override the existing computeValue(String)
 * implementation in MyValueCalculator
 */
class ReplacementComputeValue : MethodReplacer {

    override fun reimplement(obj: Any, method: Method, args: Array<out Any>): Any {
        // get the input value, work with it, and return a computed result
        val input = args[0] as String;
        ...
        return ...;
    }
}
```

The bean definition to deploy the original class and specify the method override would resemble the following example:

```
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
    <!-- arbitrary method replacement -->
    <replaced-method name="computeValue" replacer="replacementComputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

You can use one or more `<arg-type/>` elements within the `<replaced-method/>` element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match `java.lang.String`:

```
java.lang.String
String
Str
```

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by letting you type only the shortest string that matches an argument type.

2.1.5. Bean Scopes

When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition but also control the scope of the objects created from a particular bean definition. This approach is powerful and flexible, because you can choose the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes. The Spring Framework supports six scopes, four of which are available only if you use a web-aware `ApplicationContext`. You can also create a custom scope.

The following table describes the supported scopes:

Table 3. Bean scopes

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.

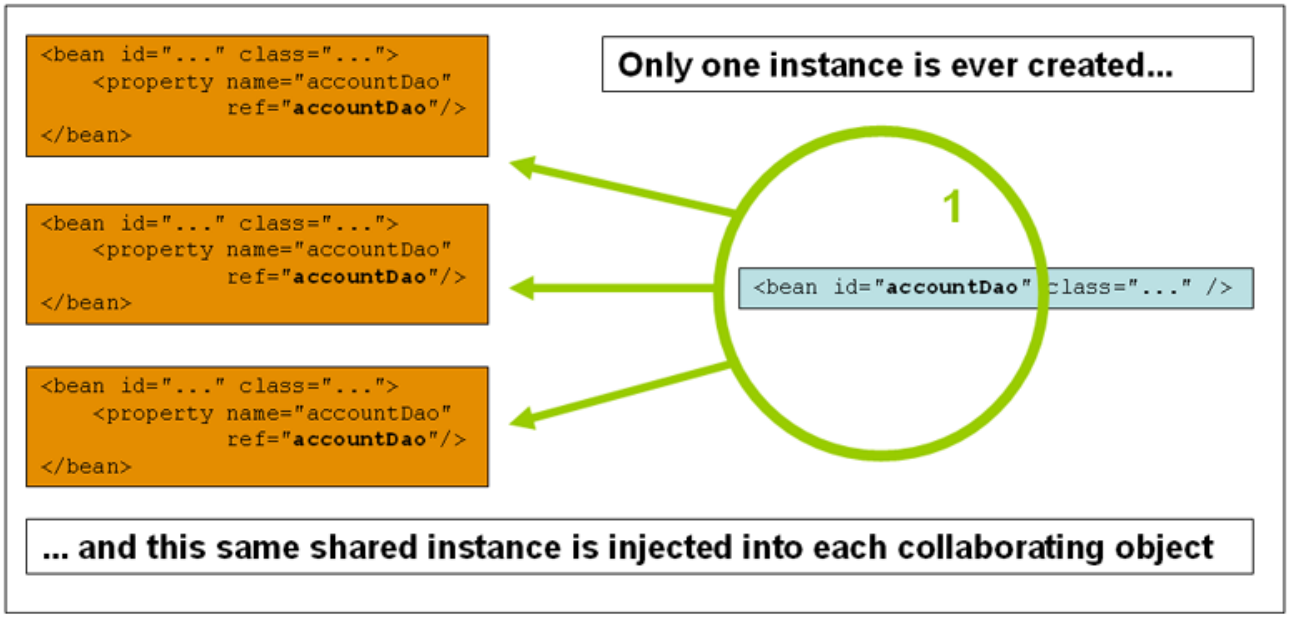
Scope	Description
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

As of Spring 3.0, a thread scope is available but is not registered by default. For more information, see the documentation for `SimpleThreadScope`. For instructions on how to register this or any other custom scope, see Using a Custom Scope.

The Singleton Scope

Only one shared instance of a singleton bean is managed, and all requests for beans with an ID or IDs that match that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. The following image shows how the singleton scope works:



Spring's concept of a singleton bean differs from the singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF singleton hard-codes the scope of an object such that one and

only one instance of a particular class is created per ClassLoader. The scope of the Spring singleton is best described as being per-container and per-bean. This means that, if you define one bean for a particular class in a single Spring container, the Spring container creates one and only one instance of the class defined by that bean definition. The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you can define a bean as shown in the following example:

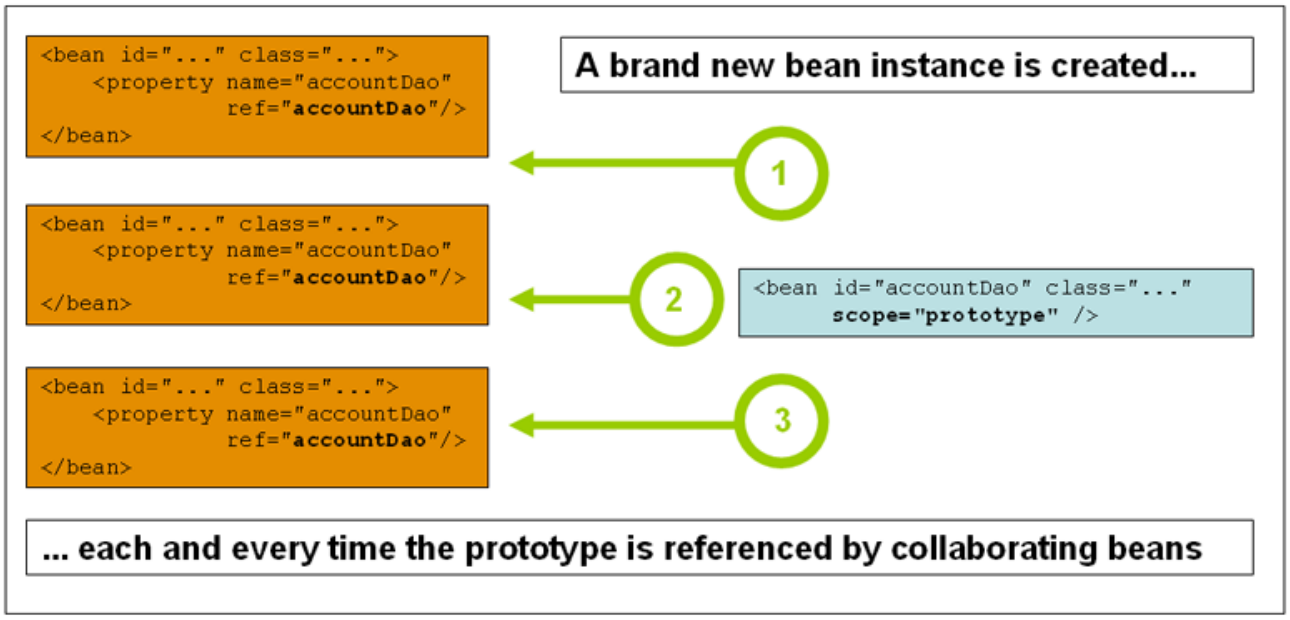
```
<bean id="accountService" class="com.something.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default)
-->
<bean id="accountService" class="com.something.DefaultAccountService"
scope="singleton"/>
```

The Prototype Scope

The non-singleton prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a `getBean()` method call on the container. As a rule, you should use the prototype scope for all stateful beans and the singleton scope for stateless beans.

The following diagram illustrates the Spring prototype scope:



(A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state. It was easier for us to reuse the core of the singleton diagram.)

The following example defines a bean as a prototype in XML:

```
<bean id="accountService" class="com.something.DefaultAccountService"
scope="prototype"/>
```

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean.

The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype beans hold. To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java `new` operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see Lifecycle Callbacks.)

Singleton Beans with Prototype-bean Dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus, if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container instantiates the singleton bean and resolves and injects its dependencies. If you need a new instance of a prototype bean at runtime more than once, see Method Injection.

Request, Session, Application, and WebSocket Scopes

The `request`, `session`, `application`, and `websocket` scopes are available only if you use a web-aware Spring `ApplicationContext` implementation (such as `XmlWebApplicationContext`). If you use these scopes with regular Spring IoC containers, such as the `ClassPathXmlApplicationContext`, an `IllegalStateException` that complains about an unknown bean scope is thrown.

Initial Web Configuration

To support the scoping of beans at the `request`, `session`, `application`, and `websocket` levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes: `singleton` and `prototype`.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring `DispatcherServlet`, no special setup is necessary. `DispatcherServlet` already exposes all relevant state.

If you use a Servlet web container, with requests processed outside of Spring's `DispatcherServlet` (for example, when using JSF), you need to register the `org.springframework.web.context.request.RequestContextListener ServletRequestListener`. This can be done programmatically by using the `WebApplicationInitializer` interface. Alternatively, add the following declaration to your web application's `web.xml` file:

```
<web-app>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

Alternatively, if there are issues with your listener setup, consider using Spring's `RequestContextFilter`. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate. The following listing shows the filter part of a web application:

```
<web-app>
    ...
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-
class>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

`DispatcherServlet`, `RequestContextListener`, and `RequestContextFilter` all do exactly the same thing, namely bind the HTTP request object to the `Thread` that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

Request scope

Consider the following XML configuration for a bean definition:

```
<bean id="loginAction" class="com.something.LoginAction" scope="request"/>
```

The Spring container creates a new instance of the `LoginAction` bean by using the `loginAction` bean definition for each and every HTTP request. That is, the `loginAction` bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same `loginAction` bean definition do not see these changes in state. They are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java configuration, the `@RequestScope` annotation can

be used to assign a component to the `request` scope. The following example shows how to do so:

Java

```
@RequestScope
@Component
public class LoginAction {
    // ...
}
```

Kotlin

```
@RequestScope
@Component
class LoginAction {
    // ...
}
```

Session Scope

Consider the following XML configuration for a bean definition:

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session"/>
```

The Spring container creates a new instance of the `UserPreferences` bean by using the `userPreferences` bean definition for the lifetime of a single HTTP `Session`. In other words, the `userPreferences` bean is effectively scoped at the HTTP `Session` level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP `Session` instances that are also using instances created from the same `userPreferences` bean definition do not see these changes in state, because they are particular to an individual HTTP `Session`. When the HTTP `Session` is eventually discarded, the bean that is scoped to that particular HTTP `Session` is also discarded.

When using annotation-driven components or Java configuration, you can use the `@SessionScope` annotation to assign a component to the `session` scope.

Java

```
@SessionScope
@Component
public class UserPreferences {
    // ...
}
```

Kotlin

```
@SessionScope
@Component
class UserPreferences {
    // ...
}
```

Application Scope

Consider the following XML configuration for a bean definition:

```
<bean id="appPreferences" class="com.something.AppPreferences" scope="application"/>
```

The Spring container creates a new instance of the `AppPreferences` bean by using the `appPreferences` bean definition once for the entire web application. That is, the `appPreferences` bean is scoped at the `ServletContext` level and stored as a regular `ServletContext` attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per `ServletContext`, not per Spring `ApplicationContext` (for which there may be several in any given web application), and it is actually exposed and therefore visible as a `ServletContext` attribute.

When using annotation-driven components or Java configuration, you can use the `@ApplicationScope` annotation to assign a component to the `application` scope. The following example shows how to do so:

Java

```
@ApplicationScope
@Component
public class AppPreferences {
    // ...
}
```

Kotlin

```
@ApplicationScope
@Component
class AppPreferences {
    // ...
}
```

WebSocket Scope

WebSocket scope is associated with the lifecycle of a WebSocket session and applies to STOMP over WebSocket applications, see WebSocket scope for more details.

Scoped Beans as Dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request-scoped bean into another bean of a longer-lived scope, you may choose to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real target object from the relevant scope (such as an HTTP request) and delegate method calls onto the real object.

You may also use `<aop:scoped-proxy/>` between beans that are scoped as `singleton`, with the reference then going through an intermediate proxy that is serializable and therefore able to re-obtain the target singleton bean on deserialization.

When declaring `<aop:scoped-proxy/>` against a bean of scope `prototype`, every method call on the shared proxy leads to the creation of a new target instance to which the call is then being forwarded.

Also, scoped proxies are not the only way to access beans from shorter scopes in a lifecycle-safe fashion. You may also declare your injection point (that is, the constructor or setter argument or autowired field) as `ObjectFactory<MyTargetBean>`, allowing for a `getObject()` call to retrieve the current instance on demand every time it is needed — without holding on to the instance or storing it separately.

As an extended variant, you may declare `ObjectProvider<MyTargetBean>` which delivers several additional access variants, including `getIfAvailable` and `getIfUnique`.

The JSR-330 variant of this is called `Provider` and is used with a `Provider<MyTargetBean>` declaration and a corresponding `get()` call for every retrieval attempt. See here for more details on JSR-330 overall.

The configuration in the following example is only one line, but it is important to understand the “why” as well as the “how” behind it:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.something.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/> ①
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.something.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

① The line that defines the proxy.

To create such a proxy, you insert a child `<aop:scoped-proxy/>` element into a scoped bean definition (see Choosing the Type of Proxy to Create and XML Schema-based configuration). Why do definitions of beans scoped at the `request`, `session` and custom-scope levels require the `<aop:scoped-proxy/>` element? Consider the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes (note that the following `userPreferences` bean definition as it stands is incomplete):

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session"/>

<bean id="userManager" class="com.something.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

In the preceding example, the singleton bean (`userManager`) is injected with a reference to the HTTP `Session`-scoped bean (`userPreferences`). The salient point here is that the `userManager` bean is a singleton: it is instantiated exactly once per container, and its dependencies (in this case only one, the `userPreferences` bean) are also injected only once. This means that the `userManager` bean operates only on the exact same `userPreferences` object (that is, the one with which it was originally injected).

This is not the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean (for example, injecting an HTTP `Session`-scoped collaborating bean as a dependency into singleton bean). Rather, you need a single `userManager` object, and, for the lifetime of an HTTP `Session`, you need a `userPreferences` object that is specific to the HTTP `Session`. Thus, the container

creates an object that exposes the exact same public interface as the `UserPreferences` class (ideally an object that is a `UserPreferences` instance), which can fetch the real `UserPreferences` object from the scoping mechanism (HTTP request, `Session`, and so forth). The container injects this proxy object into the `userManager` bean, which is unaware that this `UserPreferences` reference is a proxy. In this example, when a `UserManager` instance invokes a method on the dependency-injected `UserPreferences` object, it is actually invoking a method on the proxy. The proxy then fetches the real `UserPreferences` object from (in this case) the HTTP `Session` and delegates the method invocation onto the retrieved real `UserPreferences` object.

Thus, you need the following (correct and complete) configuration when injecting `request-` and `session-scoped` beans into collaborating objects, as the following example shows:

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.something.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

Choosing the Type of Proxy to Create

By default, when the Spring container creates a proxy for a bean that is marked up with the `<aop:scoped-proxy/>` element, a CGLIB-based class proxy is created.

CGLIB proxies intercept only public method calls! Do not call non-public methods on such a proxy. They are not delegated to the actual scoped target object.

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying `false` for the value of the `proxy-target-class` attribute of the `<aop:scoped-proxy/>` element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to affect such proxying. However, it also means that the class of the scoped bean must implement at least one interface and that all collaborators into which the scoped bean is injected must reference the bean through one of its interfaces. The following example shows a proxy based on an interface:

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.stuff.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.stuff.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

For more detailed information about choosing class-based or interface-based proxying, see Proxying Mechanisms.

Custom Scopes

The bean scoping mechanism is extensible. You can define your own scopes or even redefine existing scopes, although the latter is considered bad practice and you cannot override the built-in `singleton` and `prototype` scopes.

Creating a Custom Scope

To integrate your custom scopes into the Spring container, you need to implement the `org.springframework.beans.factory.config.Scope` interface, which is described in this section. For an idea of how to implement your own scopes, see the `Scope` implementations that are supplied with the Spring Framework itself and the `Scope` javadoc, which explains the methods you need to implement in more detail.

The `Scope` interface has four methods to get objects from the scope, remove them from the scope, and let them be destroyed.

The session scope implementation, for example, returns the session-scoped bean (if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference). The following method returns the object from the underlying scope:

Java

```
Object get(String name, ObjectFactory<?> objectFactory)
```

Kotlin

```
fun get(name: String, objectFactory: ObjectFactory<*>): Any
```

The session scope implementation, for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return `null` if the object with the specified name is not found. The following method removes the object from the underlying scope:

Java

```
Object remove(String name)
```

Kotlin

```
fun remove(name: String): Any
```

The following method registers a callback that the scope should invoke when it is destroyed or when the specified object in the scope is destroyed:

Java

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

Kotlin

```
fun registerDestructionCallback(name: String, destructionCallback: Runnable)
```

See the javadoc or a Spring scope implementation for more information on destruction callbacks.

The following method obtains the conversation identifier for the underlying scope:

Java

```
String getConversationId()
```

Kotlin

```
fun getConversationId(): String
```

This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

Using a Custom Scope

After you write and test one or more custom `Scope` implementations, you need to make the Spring container aware of your new scopes. The following method is the central method to register a new `Scope` with the Spring container:

Java

```
void registerScope(String scopeName, Scope scope);
```

Kotlin

```
fun registerScope(scopeName: String, scope: Scope)
```

This method is declared on the `ConfigurableBeanFactory` interface, which is available through the `BeanFactory` property on most of the concrete `ApplicationContext` implementations that ship with Spring.

The first argument to the `registerScope(..)` method is the unique name associated with a scope. Examples of such names in the Spring container itself are `singleton` and `prototype`. The second argument to the `registerScope(..)` method is an actual instance of the custom `Scope` implementation that you wish to register and use.

Suppose that you write your custom `Scope` implementation, and then register it as shown in the next example.

The next example uses `SimpleThreadScope`, which is included with Spring but is not registered by default. The instructions would be the same for your own custom `Scope` implementations.

Java

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

Kotlin

```
val threadScope = SimpleThreadScope()
beanFactory.registerScope("thread", threadScope)
```

You can then create bean definitions that adhere to the scoping rules of your custom `Scope`, as follows:

```
<bean id="..." class="..." scope="thread">
```

With a custom `Scope` implementation, you are not limited to programmatic registration of the scope. You can also do the `Scope` registration declaratively, by using the `CustomScopeConfigurer` class, as the following example shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean
class="org.springframework.context.support.SimpleThreadScope"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="thing2" class="x.y.Thing2" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="thing1" class="x.y.Thing1">
        <property name="thing2" ref="thing2"/>
    </bean>

</beans>
```

When you place `<aop:scoped-proxy/>` within a `<bean>` declaration for a `FactoryBean` implementation, it is the factory bean itself that is scoped, not the object returned from `getObject()`.

2.1.6. Customizing the Nature of a Bean

The Spring Framework provides a number of interfaces you can use to customize the nature of a bean. This section groups them as follows:

- Lifecycle Callbacks
- `ApplicationContextAware` and `BeanNameAware`
- Other `Aware` Interfaces

Lifecycle Callbacks

To interact with the container's management of the bean lifecycle, you can implement the Spring `InitializingBean` and `DisposableBean` interfaces. The container calls `afterPropertiesSet()` for the

former and `destroy()` for the latter to let the bean perform certain actions upon initialization and destruction of your beans.

The JSR-250 `@PostConstruct` and `@PreDestroy` annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring-specific interfaces. For details, see Using `@PostConstruct` and `@PreDestroy`.

If you do not want to use the JSR-250 annotations but you still want to remove coupling, consider `init-method` and `destroy-method` bean definition metadata.

Internally, the Spring Framework uses `BeanPostProcessor` implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring does not by default offer, you can implement a `BeanPostProcessor` yourself. For more information, see Container Extension Points.

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the `Lifecycle` interface so that those objects can participate in the startup and shutdown process, as driven by the container's own lifecycle.

The lifecycle callback interfaces are described in this section.

Initialization Callbacks

The `org.springframework.beans.factory.InitializingBean` interface lets a bean perform initialization work after the container has set all necessary properties on the bean. The `InitializingBean` interface specifies a single method:

```
void afterPropertiesSet() throws Exception;
```

We recommend that you do not use the `InitializingBean` interface, because it unnecessarily couples the code to Spring. Alternatively, we suggest using the `@PostConstruct` annotation or specifying a POJO initialization method. In the case of XML-based configuration metadata, you can use the `init-method` attribute to specify the name of the method that has a void no-argument signature. With Java configuration, you can use the `initMethod` attribute of `@Bean`. See Receiving Lifecycle Callbacks. Consider the following example:

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

Java

```
public class ExampleBean {

    public void init() {
        // do some initialization work
    }
}
```

Kotlin

```
class ExampleBean {

    fun init() {
        // do some initialization work
    }
}
```

The preceding example has almost exactly the same effect as the following example (which consists of two listings):

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

Java

```
public class AnotherExampleBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

Kotlin

```
class AnotherExampleBean : InitializingBean {

    override fun afterPropertiesSet() {
        // do some initialization work
    }
}
```

However, the first of the two preceding examples does not couple the code to Spring.

Destruction Callbacks

Implementing the `org.springframework.beans.factory.DisposableBean` interface lets a bean get a callback when the container that contains it is destroyed. The `DisposableBean` interface specifies a single method:

```
void destroy() throws Exception;
```

We recommend that you do not use the `DisposableBean` callback interface, because it unnecessarily couples the code to Spring. Alternatively, we suggest using the `@PreDestroy` annotation or specifying a generic method that is supported by bean definitions. With XML-based configuration metadata, you can use the `destroy-method` attribute on the `<bean/>`. With Java configuration, you can use the

`destroyMethod` attribute of `@Bean`. See Receiving Lifecycle Callbacks. Consider the following definition:

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

Java

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

Kotlin

```
class ExampleBean {

    fun cleanup() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

The preceding definition has almost exactly the same effect as the following definition:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

Java

```
public class AnotherExampleBean implements DisposableBean {

    @Override
    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

Kotlin

```
class AnotherExampleBean : DisposableBean {

    override fun destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

However, the first of the two preceding definitions does not couple the code to Spring.

You can assign the `destroy-method` attribute of a `<bean>` element a special `(inferred)` value, which instructs Spring to automatically detect a public `close` or `shutdown` method on the specific bean class. (Any class that implements `java.lang.AutoCloseable` or `java.io.Closeable` would therefore match.) You can also set this special `(inferred)` value on the `default-destroy-method` attribute of a `<beans>` element to apply this behavior to an entire set of beans (see Default Initialization and Destroy Methods). Note that this is the default behavior with Java configuration.

Default Initialization and Destroy Methods

When you write initialization and destroy method callbacks that do not use the Spring-specific `InitializingBean` and `DisposableBean` callback interfaces, you typically write methods with names such as `init()`, `initialize()`, `dispose()`, and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

You can configure the Spring container to "look" for named initialization and destroy callback method names on every bean. This means that you, as an application developer, can write your application classes and use an initialization callback called `init()`, without having to configure an `init-method="init"` attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Suppose that your initialization callback methods are named `init()` and your destroy callback methods are named `destroy()`. Your class then resembles the class in the following example:

Java

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }
}
```

Kotlin

```
class DefaultBlogService : BlogService {

    private var blogDao: BlogDao? = null

    // this is (unsurprisingly) the initialization callback method
    fun init() {
        if (blogDao == null) {
            throw IllegalStateException("The [blogDao] property must be set.")
        }
    }
}
```

You could then use that class in a bean resembling the following:

```
<beans default-init-method="init">

    <bean id="blogService" class="com.something.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

The presence of the `default-init-method` attribute on the top-level `<beans/>` element attribute causes the Spring IoC container to recognize a method called `init` on the bean class as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

You can configure destroy method callbacks similarly (in XML, that is) by using the `default-destroy-method` attribute on the top-level `<beans/>` element.

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name by using the `init-method` and `destroy-method` attributes of the `<bean/>` itself.

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus, the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created first and then an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the `init` method, because doing so would couple the lifecycle of the target bean to its proxy or interceptors and leave strange semantics when your code interacts directly with the raw target bean.

Combining Lifecycle Mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior:

- The `InitializingBean` and `DisposableBean` callback interfaces
- Custom `init()` and `destroy()` methods
- The `@PostConstruct` and `@PreDestroy` annotations. You can combine these mechanisms to control a given bean.

If multiple lifecycle mechanisms are configured for a bean and each mechanism is configured with a different method name, then each configured method is run in the order listed after this note. However, if the same method name is configured — for example, `init()` for an initialization method — for more than one of these lifecycle mechanisms, that method is run once, as explained in the preceding section.

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

1. Methods annotated with `@PostConstruct`
2. `afterPropertiesSet()` as defined by the `InitializingBean` callback interface
3. A custom configured `init()` method

Destroy methods are called in the same order:

1. Methods annotated with `@PreDestroy`
2. `destroy()` as defined by the `DisposableBean` callback interface
3. A custom configured `destroy()` method

Startup and Shutdown Callbacks

The `Lifecycle` interface defines the essential methods for any object that has its own lifecycle requirements (such as starting and stopping some background process):

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();
}
```

Any Spring-managed object may implement the `Lifecycle` interface. Then, when the `ApplicationContext` itself receives start and stop signals (for example, for a stop/restart scenario at runtime), it cascades those calls to all `Lifecycle` implementations defined within that context. It does this by delegating to a `LifecycleProcessor`, shown in the following listing:

```
public interface LifecycleProcessor extends Lifecycle {

    void onRefresh();

    void onClose();
}
```

Notice that the `LifecycleProcessor` is itself an extension of the `Lifecycle` interface. It also adds two other methods for reacting to the context being refreshed and closed.

Note that the regular `org.springframework.context.Lifecycle` interface is a plain contract for explicit start and stop notifications and does not imply auto-startup at context refresh time. For fine-grained control over auto-startup of a specific bean (including startup phases), consider implementing `org.springframework.context.SmartLifecycle` instead.

Also, please note that stop notifications are not guaranteed to come before destruction. On regular shutdown, all `Lifecycle` beans first receive a stop notification before the general destruction callbacks are being propagated. However, on hot refresh during a context's lifetime or on stopped refresh attempts, only destroy methods are called.

The order of startup and shutdown invocations can be important. If a "depends-on" relationship exists between any two objects, the dependent side starts after its dependency, and it stops before its dependency. However, at times, the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the `SmartLifecycle` interface defines another option, namely the `getPhase()` method as defined on its super-interface, `Phased`. The following listing shows the definition of the `Phased` interface:

```
public interface Phased {

    int getPhase();
}
```

The following listing shows the definition of the `SmartLifecycle` interface:

```
public interface SmartLifecycle extends Lifecycle, Phased {

    boolean isAutoStartup();

    void stop(Runnable callback);
}
```

When starting, the objects with the lowest phase start first. When stopping, the reverse order is followed. Therefore, an object that implements `SmartLifecycle` and whose `getPhase()` method returns `Integer.MIN_VALUE` would be among the first to start and the last to stop. At the other end of

the spectrum, a phase value of `Integer.MAX_VALUE` would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it is also important to know that the default phase for any "normal" `Lifecycle` object that does not implement `SmartLifecycle` is `0`. Therefore, any negative phase value indicates that an object should start before those standard components (and stop after them). The reverse is true for any positive phase value.

The stop method defined by `SmartLifecycle` accepts a callback. Any implementation must invoke that callback's `run()` method after that implementation's shutdown process is complete. That enables asynchronous shutdown where necessary, since the default implementation of the `LifecycleProcessor` interface, `DefaultLifecycleProcessor`, waits up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named `lifecycleProcessor` within the context. If you want only to modify the timeout, defining the following would suffice:

```
<bean id="lifecycleProcessor"
class="org.springframework.context.support.DefaultLifecycleProcessor">
    <!-- timeout value in milliseconds -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

As mentioned earlier, the `LifecycleProcessor` interface defines callback methods for the refreshing and closing of the context as well. The latter drives the shutdown process as if `stop()` had been called explicitly, but it happens when the context is closing. The 'refresh' callback, on the other hand, enables another feature of `SmartLifecycle` beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback is invoked. At that point, the default lifecycle processor checks the boolean value returned by each `SmartLifecycle` object's `isAutoStartup()` method. If `true`, that object is started at that point rather than waiting for an explicit invocation of the context's or its own `start()` method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The `phase` value and any "depends-on" relationships determine the startup order as described earlier.

Shutting Down the Spring IoC Container Gracefully in Non-Web Applications

This section applies only to non-web applications. Spring's web-based `ApplicationContext` implementations already have code in place to gracefully shut down the Spring IoC container when the relevant web application is shut down.

If you use Spring's IoC container in a non-web application environment (for example, in a rich client desktop environment), register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, call the `registerShutdownHook()` method that is declared on the `ConfigurableApplicationContext` interface, as the following example shows:

Java

```
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ConfigurableApplicationContext ctx = new
ClassPathXmlApplicationContext("beans.xml");

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...
    }
}
```

Kotlin

```
import org.springframework.context.support.ClassPathXmlApplicationContext

fun main() {
    val ctx = ClassPathXmlApplicationContext("beans.xml")

    // add a shutdown hook for the above context...
    ctx.registerShutdownHook()

    // app runs here...

    // main method exits, hook is called prior to the app shutting down...
}
```

`ApplicationContextAware` and `BeanNameAware`

When an `ApplicationContext` creates an object instance that implements the `org.springframework.context.ApplicationContextAware` interface, the instance is provided with a reference to that `ApplicationContext`. The following listing shows the definition of the `ApplicationContextAware` interface:

```
public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext applicationContext) throws
BeansException;
}
```

Thus, beans can programmatically manipulate the `ApplicationContext` that created them, through the `ApplicationContext` interface or by casting the reference to a known subclass of this interface (such as `ConfigurableApplicationContext`, which exposes additional functionality). One use would be the programmatic retrieval of other beans. Sometimes this capability is useful. However, in general, you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the `ApplicationContext` provide access to file resources, publishing application events, and accessing a `MessageSource`. These additional features are described in Additional Capabilities of the `ApplicationContext`.

Autowiring is another alternative to obtain a reference to the `ApplicationContext`. The *traditional* `constructor` and `byType` autowiring modes (as described in Autowiring Collaborators) can provide a dependency of type `ApplicationContext` for a constructor argument or a setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the annotation-based autowiring features. If you do, the `ApplicationContext` is autowired into a field, constructor argument, or method parameter that expects the `ApplicationContext` type if the field, constructor, or method in question carries the `@Autowired` annotation. For more information, see Using `@Autowired`.

When an `ApplicationContext` creates a class that implements the `org.springframework.beans.factory.BeanNameAware` interface, the class is provided with a reference to the name defined in its associated object definition. The following listing shows the definition of the BeanNameAware interface:

```
public interface BeanNameAware {

    void setBeanName(String name) throws BeansException;
}
```

The callback is invoked after population of normal bean properties but before an initialization callback such as `InitializingBean.afterPropertiesSet()` or a custom init-method.

Other `Aware` Interfaces

Besides `ApplicationContextAware` and `BeanNameAware` (discussed earlier), Spring offers a wide range of `Aware` callback interfaces that let beans indicate to the container that they require a certain infrastructure dependency. As a general rule, the name indicates the dependency type. The following table summarizes the most important `Aware` interfaces:

Table 4. Aware interfaces

Name	Injected Dependency	Explained in...
`ApplicationContextAware`	Declaring `ApplicationContext`.	`ApplicationContextAware` and `BeanNameAware`
`ApplicationEventPublisherAware`	Event publisher of the enclosing `ApplicationContext`.	Additional Capabilities of the `ApplicationContext`

Name	Injected Dependency	Explained in...
`BeanClassLoaderAware`	Class loader used to load the bean classes.	Instantiating Beans
`BeanFactoryAware`	Declaring `BeanFactory`.	The `BeanFactory` API
`BeanNameAware`	Name of the declaring bean.	`ApplicationContextAware` and `BeanNameAware`
`LoadTimeWeaverAware`	Defined weaver for processing class definition at load time.	Load-time Weaving with AspectJ in the Spring Framework
`MessageSourceAware`	Configured strategy for resolving messages (with support for parametrization and internationalization).	Additional Capabilities of the `ApplicationContext`
`NotificationPublisherAware`	Spring JMX notification publisher.	Notifications
`ResourceLoaderAware`	Configured loader for low-level access to resources.	Resources
`ServletConfigAware`	Current `ServletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`.	Spring MVC
`ServletContextAware`	Current `ServletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`.	Spring MVC

Note again that using these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As a result, we recommend them for infrastructure beans that require programmatic access to the container.

2.1.7. Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information, such as the initialization method, a static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values or add others as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

If you work with an `ApplicationContext` interface programmatically, child bean definitions are represented by the `ChildBeanDefinition` class. Most users do not work with them on this level. Instead, they configure bean definitions declaratively in a class such as the `ClassPathXmlApplicationContext`. When you use XML-based configuration metadata, you can indicate a child bean definition by using the `parent` attribute, specifying the parent bean as the value of this attribute. The following example shows how to do so:

```
<bean id="inheritedTestBean" abstract="true"
        class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
        class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBean" init-method="initialize">  ①
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>
```

① Note the `parent` attribute.

A child bean definition uses the bean class from the parent definition if none is specified but can also override it. In the latter case, the child bean class must be compatible with the parent (that is, it must accept the parent's property values).

A child bean definition inherits scope, constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any scope, initialization method, destroy method, or `static` factory method settings that you specify override the corresponding parent settings.

The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, and lazy init.

The preceding example explicitly marks the parent bean definition as abstract by using the `abstract` attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as `abstract` is required, as the following example shows:

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBeanWithoutClass" init-method="initialize">
    <property name="name" value="override"/>
    <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as `abstract`. When a definition is `abstract`, it is usable only as a pure template bean definition that serves as a parent definition for child definitions. Trying to use such an `abstract` parent bean on its own, by referring to it as a ref property of another bean or doing an explicit `getBean()` call with the parent bean ID returns an error. Similarly, the container's internal `preInstantiateSingletons()` method ignores bean definitions that are defined as abstract.

`ApplicationContext` pre-instantiates all singletons by default. Therefore, it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually (attempt to) pre-instantiate the `abstract` bean.

2.1.8. Container Extension Points

Typically, an application developer does not need to subclass `ApplicationContext` implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.

Customizing Beans by Using a `BeanPostProcessor`

The `BeanPostProcessor` interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more custom `BeanPostProcessor` implementations.

You can configure multiple `BeanPostProcessor` instances, and you can control the order in which these `BeanPostProcessor` instances run by setting the `order` property. You can set this property only if the `BeanPostProcessor` implements the `Ordered` interface. If you write your own `BeanPostProcessor`, you should consider implementing the `Ordered` interface, too. For further details, see the javadoc of the `BeanPostProcessor` and `Ordered` interfaces. See also the note on programmatic registration of `BeanPostProcessor` instances.

`BeanPostProcessor` instances operate on bean (or object) instances. That is, the Spring IoC container instantiates a bean instance and then `BeanPostProcessor` instances do their work.

`BeanPostProcessor` instances are scoped per-container. This is relevant only if you use container hierarchies. If you define a `BeanPostProcessor` in one container, it post-processes only the beans in that container. In other words, beans that are defined in one container are not post-processed by a `BeanPostProcessor` defined in another container, even if both containers are part of the same hierarchy.

To change the actual bean definition (that is, the blueprint that defines the bean), you instead need to use a `BeanFactoryPostProcessor`, as described in Customizing Configuration Metadata with a `BeanFactoryPostProcessor`.

The `org.springframework.beans.factory.config.BeanPostProcessor` interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container, for each bean instance that is created by the container, the post-processor gets a callback from the container both before container initialization methods (such as `InitializingBean.afterPropertiesSet()` or any declared `init` method) are called, and after any bean initialization callbacks. The post-processor can take any action with the bean instance, including ignoring the callback completely. A bean post-processor typically checks for callback interfaces, or it may wrap a bean with a proxy. Some Spring

AOP infrastructure classes are implemented as bean post-processors in order to provide proxy-wrapping logic.

An `ApplicationContext` automatically detects any beans that are defined in the configuration metadata that implement the `BeanPostProcessor` interface. The `ApplicationContext` registers these beans as post-processors so that they can be called later, upon bean creation. Bean post-processors can be deployed in the container in the same fashion as any other beans.

Note that, when declaring a `BeanPostProcessor` by using an `@Bean` factory method on a configuration class, the return type of the factory method should be the implementation class itself or at least the `org.springframework.beans.factory.config.BeanPostProcessor` interface, clearly indicating the post-processor nature of that bean. Otherwise, the `ApplicationContext` cannot autodetect it by type before fully creating it. Since a `BeanPostProcessor` needs to be instantiated early in order to apply to the initialization of other beans in the context, this early type detection is critical.

Programmatically registering `BeanPostProcessor` *instances*

While the recommended approach for `BeanPostProcessor` registration is through `ApplicationContext` auto-detection (as described earlier), you can register them programmatically against a `ConfigurableBeanFactory` by using the `addBeanPostProcessor` method. This can be useful when you need to evaluate conditional logic before registration or even for copying bean post processors across contexts in a hierarchy. Note, however, that `BeanPostProcessor` instances added programmatically do not respect the `Ordered` interface. Here, it is the order of registration that dictates the order of execution. Note also that `BeanPostProcessor` instances registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering.

`BeanPostProcessor` *instances and AOP auto-proxying*

Classes that implement the `BeanPostProcessor` interface are special and are treated differently by the container. All `BeanPostProcessor` instances and beans that they directly reference are instantiated on startup, as part of the special startup phase of the `ApplicationContext`. Next, all `BeanPostProcessor` instances are registered in a sorted fashion and applied to all further beans in the container. Because AOP auto-proxying is implemented as a `BeanPostProcessor` itself, neither `BeanPostProcessor` instances nor the beans they directly reference are eligible for auto-proxying and, thus, do not have aspects woven into them.

For any such bean, you should see an informational log message: `Bean someBean is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)`.

If you have beans wired into your `BeanPostProcessor` by using autowiring or `@Resource` (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates and, therefore, make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with `@Resource` where the field or setter name does not directly correspond to the declared name of a bean and no name attribute is used, Spring accesses other beans for matching them by type.

The following examples show how to write, register, and use `BeanPostProcessor` instances in an `ApplicationContext`.

Example: Hello World, `BeanPostProcessor`-style

This first example illustrates basic usage. The example shows a custom `BeanPostProcessor` implementation that invokes the `toString()` method of each bean as it is created by the container and prints the resulting string to the system console.

The following listing shows the custom `BeanPostProcessor` implementation class definition:

Java

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }
}
```

Kotlin

```
import org.springframework.beans.factory.config.BeanPostProcessor

class InstantiationTracingBeanPostProcessor : BeanPostProcessor {

    // simply return the instantiated bean as-is
    override fun postProcessBeforeInitialization(bean: Any, beanName: String): Any? {
        return bean // we could potentially return any object reference here...
    }

    override fun postProcessAfterInitialization(bean: Any, beanName: String): Any? {
        println("Bean '$beanName' created : $bean")
        return bean
    }
}
```

The following `beans` element uses the `InstantiationTracingBeanPostProcessor`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
        https://www.springframework.org/schema/lang/spring-lang.xsd">

    <lang:groovy id="messenger"
            script-
source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
    when the above bean (messenger) is instantiated, this custom
    BeanPostProcessor implementation will output the fact to the system console
    -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the `InstantiationTracingBeanPostProcessor` is merely defined. It does not even have a name, and, because it is a bean, it can be dependency-injected as you would any other bean. (The preceding configuration also defines a bean that is backed by a Groovy script. The Spring dynamic language support is detailed in the chapter entitled Dynamic Language Support.)

The following Java application runs the preceding code and configuration:

Java

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("scripting/beans.xml");
        Messenger messenger = ctx.getBean("messenger", Messenger.class);
        System.out.println(messenger);
    }

}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = ClassPathXmlApplicationContext("scripting/beans.xml")
    val messenger = ctx.getBean<Messenger>("messenger")
    println(messenger)
}
```

The output of the preceding application resembles the following:

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961
org.springframework.scripting.groovy.GroovyMessenger@272961
```

Example: The `AutowiredAnnotationBeanPostProcessor`

Using callback interfaces or annotations in conjunction with a custom `BeanPostProcessor` implementation is a common means of extending the Spring IoC container. An example is Spring's `AutowiredAnnotationBeanPostProcessor` — a `BeanPostProcessor` implementation that ships with the Spring distribution and autowires annotated fields, setter methods, and arbitrary config methods.

Customizing Configuration Metadata with a `BeanFactoryPostProcessor`

The next extension point that we look at is the `org.springframework.beans.factory.config.BeanFactoryPostProcessor`. The semantics of this interface are similar to those of the `BeanPostProcessor`, with one major difference: `BeanFactoryPostProcessor` operates on the bean configuration metadata. That is, the Spring IoC container lets a `BeanFactoryPostProcessor` read the configuration metadata and potentially change it *before* the container instantiates any beans other than `BeanFactoryPostProcessor` instances.

You can configure multiple `BeanFactoryPostProcessor` instances, and you can control the order in which these `BeanFactoryPostProcessor` instances run by setting the `order` property. However, you can only set this property if the `BeanFactoryPostProcessor` implements the `Ordered` interface. If you write your own `BeanFactoryPostProcessor`, you should consider implementing the `Ordered` interface, too. See the javadoc of the `BeanFactoryPostProcessor` and `Ordered` interfaces for more details.

If you want to change the actual bean instances (that is, the objects that are created from the configuration metadata), then you instead need to use a `BeanPostProcessor` (described earlier in Customizing Beans by Using a `BeanPostProcessor`). While it is technically possible to work with bean instances within a `BeanFactoryPostProcessor` (for example, by using `BeanFactory.getBean()`), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects, such as bypassing bean post processing.

Also, `BeanFactoryPostProcessor` instances are scoped per-container. This is only relevant if you use container hierarchies. If you define a `BeanFactoryPostProcessor` in one container, it is applied only to the bean definitions in that container. Bean definitions in one container are not post-processed by `BeanFactoryPostProcessor` instances in another container, even if both containers are part of the same hierarchy.

A bean factory post-processor is automatically run when it is declared inside an `ApplicationContext`, in order to apply changes to the configuration metadata that define the container. Spring includes a number of predefined bean factory post-processors, such as `PropertyOverrideConfigurer` and `PropertySourcesPlaceholderConfigurer`. You can also use a custom `BeanFactoryPostProcessor` — for example, to register custom property editors.

An `ApplicationContext` automatically detects any beans that are deployed into it that implement the `BeanFactoryPostProcessor` interface. It uses these beans as bean factory post-processors, at the appropriate time. You can deploy these post-processor beans as you would any other bean.

As with `BeanPostProcessor`s , you typically do not want to configure `BeanFactoryPostProcessor`s for lazy initialization. If no other bean references a `Bean(Factory)PostProcessor`, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the `Bean(Factory)PostProcessor` will be instantiated eagerly even if you set the `default-lazy-init` attribute to `true` on the declaration of your `<beans />` element.

Example: The Class Name Substitution `PropertySourcesPlaceholderConfigurer`

You can use the `PropertySourcesPlaceholderConfigurer` to externalize property values from a bean definition in a separate file by using the standard Java `Properties` format. Doing so enables the person deploying an application to customize environment-specific properties, such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

Consider the following XML-based configuration metadata fragment, where a `DataSource` with placeholder values is defined:

```
<bean
class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="locations" value="classpath:com/something/jdbc.properties"/>
</bean>

<bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

The example shows properties configured from an external `Properties` file. At runtime, a `PropertySourcesPlaceholderConfigurer` is applied to the metadata that replaces some properties of the DataSource. The values to replace are specified as placeholders of the form `${property-name}`, which follows the Ant and log4j and JSP EL style.

The actual values come from another file in the standard Java `Properties` format:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

Therefore, the `${jdbc.username}` string is replaced at runtime with the value, 'sa', and the same applies for other placeholder values that match keys in the properties file. The `PropertySourcesPlaceholderConfigurer` checks for placeholders in most properties and attributes of a bean definition. Furthermore, you can customize the placeholder prefix and suffix.

With the `context` namespace introduced in Spring 2.5, you can configure property placeholders with a dedicated configuration element. You can provide one or more locations as a comma-separated list in the `location` attribute, as the following example shows:

```
<context:property-placeholder location="classpath:com/something/jdbc.properties"/>
```

The `PropertySourcesPlaceholderConfigurer` not only looks for properties in the `Properties` file you specify. By default, if it cannot find a property in the specified properties files, it checks against Spring `Environment` properties and regular Java `System` properties.

You can use the `PropertySourcesPlaceholderConfigurer` to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. The following example shows how to do so:

```
<bean
class="org.springframework.beans.factory.config.PropertySourcesPlacehol
derConfigurer">
    <property name="locations">
        <value>classpath:com/something/strategy.properties</value>
    </property>
    <property name="properties">

<value>custom.strategy.class=com.something.DefaultStrategy</value>
    </property>
</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the `preInstantiateSingletons()` phase of an `ApplicationContext` for a non-lazy-init bean.

Example: The `PropertyOverrideConfigurer`

The `PropertyOverrideConfigurer`, another bean factory post-processor, resembles the `PropertySourcesPlaceholderConfigurer`, but unlike the latter, the original definitions can have default values or no values at all for bean properties. If an overriding `Properties` file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean definition is not aware of being overridden, so it is not immediately obvious from the XML definition file that the override configurer is being used. In case of multiple `PropertyOverrideConfigurer` instances that define different values for the same bean property, the last one wins, due to the overriding mechanism.

Properties file configuration lines take the following format:

```
beanName.property=value
```

The following listing shows an example of the format:

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

This example file can be used with a container definition that contains a bean called `dataSource` that has `driver` and `url` properties.

Compound property names are also supported, as long as every component of the path except the

final property being overridden is already non-null (presumably initialized by the constructors). In the following example, the `sammy` property of the `bob` property of the `fred` property of the `tom` bean is set to the scalar value `123`:

```
tom.fred.bob.sammy=123
```

Specified override values are always literal values. They are not translated into bean references. This convention also applies when the original value in the XML bean definition specifies a bean reference.

With the `context` namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element, as the following example shows:

```
<context:property-override location="classpath:override.properties"/>
```

Customizing Instantiation Logic with a `FactoryBean`

You can implement the `org.springframework.beans.factory.FactoryBean` interface for objects that are themselves factories.

The `FactoryBean` interface is a point of pluggability into the Spring IoC container's instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own `FactoryBean`, write the complex initialization inside that class, and then plug your custom `FactoryBean` into the container.

The `FactoryBean<T>` interface provides three methods:

- `T getObject()`: Returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.
- `boolean isSingleton()`: Returns `true` if this `FactoryBean` returns singletons or `false` otherwise. The default implementation of this method returns `true`.
- `Class<?> getObjectType()`: Returns the object type returned by the `getObject()` method or `null` if the type is not known in advance.

The `FactoryBean` concept and interface are used in a number of places within the Spring Framework. More than 50 implementations of the `FactoryBean` interface ship with Spring itself.

When you need to ask a container for an actual `FactoryBean` instance itself instead of the bean it produces, prefix the bean's `id` with the ampersand symbol (`&`) when calling the `getBean()` method of the `ApplicationContext`. So, for a given `FactoryBean` with an `id` of `myBean`, invoking `getBean("myBean")` on the container returns the product of the `FactoryBean`, whereas invoking `getBean("&myBean")` returns the `FactoryBean` instance itself.

2.1.9. Annotation-based Container Configuration

Are annotations better than XML for configuring Spring?

The introduction of annotation-based configuration raised the question of whether this approach is "better" than XML. The short answer is "it depends." The long answer is that each approach has its pros and cons, and, usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It is worth pointing out that through its JavaConfig option, Spring lets annotations be used in a non-invasive way, without touching the target components source code and that, in terms of tooling, all configuration styles are supported by the Spring Tools for Eclipse.

An alternative to XML setup is provided by annotation-based configuration, which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. As mentioned in Example: The `AutowiredAnnotationBeanPostProcessor`, using a `BeanPostProcessor` in conjunction with annotations is a common means of extending the Spring IoC container. For example, Spring 2.5 introduced an annotation-based approach to drive Spring's dependency injection. Essentially, the `@Autowired` annotation provides the same capabilities as described in Autowiring Collaborators but with more fine-grained control and wider applicability. Spring 2.5 also added support for JSR-250 annotations, such as `@PostConstruct` and `@PreDestroy`. Spring 3.0 added support for JSR-330 (Dependency Injection for Java) annotations contained in the `jakarta.inject` package such as `@Inject` and `@Named`. Details about those annotations can be found in the relevant section.

Annotation injection is performed before XML injection. Thus, the XML configuration overrides the annotations for properties wired through both approaches.

As always, you can register the post-processors as individual bean definitions, but they can also be implicitly registered by including the following tag in an XML-based Spring configuration (notice the inclusion of the `context` namespace):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

The `<context:annotation-config/>` element implicitly registers the following post-processors:

- `ConfigurationClassPostProcessor`
- `AutowiredAnnotationBeanPostProcessor`
- `CommonAnnotationBeanPostProcessor`
- `PersistenceAnnotationBeanPostProcessor`
- `EventListenerMethodProcessor`

`<context:annotation-config/>` only looks for annotations on beans in the same application context in which it is defined. This means that, if you put `<context:annotation-config/>` in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for `@Autowired` beans in your controllers, and not your services. See The DispatcherServlet for more information.

Using `@Autowired`

JSR 330's `@Inject` annotation can be used in place of Spring's `@Autowired` annotation in the examples included in this section. See here for more details.

You can apply the `@Autowired` annotation to constructors, as the following example shows:

Java

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender @Autowired constructor(
    private val customerPreferenceDao: CustomerPreferenceDao)
```

As of Spring Framework 4.3, an `@Autowired` annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with. However, if several constructors are available and there is no primary/default constructor, at least one of the constructors must be annotated with `@Autowired` in order to instruct the container which one to use. See the discussion on constructor resolution for details.

You can also apply the `@Autowired` annotation to *traditional* setter methods, as the following example shows:

Java

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Kotlin

```
class SimpleMovieLister {

    @set:Autowired
    lateinit var movieFinder: MovieFinder

    // ...

}
```

You can also apply the annotation to methods with arbitrary names and multiple arguments, as the following example shows:

Java

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender {

    private lateinit var movieCatalog: MovieCatalog

    private lateinit var customerPreferenceDao: CustomerPreferenceDao

    @Autowired
    fun prepare(movieCatalog: MovieCatalog,
                customerPreferenceDao: CustomerPreferenceDao) {
        this.movieCatalog = movieCatalog
        this.customerPreferenceDao = customerPreferenceDao
    }

    // ...
}
```

You can apply `@Autowired` to fields as well and even mix it with constructors, as the following example shows:

Java

```
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender @Autowired constructor(
    private val customerPreferenceDao: CustomerPreferenceDao) {

    @Autowired
    private lateinit var movieCatalog: MovieCatalog

    // ...
}
```

Make sure that your target components (for example, `MovieCatalog` or `CustomerPreferenceDao`) are consistently declared by the type that you use for your `@Autowired`-annotated injection points. Otherwise, injection may fail due to a "no type match found" error at runtime.

For XML-defined beans or component classes found via classpath scanning, the container usually knows the concrete type up front. However, for `@Bean` factory methods, you need to make sure that the declared return type is sufficiently expressive. For components that implement several interfaces or for components potentially referred to by their implementation type, consider declaring the most specific return type on your factory method (at least as specific as required by the injection points referring to your bean).

You can also instruct Spring to provide all beans of a particular type from the `ApplicationContext` by adding the `@Autowired` annotation to a field or method that expects an array of that type, as the following example shows:

Java

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    private lateinit var movieCatalogs: Array<MovieCatalog>

    // ...
}
```

The same applies for typed collections, as the following example shows:

Java

```
public class MovieRecommender {

    private Set<MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    lateinit var movieCatalogs: Set<MovieCatalog>

    // ...
}
```

Your target beans can implement the `org.springframework.core.Ordered` interface or use the `@Order` or standard `@Priority` annotation if you want items in the array or list to be sorted in a specific order. Otherwise, their order follows the registration order of the corresponding target bean definitions in the container.

You can declare the `@Order` annotation at the target class level and on `@Bean` methods, potentially for individual bean definitions (in case of multiple definitions that use the same bean class). `@Order` values may influence priorities at injection points, but be aware that they do not influence singleton startup order, which is an orthogonal concern determined by dependency relationships and `@DependsOn` declarations.

Note that the standard `jakarta.annotation.Priority` annotation is not available at the `@Bean` level, since it cannot be declared on methods. Its semantics can be modeled through `@Order` values in combination with `@Primary` on a single bean for each type.

Even typed `Map` instances can be autowired as long as the expected key type is `String`. The map values contain all beans of the expected type, and the keys contain the corresponding bean names, as the following example shows:

Java

```
public class MovieRecommender {

    private Map<String, MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    lateinit var movieCatalogs: Map<String, MovieCatalog>

    // ...
}
```

By default, autowiring fails when no matching candidate beans are available for a given injection point. In the case of a declared array, collection, or map, at least one matching element is expected.

The default behavior is to treat annotated methods and fields as indicating required dependencies.

You can change this behavior as demonstrated in the following example, enabling the framework to skip a non-satisfiable injection point through marking it as non-required (i.e., by setting the `required` attribute in `@Autowired` to `false`):

Java

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired(required = false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Kotlin

```
class SimpleMovieLister {

    @Autowired(required = false)
    var movieFinder: MovieFinder? = null

    // ...
}
```

A non-required method will not be called at all if its dependency (or one of its dependencies, in case of multiple arguments) is not available. A non-required field will not get populated at all in such cases, leaving its default value in place.

In other words, setting the `required` attribute to `false` indicates that the corresponding property is *optional* for autowiring purposes, and the property will be ignored if it cannot be autowired. This allows properties to be assigned default values that can be optionally overridden via dependency injection.

Injected constructor and factory method arguments are a special case since the `required` attribute in `@Autowired` has a somewhat different meaning due to Spring's constructor resolution algorithm that may potentially deal with multiple constructors. Constructor and factory method arguments are effectively required by default but with a few special rules in a single-constructor scenario, such as multi-element injection points (arrays, collections, maps) resolving to empty instances if no matching beans are available. This allows for a common implementation pattern where all dependencies can be declared in a unique multi-argument constructor — for example, declared as a single public constructor without an `@Autowired` annotation.

Only one constructor of any given bean class may declare `@Autowired` with the `required` attribute set to `true`, indicating *the* constructor to autowire when used as a Spring bean. As a consequence, if the `required` attribute is left at its default value `true`, only a single constructor may be annotated with `@Autowired`. If multiple constructors declare the annotation, they will all have to declare `required=false` in order to be considered as candidates for autowiring (analogous to `autowire=constructor` in XML). The constructor with the greatest number of dependencies that can be satisfied by matching beans in the Spring container will be chosen. If none of the candidates can be satisfied, then a primary/default constructor (if present) will be used. Similarly, if a class declares multiple constructors but none of them is annotated with `@Autowired`, then a primary/default constructor (if present) will be used. If a class only declares a single constructor to begin with, it will always be used, even if not annotated. Note that an annotated constructor does not have to be public.

Alternatively, you can express the non-required nature of a particular dependency through Java 8's `java.util.Optional`, as the following example shows:

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        ...
    }
}
```

As of Spring Framework 5.0, you can also use a `@Nullable` annotation (of any kind in any package — for example, `javax.annotation.Nullable` from JSR-305) or just leverage Kotlin builtin null-safety support:

Java

```
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

Kotlin

```
class SimpleMovieLister {

    @Autowired
    var movieFinder: MovieFinder? = null

    // ...
}
```

You can also use `@Autowired` for interfaces that are well-known resolvable dependencies: `BeanFactory`, `ApplicationContext`, `Environment`, `ResourceLoader`, `ApplicationEventPublisher`, and `MessageSource`. These interfaces and their extended interfaces, such as `ConfigurableApplicationContext` or `ResourcePatternResolver`, are automatically resolved, with no special setup necessary. The following example autowires an `ApplicationContext` object:

Java

```
public class MovieRecommender {

    @Autowired
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    lateinit var context: ApplicationContext

    // ...
}
```

The `@Autowired`, `@Inject`, `@Value`, and `@Resource` annotations are handled by Spring `BeanPostProcessor` implementations. This means that you cannot apply these annotations within your own `BeanPostProcessor` or `BeanFactoryPostProcessor` types (if any). These types must be 'wired up' explicitly by using XML or a Spring `@Bean` method.

Fine-tuning Annotation-based Autowiring with `@Primary`

Because autowiring by type may lead to multiple candidates, it is often necessary to have more

control over the selection process. One way to accomplish this is with Spring's `@Primary` annotation. `@Primary` indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one primary bean exists among the candidates, it becomes the autowired value.

Consider the following configuration that defines `firstMovieCatalog` as the primary `MovieCatalog`:

Java

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

Kotlin

```
@Configuration
class MovieConfiguration {

    @Bean
    @Primary
    fun firstMovieCatalog(): MovieCatalog { ... }

    @Bean
    fun secondMovieCatalog(): MovieCatalog { ... }

    // ...
}
```

With the preceding configuration, the following `MovieRecommender` is autowired with the `firstMovieCatalog`:

Java

```
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    private lateinit var movieCatalog: MovieCatalog

    // ...
}
```

The corresponding bean definitions follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog" primary="true">
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

Fine-tuning Annotation-based Autowiring with Qualifiers

`@Primary` is an effective way to use autowiring by type with several instances when one primary candidate can be determined. When you need more control over the selection process, you can use Spring's `@Qualifier` annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument. In the simplest case, this can be a plain descriptive value, as shown in the following example:

Java

```
public class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private lateinit var movieCatalog: MovieCatalog

    // ...
}
```

You can also specify the `@Qualifier` annotation on individual constructor arguments or method parameters, as shown in the following example:

Java

```
public class MovieRecommender {

    private final MovieCatalog movieCatalog;

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender {

    private lateinit var movieCatalog: MovieCatalog

    private lateinit var customerPreferenceDao: CustomerPreferenceDao

    @Autowired
    fun prepare(@Qualifier("main") movieCatalog: MovieCatalog,
                customerPreferenceDao: CustomerPreferenceDao) {
        this.movieCatalog = movieCatalog
        this.customerPreferenceDao = customerPreferenceDao
    }

    // ...
}
```

The following example shows corresponding bean definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/> ①

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/> ②

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

① The bean with the `main` qualifier value is wired with the constructor argument that is qualified with the same value.

② The bean with the `action` qualifier value is wired with the constructor argument that is qualified with the same value.

For a fallback match, the bean name is considered a default qualifier value. Thus, you can define the bean with an `id` of `main` instead of the nested qualifier element, leading to the same matching result. However, although you can use this convention to refer to specific beans by name, `@Autowired` is fundamentally about type-driven injection with optional semantic qualifiers. This means that qualifier values, even with the bean name fallback, always have narrowing semantics within the set of type matches. They do not semantically express a reference to a unique bean `id`. Good qualifier values are `main` or `EMEA` or `persistent`, expressing characteristics of a specific component that are independent from the bean `id`, which may be auto-generated in case of an anonymous bean definition such as the one in the preceding example.

Qualifiers also apply to typed collections, as discussed earlier — for example, to `Set<MovieCatalog>`. In this case, all matching beans, according to the declared qualifiers, are injected as a collection. This implies that qualifiers do not have to be unique. Rather, they constitute filtering criteria. For example, you can define multiple `MovieCatalog` beans with the same qualifier value “action”, all of which are injected into a `Set<MovieCatalog>` annotated with `@Qualifier("action")`.

Letting qualifier values select against target bean names, within the type-matching candidates, does not require a `@Qualifier` annotation at the injection point. If there is no other resolution indicator (such as a qualifier or a primary marker), for a non-unique dependency situation, Spring matches the injection point name (that is, the field name or parameter name) against the target bean names and chooses the same-named candidate, if any.

That said, if you intend to express annotation-driven injection by name, do not primarily use `@Autowired`, even if it is capable of selecting by bean name among type-matching candidates. Instead, use the JSR-250 `@Resource` annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process. `@Autowired` has rather different semantics: After selecting candidate beans by type, the specified `String` qualifier value is considered within those type-selected candidates only (for example, matching an `account` qualifier against beans marked with the same qualifier label).

For beans that are themselves defined as a collection, `Map`, or array type, `@Resource` is a fine solution, referring to the specific collection or array bean by unique name. That said, as of 4.3, you can match collection, `Map`, and array types through Spring’s `@Autowired` type matching algorithm as well, as long as the element type information is preserved in `@Bean` return type signatures or collection inheritance hierarchies. In this case, you can use qualifier values to select among same-typed collections, as outlined in the previous paragraph.

As of 4.3, `@Autowired` also considers self references for injection (that is, references back to the bean that is currently injected). Note that self injection is a fallback. Regular dependencies on other components always have precedence. In that sense, self references do not participate in regular candidate selection and are therefore in particular never primary. On the contrary, they always end up as lowest precedence. In practice, you should use self references as a last resort only (for example, for calling other methods on the same instance through the bean’s transactional proxy). Consider factoring out the affected methods to a separate delegate bean in such a scenario.

Alternatively, you can use `@Resource`, which may obtain a proxy back to the current bean by its unique name.

Trying to inject the results from `@Bean` methods on the same configuration class is effectively a self-reference scenario as well. Either lazily resolve such references in the method signature where it is actually needed (as opposed to an autowired field in the configuration class) or declare the affected `@Bean` methods as `static`, decoupling them from the containing configuration class instance and its lifecycle. Otherwise, such beans are only considered in the fallback phase, with matching beans on other configuration classes selected as primary candidates instead (if available).

`@Autowired` applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level. In contrast, `@Resource` is supported only for fields and bean property setter methods with a single argument. As a consequence, you should stick with qualifiers if your injection target is a constructor or a multi-argument method.

You can create your own custom qualifier annotations. To do so, define an annotation and provide the `@Qualifier` annotation within your definition, as the following example shows:

Java

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {

    String value();
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class Genre(val value: String)
```

Then you can provide the custom qualifier on autowired fields and parameters, as the following example shows:

Java

```
public class MovieRecommender {

    @Autowired
    @Genre("Action")
    private MovieCatalog actionCatalog;

    private MovieCatalog comedyCatalog;

    @Autowired
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {
        this.comedyCatalog = comedyCatalog;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    @Genre("Action")
    private lateinit var actionCatalog: MovieCatalog

    private lateinit var comedyCatalog: MovieCatalog

    @Autowired
    fun setComedyCatalog(@Genre("Comedy") comedyCatalog: MovieCatalog) {
        this.comedyCatalog = comedyCatalog
    }

    // ...
}
```

Next, you can provide the information for the candidate bean definitions. You can add `<qualifier/>` tags as sub-elements of the `<bean/>` tag and then specify the `type` and `value` to match your custom qualifier annotations. The type is matched against the fully-qualified class name of the annotation. Alternately, as a convenience if no risk of conflicting names exists, you can use the short class name. The following example demonstrates both approaches:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="Genre" value="Action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="example.Genre" value="Comedy"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

In Classpath Scanning and Managed Components, you can see an annotation-based alternative to providing the qualifier metadata in XML. Specifically, see Providing Qualifier Metadata with Annotations.

In some cases, using an annotation without a value may suffice. This can be useful when the annotation serves a more generic purpose and can be applied across several different types of dependencies. For example, you may provide an offline catalog that can be searched when no Internet connection is available. First, define the simple annotation, as the following example shows:

Java

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class Offline
```

Then add the annotation to the field or property to be autowired, as shown in the following example:

Java

```
public class MovieRecommender {

    @Autowired
    @Offline ①
    private MovieCatalog offlineCatalog;

    // ...
}
```

① This line adds the `@Offline` annotation.

Kotlin

```
class MovieRecommender {

    @Autowired
    @Offline ①
    private lateinit var offlineCatalog: MovieCatalog

    // ...
}
```

① This line adds the `@Offline` annotation.

Now the bean definition only needs a qualifier `type`, as shown in the following example:

```
<bean class="example.SimpleMovieCatalog">
    <qualifier type="Offline"/> ①
    <!-- inject any dependencies required by this bean -->
</bean>
```

① This element specifies the qualifier.

You can also define custom qualifier annotations that accept named attributes in addition to or instead of the simple `value` attribute. If multiple attribute values are then specified on a field or parameter to be autowired, a bean definition must match all such attribute values to be considered an autowire candidate. As an example, consider the following annotation definition:

Java

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MovieQualifier {

    String genre();

    Format format();
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class MovieQualifier(val genre: String, val format: Format)
```

In this case `Format` is an enum, defined as follows:

Java

```
public enum Format {
    VHS, DVD, BLURAY
}
```

Kotlin

```
enum class Format {
    VHS, DVD, BLURAY
}
```

The fields to be autowired are annotated with the custom qualifier and include values for both attributes: `genre` and `format`, as the following example shows:

Java

```
public class MovieRecommender {

    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Action")
    private MovieCatalog actionVhsCatalog;

    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Comedy")
    private MovieCatalog comedyVhsCatalog;

    @Autowired
    @MovieQualifier(format=Format.DVD, genre="Action")
    private MovieCatalog actionDvdCatalog;

    @Autowired
    @MovieQualifier(format=Format.BLURAY, genre="Comedy")
    private MovieCatalog comedyBluRayCatalog;

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    @MovieQualifier(format = Format.VHS, genre = "Action")
    private lateinit var actionVhsCatalog: MovieCatalog

    @Autowired
    @MovieQualifier(format = Format.VHS, genre = "Comedy")
    private lateinit var comedyVhsCatalog: MovieCatalog

    @Autowired
    @MovieQualifier(format = Format.DVD, genre = "Action")
    private lateinit var actionDvdCatalog: MovieCatalog

    @Autowired
    @MovieQualifier(format = Format.BLURAY, genre = "Comedy")
    private lateinit var comedyBluRayCatalog: MovieCatalog

    // ...
}
```

Finally, the bean definitions should contain matching qualifier values. This example also demonstrates that you can use bean meta attributes instead of the `<qualifier/>` elements. If available, the `<qualifier/>` element and its attributes take precedence, but the autowiring

mechanism falls back on the values provided within the `<meta/>` tags if no such qualifier is present, as in the last two bean definitions in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Action"/>
        </qualifier>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Comedy"/>
        </qualifier>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="DVD"/>
        <meta key="genre" value="Action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="BLURAY"/>
        <meta key="genre" value="Comedy"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

</beans>
```

Using Generics as Autowiring Qualifiers

In addition to the `@Qualifier` annotation, you can use Java generic types as an implicit form of qualification. For example, suppose you have the following configuration:

Java

```
@Configuration
public class MyConfiguration {

    @Bean
    public StringStore stringStore() {
        return new StringStore();
    }

    @Bean
    public IntegerStore integerStore() {
        return new IntegerStore();
    }
}
```

Kotlin

```
@Configuration
class MyConfiguration {

    @Bean
    fun stringStore() = StringStore()

    @Bean
    fun integerStore() = IntegerStore()
}
```

Assuming that the preceding beans implement a generic interface, (that is, `Store<String>` and `Store<Integer>`), you can `@Autowire` the `Store` interface and the generic is used as a qualifier, as the following example shows:

Java

```
@Autowired
private Store<String> s1; // <String> qualifier, injects the stringStore bean

@Autowired
private Store<Integer> s2; // <Integer> qualifier, injects the integerStore bean
```

Kotlin

```
@Autowired
private lateinit var s1: Store<String> // <String> qualifier, injects the stringStore
bean

@Autowired
private lateinit var s2: Store<Integer> // <Integer> qualifier, injects the
integerStore bean
```

Generic qualifiers also apply when autowiring lists, `Map` instances and arrays. The following example autowires a generic `List`:

Java

```
// Inject all Store beans as long as they have an <Integer> generic
// Store<String> beans will not appear in this list
@Autowired
private List<Store<Integer>> s;
```

Kotlin

```
// Inject all Store beans as long as they have an <Integer> generic
// Store<String> beans will not appear in this list
@Autowired
private lateinit var s: List<Store<Integer>>
```

Using `CustomAutowireConfigurer`

`CustomAutowireConfigurer` is a `BeanFactoryPostProcessor` that lets you register your own custom qualifier annotation types, even if they are not annotated with Spring's `@Qualifier` annotation. The following example shows how to use `CustomAutowireConfigurer`:

```
<bean id="customAutowireConfigurer"
        class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>example.CustomQualifier</value>
        </set>
    </property>
</bean>
```

The `AutowireCandidateResolver` determines autowire candidates by:

- The `autowire-candidate` value of each bean definition
- Any `default-autowire-candidates` patterns available on the `<beans/>` element
- The presence of `@Qualifier` annotations and any custom annotations registered with the

`CustomAutowireConfigurer`

When multiple beans qualify as autowire candidates, the determination of a "primary" is as follows: If exactly one bean definition among the candidates has a `primary` attribute set to `true`, it is selected.

Injection with `@Resource`

Spring also supports injection by using the JSR-250 `@Resource` annotation (`jakarta.annotation.Resource`) on fields or bean property setter methods. This is a common pattern in Jakarta EE: for example, in JSF-managed beans and JAX-WS endpoints. Spring supports this pattern for Spring-managed objects as well.

`@Resource` takes a name attribute. By default, Spring interprets that value as the bean name to be injected. In other words, it follows by-name semantics, as demonstrated in the following example:

Java

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource(name="myMovieFinder") ①
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

① This line injects a `@Resource`.

Kotlin

```
class SimpleMovieLister {

    @Resource(name="myMovieFinder") ①
    private lateinit var movieFinder:MovieFinder
}
```

① This line injects a `@Resource`.

If no name is explicitly specified, the default name is derived from the field name or setter method. In case of a field, it takes the field name. In case of a setter method, it takes the bean property name. The following example is going to have the bean named `movieFinder` injected into its setter method:

Java

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

Kotlin

```
class SimpleMovieLister {

    @set:Resource
    private lateinit var movieFinder: MovieFinder

}
```

The name provided with the annotation is resolved as a bean name by the `ApplicationContext` of which the `CommonAnnotationBeanPostProcessor` is aware. The names can be resolved through JNDI if you configure Spring's `SimpleJndiBeanFactory` explicitly. However, we recommend that you rely on the default behavior and use Spring's JNDI lookup capabilities to preserve the level of indirection.

In the exclusive case of `@Resource` usage with no explicit name specified, and similar to `@Autowired`, `@Resource` finds a primary type match instead of a specific named bean and resolves well known resolvable dependencies: the `BeanFactory`, `ApplicationContext`, `ResourceLoader`, `ApplicationEventPublisher`, and `MessageSource` interfaces.

Thus, in the following example, the `customerPreferenceDao` field first looks for a bean named "customerPreferenceDao" and then falls back to a primary type match for the type `CustomerPreferenceDao`:

Java

```
public class MovieRecommender {

    @Resource
    private CustomerPreferenceDao customerPreferenceDao;

    @Resource
    private ApplicationContext context; ①

    public MovieRecommender() {
    }

    // ...
}
```

① The `context` field is injected based on the known resolvable dependency type: `ApplicationContext`.

Kotlin

```
class MovieRecommender {

    @Resource
    private lateinit var customerPreferenceDao: CustomerPreferenceDao


    @Resource
    private lateinit var context: ApplicationContext ①

    // ...
}
```

① The `context` field is injected based on the known resolvable dependency type: `ApplicationContext`.

Using `@Value`

`@Value` is typically used to inject externalized properties:

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name}") String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(@Value("\${catalog.name}") private val catalog: String)
```

With the following configuration:

Java

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { }
```

Kotlin

```
@Configuration
@PropertySource("classpath:application.properties")
class AppConfig
```

And the following `application.properties` file:

```
catalog.name=MovieCatalog
```

In that case, the `catalog` parameter and field will be equal to the `MovieCatalog` value.

A default lenient embedded value resolver is provided by Spring. It will try to resolve the property value and if it cannot be resolved, the property name (for example `${catalog.name}`) will be injected as the value. If you want to maintain strict control over nonexistent values, you should declare a `PropertySourcesPlaceholderConfigurer` bean, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer()
{
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun propertyPlaceholderConfigurer() = PropertySourcesPlaceholderConfigurer()
}
```

When configuring a `PropertySourcesPlaceholderConfigurer` using JavaConfig, the `@Bean` method must be `static`.

Using the above configuration ensures Spring initialization failure if any `${}` placeholder could not be resolved. It is also possible to use methods like `setPlaceholderPrefix`, `setPlaceholderSuffix`, or `setValueSeparator` to customize placeholders.

Spring Boot configures by default a `PropertySourcesPlaceholderConfigurer` bean that will get properties from `application.properties` and `application.yml` files.

Built-in converter support provided by Spring allows simple type conversion (to `Integer` or `int` for example) to be automatically handled. Multiple comma-separated values can be automatically converted to `String` array without extra effort.

It is possible to provide a default value as following:

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name:defaultCatalog}") String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(@Value("\${catalog.name:defaultCatalog}") private val catalog:
String)
```

A Spring `BeanPostProcessor` uses a `ConversionService` behind the scenes to handle the process for converting the `String` value in `@Value` to the target type. If you want to provide conversion support for your own custom type, you can provide your own `ConversionService` bean instance as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public ConversionService conversionService() {
        DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService();
        conversionService.addConverter(new MyCustomConverter());
        return conversionService;
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun conversionService(): ConversionService {
        return DefaultFormattingConversionService().apply {
            addConverter(MyCustomConverter())
        }
    }
}
```

When `@Value` contains a `SpEL` expression the value will be dynamically computed at runtime as the following example shows:

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("#{systemProperties['user.catalog'] + 'Catalog' }")
String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(
    @Value("#{systemProperties['user.catalog'] + 'Catalog' }") private val catalog:
String)
```

SpEL also enables the use of more complex data structures:

Java

```
@Component
public class MovieRecommender {

    private final Map<String, Integer> countOfMoviesPerCatalog;

    public MovieRecommender(
            @Value("#{{'Thriller': 100, 'Comedy': 300}}") Map<String, Integer>
countOfMoviesPerCatalog) {
        this.countOfMoviesPerCatalog = countOfMoviesPerCatalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(
    @Value("#{{'Thriller': 100, 'Comedy': 300}}") private val countOfMoviesPerCatalog:
Map<String, Int>)
```

Using `@PostConstruct` and `@PreDestroy`

The `CommonAnnotationBeanPostProcessor` not only recognizes the `@Resource` annotation but also the JSR-250 lifecycle annotations: `jakarta.annotation.PostConstruct` and `jakarta.annotation.PreDestroy`. Introduced in Spring 2.5, the support for these annotations offers an alternative to the lifecycle

callback mechanism described in initialization callbacks and destruction callbacks. Provided that the `CommonAnnotationBeanPostProcessor` is registered within the Spring `ApplicationContext`, a method carrying one of these annotations is invoked at the same point in the lifecycle as the corresponding Spring lifecycle interface method or explicitly declared callback method. In the following example, the cache is pre-populated upon initialization and cleared upon destruction:

Java

```
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }
}
```

Kotlin

```
class CachingMovieLister {

    @PostConstruct
    fun populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    fun clearMovieCache() {
        // clears the movie cache upon destruction...
    }
}
```

For details about the effects of combining various lifecycle mechanisms, see Combining Lifecycle Mechanisms.

Like `@Resource`, the `@PostConstruct` and `@PreDestroy` annotation types were a part of the standard Java libraries from JDK 6 to 8. However, the entire `javax.annotation` package got separated from the core Java modules in JDK 9 and eventually removed in JDK 11. As of Jakarta EE 9, the package lives in `jakarta.annotation` now. If needed, the `jakarta.annotation-api` artifact needs to be obtained via Maven Central now, simply to be added to the application's classpath like any other library.

2.1.10. Classpath Scanning and Managed Components

Most examples in this chapter use XML to specify the configuration metadata that produces each `BeanDefinition` within the Spring container. The previous section (Annotation-based Container Configuration) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file, while the annotations drive only the dependency injection. This section describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, `@Component`), AspectJ type expressions, or your own custom filter criteria to select which classes have bean definitions registered with the container.

Starting with Spring 3.0, many features provided by the Spring JavaConfig project are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files. Take a look at the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations for examples of how to use these new features.

`@Component` and Further Stereotype Annotations

The `@Repository` annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO). Among the uses of this marker is the automatic translation of exceptions, as described in Exception Translation.

Spring provides further stereotype annotations: `@Component`, `@Service`, and `@Controller`. `@Component` is a generic stereotype for any Spring-managed component. `@Repository`, `@Service`, and `@Controller` are specializations of `@Component` for more specific use cases (in the persistence, service, and presentation layers, respectively). Therefore, you can annotate your component classes with `@Component`, but, by annotating them with `@Repository`, `@Service`, or `@Controller` instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. `@Repository`, `@Service`, and `@Controller` can also carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using `@Component` or `@Service` for your service layer, `@Service` is clearly the better choice. Similarly, as stated earlier, `@Repository` is already supported as a marker for automatic exception translation in your persistence layer.

Using Meta-annotations and Composed Annotations

Many of the annotations provided by Spring can be used as meta-annotations in your own code. A meta-annotation is an annotation that can be applied to another annotation. For example, the `@Service` annotation mentioned earlier is meta-annotated with `@Component`, as the following example shows:

Java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component ①
public @interface Service {

    // ...
}
```

① The `@Component` causes `@Service` to be treated in the same way as `@Component`.

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Component ①
annotation class Service {

    // ...
}
```

① The `@Component` causes `@Service` to be treated in the same way as `@Component`.

You can also combine meta-annotations to create “composed annotations”. For example, the `@RestController` annotation from Spring MVC is composed of `@Controller` and `@ResponseBody`.

In addition, composed annotations can optionally redeclare attributes from meta-annotations to allow customization. This can be particularly useful when you want to only expose a subset of the meta-annotation’s attributes. For example, Spring’s `@SessionScope` annotation hardcodes the scope name to `session` but still allows customization of the `proxyMode`. The following listing shows the definition of the `SessionScope` annotation:

Java

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope(WebApplicationContext.SCOPE_SESSION)
public @interface SessionScope {

    /**
     * Alias for {@link Scope#proxyMode}.
     * <p>Defaults to {@link ScopedProxyMode#TARGET_CLASS}.
     */
    @AliasFor(annotation = Scope.class)
    ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;

}
```

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Scope(WebApplicationContext.SCOPE_SESSION)
annotation class SessionScope(
        @get:AliasFor(annotation = Scope::class)
        val proxyMode: ScopedProxyMode = ScopedProxyMode.TARGET_CLASS
)
```

You can then use `@SessionScope` without declaring the `proxyMode` as follows:

Java

```
@Service
@SessionScope
public class SessionScopedService {
    // ...
}
```

Kotlin

```
@Service
@SessionScope
class SessionScopedService {
    // ...
}
```

You can also override the value for the `proxyMode`, as the following example shows:

Java

```
@Service
@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)
public class SessionScopedUserService implements UserService {
    // ...
}
```

Kotlin

```
@Service
@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)
class SessionScopedUserService : UserService {
    // ...
}
```

For further details, see the Spring Annotation Programming Model wiki page.

Automatically Detecting Classes and Registering Bean Definitions

Spring can automatically detect stereotyped classes and register corresponding `BeanDefinition` instances with the `ApplicationContext`. For example, the following two classes are eligible for such autodetection:

Java

```
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

Kotlin

```
@Service
class SimpleMovieLister(private val movieFinder: MovieFinder)
```

Java

```
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

Kotlin

```
@Repository
class JpaMovieFinder : MovieFinder {
    // implementation elided for clarity
}
```

To autodetect these classes and register the corresponding beans, you need to add `@ComponentScan` to your `@Configuration` class, where the `basePackages` attribute is a common parent package for the two classes. (Alternatively, you can specify a comma- or semicolon- or space-separated list that includes the parent package of each class.)

Java

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["org.example"])
class AppConfig  {
    // ...
}
```

For brevity, the preceding example could have used the `value` attribute of the annotation (that is, `@ComponentScan("org.example")`).

The following alternative uses XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

The use of `<context:component-scan>` implicitly enables the functionality of `<context:annotation-config>`. There is usually no need to include the `<context:annotation-config>` element when using `<context:component-scan>`.

The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do not activate the files-only switch of the JAR task. Also, classpath directories may not be exposed based on security policies in some environments — for example, standalone apps on JDK 1.7.0_45 and higher (which requires 'Trusted-Library' setup in your manifests — see https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources).

On JDK 9's module path (Jigsaw), Spring's classpath scanning generally works as expected. However, make sure that your component classes are exported in your `module-info` descriptors. If you expect Spring to invoke non-public members of your classes, make sure that they are 'opened' (that is, that they use an `opens` declaration instead of an `exports` declaration in your `module-info` descriptor).

Furthermore, the `AutowiredAnnotationBeanPostProcessor` and `CommonAnnotationBeanPostProcessor` are both implicitly included when you use the component-scan element. That means that the two components are autodetected and wired together — all without any bean configuration metadata provided in XML.

You can disable the registration of `AutowiredAnnotationBeanPostProcessor` and `CommonAnnotationBeanPostProcessor` by including the `annotation-config` attribute with a value of `false`.

Using Filters to Customize Scanning

By default, classes annotated with `@Component`, `@Repository`, `@Service`, `@Controller`, `@Configuration`, or a custom annotation that itself is annotated with `@Component` are the only detected candidate components. However, you can modify and extend this behavior by applying custom filters. Add them as `includeFilters` or `excludeFilters` attributes of the `@ComponentScan` annotation (or as `<context:include-filter />` or `<context:exclude-filter />` child elements of the `<context:component-scan>` element in XML configuration). Each filter element requires the `type` and `expression` attributes. The following table describes the filtering options:

Table 5. Filter Types

Filter Type	Example Expression	Description
annotation (default)	`org.example.SomeAnnotation`	An annotation to be *present* or *meta-present* at the type level in target components.
assignable	`org.example.SomeClass`	A class (or interface) that the target components are assignable to (extend or implement).

Filter Type	Example Expression	Description
aspectj	`org.example..*Service+`	An AspectJ type expression to be matched by the target components.
regex	`org\.example\.Default.*`	A regex expression to be matched by the target components' class names.
custom	`org.example.MyTypeFilter`	A custom implementation of the `org.springframework.core.type.TypeFilter` interface.

The following example shows the configuration ignoring all `@Repository` annotations and using "stub" repositories instead:

Java

```
@Configuration
@ComponentScan(basePackages = "org.example",
        includeFilters = @Filter(type = FilterType.REGEX, pattern =
".*Stub.*Repository"),
        excludeFilters = @Filter(Repository.class))
public class AppConfig {
    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["org.example"],
        includeFilters = [Filter(type = FilterType.REGEX, pattern =
[".*Stub.*Repository"])],
        excludeFilters = [Filter(Repository::class)])
class AppConfig {
    // ...
}
```

The following listing shows the equivalent XML:

```
<beans>
    <context:component-scan base-package="org.example">
        <context:include-filter type="regex"
                expression=".*Stub.*Repository"/>
        <context:exclude-filter type="annotation"
                expression="org.springframework.stereotype.Repository"/>
    </context:component-scan>
</beans>
```

You can also disable the default filters by setting `useDefaultFilters=false` on the annotation or by providing `use-default-filters="false"` as an attribute of the `<component-scan/>` element. This effectively disables automatic detection of classes annotated or meta-annotated with `@Component`, `@Repository`, `@Service`, `@Controller`, `@RestController`, or `@Configuration`.

Defining Bean Metadata within Components

Spring components can also contribute bean definition metadata to the container. You can do this with the same `@Bean` annotation used to define bean metadata within `@Configuration` annotated classes. The following example shows how to do so:

Java

```
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    fun publicInstance() = TestBean("publicInstance")

    fun doWork() {
        // Component method implementation omitted
    }
}
```

The preceding class is a Spring component that has application-specific code in its `doWork()` method. However, it also contributes a bean definition that has a factory method referring to the method `publicInstance()`. The `@Bean` annotation identifies the factory method and other bean definition properties, such as a qualifier value through the `@Qualifier` annotation. Other method-level annotations that can be specified are `@Scope`, `@Lazy`, and custom qualifier annotations.

In addition to its role for component initialization, you can also place the `@Lazy` annotation on injection points marked with `@Autowired` or `@Inject`. In this context, it leads to the injection of a lazy-resolution proxy. However, such a proxy approach is rather limited. For sophisticated lazy interactions, in particular in combination with optional dependencies, we recommend `ObjectProvider<MyTargetBean>` instead.

Autowired fields and methods are supported, as previously discussed, with additional support for autowiring of `@Bean` methods. The following example shows how to do so:

Java

```
@Component
public class FactoryMethodComponent {

    private static int i;

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    // use of a custom qualifier and autowiring of method parameters
    @Bean
    protected TestBean protectedInstance(
            @Qualifier("public") TestBean spouse,
            @Value("#{privateInstance.age}") String country) {
        TestBean tb = new TestBean("protectedInstance", 1);
        tb.setSpouse(spouse);
        tb.setCountry(country);
        return tb;
    }

    @Bean
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }

    @Bean
    @RequestScope
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    companion object {
        private var i: Int = 0
    }

    @Bean
    @Qualifier("public")
    fun publicInstance() = TestBean("publicInstance")

    // use of a custom qualifier and autowiring of method parameters
    @Bean
    protected fun protectedInstance(
            @Qualifier("public") spouse: TestBean,
            @Value("#{privateInstance.age}") country: String) =
TestBean("protectedInstance", 1).apply {
        this.spouse = spouse
        this.country = country
    }

    @Bean
    private fun privateInstance() = TestBean("privateInstance", i++)

    @Bean
    @RequestScope
    fun requestScopedInstance() = TestBean("requestScopedInstance", 3)
}
```

The example autowires the `String` method parameter `country` to the value of the `age` property on another bean named `privateInstance`. A Spring Expression Language element defines the value of the property through the notation `#{ <expression> }`. For `@Value` annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

As of Spring Framework 4.3, you may also declare a factory method parameter of type `InjectionPoint` (or its more specific subclass: `DependencyDescriptor`) to access the requesting injection point that triggers the creation of the current bean. Note that this applies only to the actual creation of bean instances, not to the injection of existing instances. As a consequence, this feature makes most sense for beans of prototype scope. For other scopes, the factory method only ever sees the injection point that triggered the creation of a new bean instance in the given scope (for example, the dependency that triggered the creation of a lazy singleton bean). You can use the provided injection point metadata with semantic care in such scenarios. The following example shows how to use `InjectionPoint`:

Java

```
@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember());
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    @Bean
    @Scope("prototype")
    fun prototypeInstance(injectionPoint: InjectionPoint) =
            TestBean("prototypeInstance for ${injectionPoint.member}")
}
```

The `@Bean` methods in a regular Spring component are processed differently than their counterparts inside a Spring `@Configuration` class. The difference is that `@Component` classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within `@Bean` methods in `@Configuration` classes creates bean metadata references to collaborating objects. Such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans, even when referring to other beans through programmatic calls to `@Bean` methods. In contrast, invoking a method or field in a `@Bean` method within a plain `@Component` class has standard Java semantics, with no special CGLIB processing or other constraints applying.

You may declare `@Bean` methods as `static`, allowing for them to be called without creating their containing configuration class as an instance. This makes particular sense when defining post-processor beans (for example, of type `BeanFactoryPostProcessor` or `BeanPostProcessor`), since such beans get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.

Calls to static `@Bean` methods never get intercepted by the container, not even within `@Configuration` classes (as described earlier in this section), due to technical limitations: CGLIB subclassing can override only non-static methods. As a consequence, a direct call to another `@Bean` method has standard Java semantics, resulting in an independent instance being returned straight from the factory method itself.

The Java language visibility of `@Bean` methods does not have an immediate impact on the resulting bean definition in Spring's container. You can freely declare your factory methods as you see fit in non-`@Configuration` classes and also for static methods anywhere. However, regular `@Bean` methods in `@Configuration` classes need to be overridable — that is, they must not be declared as `private` or `final`.

`@Bean` methods are also discovered on base classes of a given component or configuration class, as well as on Java 8 default methods declared in interfaces implemented by the component or configuration class. This allows for a lot of flexibility in composing complex configuration arrangements, with even multiple inheritance being possible through Java 8 default methods as of Spring 4.2.

Finally, a single class may hold multiple `@Bean` methods for the same bean, as an arrangement of multiple factory methods to use depending on available dependencies at runtime. This is the same algorithm as for choosing the "greediest" constructor or factory method in other configuration scenarios: The variant with the largest number of satisfiable dependencies is picked at construction time, analogous to how the container selects between multiple `@Autowired` constructors.

Naming Autodetected Components

When a component is autodetected as part of the scanning process, its bean name is generated by the `BeanNameGenerator` strategy known to that scanner. By default, any Spring stereotype annotation (`@Component`, `@Repository`, `@Service`, and `@Controller`) that contains a name `value` thereby provides that name to the corresponding bean definition.

If such an annotation contains no name `value` or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following component classes were detected, the names would be `myMovieLister` and `movieFinderImpl`:

Java

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

Kotlin

```
@Service("myMovieLister")
class SimpleMovieLister {
    // ...
}
```

Java

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

Kotlin

```
@Repository
class MovieFinderImpl : MovieFinder {
    // ...
}
```

If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the `BeanNameGenerator` interface, and be sure to include a default no-arg constructor. Then, provide the fully qualified class name when configuring the scanner, as the following example annotation and bean definition show.

If you run into naming conflicts due to multiple autodetected components having the same non-qualified class name (i.e., classes with identical names but residing in different packages), you may need to configure a `BeanNameGenerator` that defaults to the fully qualified class name for the generated bean name. As of Spring Framework 5.2.3, the `FullyQualifiedAnnotationBeanNameGenerator` located in package `org.springframework.context.annotation` can be used for such purposes.

Java

```
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)
public class AppConfig {
    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["org.example"], nameGenerator = MyNameGenerator::class)
class AppConfig {
    // ...
}
```

```
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

Providing a Scope for Autodetected Components

As with Spring-managed components in general, the default and most common scope for autodetected components is `singleton`. However, sometimes you need a different scope that can be specified by the `@Scope` annotation. You can provide the name of the scope within the annotation, as the following example shows:

Java

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

Kotlin

```
@Scope("prototype")
@Repository
class MovieFinderImpl : MovieFinder {
    // ...
}
```

`@Scope` annotations are only introspected on the concrete bean class (for annotated components) or the factory method (for `@Bean` methods). In contrast to XML bean definitions, there is no notion of bean definition inheritance, and inheritance hierarchies at the class level are irrelevant for metadata purposes.

For details on web-specific scopes such as “request” or “session” in a Spring context, see Request, Session, Application, and WebSocket Scopes. As with the pre-built annotations for those scopes, you

may also compose your own scoping annotations by using Spring's meta-annotation approach: for example, a custom annotation meta-annotated with `@Scope("prototype")`, possibly also declaring a custom scoped-proxy mode.

To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, you can implement the `ScopeMetadataResolver` interface. Be sure to include a default no-arg constructor. Then you can provide the fully qualified class name when configuring the scanner, as the following example of both an annotation and a bean definition shows:

Java

```
@Configuration
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)
public class AppConfig {
    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["org.example"], scopeResolver = MyScopeResolver::class)
class AppConfig {
    // ...
}
```

```
<beans>
    <context:component-scan base-package="org.example" scope-
resolver="org.example.MyScopeResolver"/>
</beans>
```

When using certain non-singleton scopes, it may be necessary to generate proxies for the scoped objects. The reasoning is described in Scoped Beans as Dependencies. For this purpose, a scoped-proxy attribute is available on the component-scan element. The three possible values are: `no`, `interfaces`, and `targetClass`. For example, the following configuration results in standard JDK dynamic proxies:

Java

```
@Configuration
@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)
public class AppConfig {
    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["org.example"], scopedProxy =
ScopedProxyMode.INTERFACES)
class AppConfig {
    // ...
}
```

```
<beans>
    <context:component-scan base-package="org.example" scoped-proxy="interfaces"/>
</beans>
```

Providing Qualifier Metadata with Annotations

The `@Qualifier` annotation is discussed in Fine-tuning Annotation-based Autowiring with Qualifiers. The examples in that section demonstrate the use of the `@Qualifier` annotation and custom qualifier annotations to provide fine-grained control when you resolve autowire candidates. Because those examples were based on XML bean definitions, the qualifier metadata was provided on the candidate bean definitions by using the `qualifier` or `meta` child elements of the `bean` element in the XML. When relying upon classpath scanning for auto-detection of components, you can provide the qualifier metadata with type-level annotations on the candidate class. The following three examples demonstrate this technique:

Java

```
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

Kotlin

```
@Component
@Qualifier("Action")
class ActionMovieCatalog : MovieCatalog
```

Java

```
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

Kotlin

```
@Component
@Genre("Action")
class ActionMovieCatalog : MovieCatalog {
    // ...
}
```

Java

```
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

Kotlin

```
@Component
@Offline
class CachingMovieCatalog : MovieCatalog {
    // ...
}
```

As with most annotation-based alternatives, keep in mind that the annotation metadata is bound to the class definition itself, while the use of XML allows for multiple beans of the same type to provide variations in their qualifier metadata, because that metadata is provided per-instance rather than per-class.

Generating an Index of Candidate Components

While classpath scanning is very fast, it is possible to improve the startup performance of large applications by creating a static list of candidates at compilation time. In this mode, all modules that are targets of component scanning must use this mechanism.

Your existing `@ComponentScan` or `<context:component-scan/>` directives must remain unchanged to request the context to scan candidates in certain packages. When the `ApplicationContext` detects such an index, it automatically uses it rather than scanning the classpath.

To generate the index, add an additional dependency to each module that contains components that are targets for component scan directives. The following example shows how to do so with Maven:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-indexer</artifactId>
        <version>6.0.1</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

With Gradle 4.5 and earlier, the dependency should be declared in the `compileOnly` configuration, as shown in the following example:

```
dependencies {
    compileOnly "org.springframework:spring-context-indexer:6.0.1"
}
```

With Gradle 4.6 and later, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```
dependencies {
    annotationProcessor "org.springframework:spring-context-indexer:6.0.1"
}
```

The `spring-context-indexer` artifact generates a `META-INF/spring.components` file that is included in the jar file.

When working with this mode in your IDE, the `spring-context-indexer` must be registered as an annotation processor to make sure the index is up-to-date when candidate components are updated.

The index is enabled automatically when a `META-INF/spring.components` file is found on the classpath. If an index is partially available for some libraries (or use cases) but could not be built for the whole application, you can fall back to a regular classpath arrangement (as though no index were present at all) by setting `spring.index.ignore` to `true`, either as a JVM system property or via the `SpringProperties` mechanism.

2.1.11. Using JSR 330 Standard Annotations

Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. To use them, you need to have the relevant jars in your classpath.

If you use Maven, the `jakarta.inject` artifact is available in the standard Maven repository (https://repo1.maven.org/maven2/jakarta/inject/jakarta.inject-api/2.0.0/). You can add the following dependency to your file pom.xml:

```
<dependency>
    <groupId>jakarta.inject</groupId>
    <artifactId>jakarta.inject-api</artifactId>
    <version>1</version>
</dependency>
```

Dependency Injection with `@Inject` and `@Named`

Instead of `@Autowired`, you can use `@jakarta.inject.Inject` as follows:

Java

```
import jakarta.inject.Inject;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    public void listMovies() {
        this.movieFinder.findMovies(...);
        // ...
    }
}
```

Kotlin

```
import jakarta.inject.Inject

class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder


    fun listMovies() {
        movieFinder.findMovies(...)
        // ...
    }
}
```

As with `@Autowired`, you can use `@Inject` at the field level, method level and constructor-argument level. Furthermore, you may declare your injection point as a `Provider`, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a `Provider.get()` call. The following example offers a variant of the preceding example:

Java

```
import jakarta.inject.Inject;
import jakarta.inject.Provider;

public class SimpleMovieLister {

    private Provider<MovieFinder> movieFinder;

    @Inject
    public void setMovieFinder(Provider<MovieFinder> movieFinder) {
        this.movieFinder = movieFinder;
    }

    public void listMovies() {
        this.movieFinder.get().findMovies(...);
        // ...
    }
}
```

Kotlin

```
import jakarta.inject.Inject

class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: Provider<MovieFinder>


    fun listMovies() {
        movieFinder.get().findMovies(...)
        // ...
    }
}
```

If you would like to use a qualified name for the dependency that should be injected, you should use the `@Named` annotation, as the following example shows:

Java

```
import jakarta.inject.Inject;
import jakarta.inject.Named;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Kotlin

```
import jakarta.inject.Inject
import jakarta.inject.Named

class SimpleMovieLister {

    private lateinit var movieFinder: MovieFinder

    @Inject
    fun setMovieFinder(@Named("main") movieFinder: MovieFinder) {
        this.movieFinder = movieFinder
    }

    // ...
}
```

As with `@Autowired`, `@Inject` can also be used with `java.util.Optional` or `@Nullable`. This is even more applicable here, since `@Inject` does not have a `required` attribute. The following pair of examples show how to use `@Inject` and `@Nullable`:

```
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        // ...
    }
}
```

Java

```
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        // ...
    }
}
```

Kotlin

```
class SimpleMovieLister {

    @Inject
    var movieFinder: MovieFinder? = null
}
```

`@Named` and `@ManagedBean`: Standard Equivalents to the `@Component` Annotation

Instead of `@Component`, you can use `@jakarta.inject.Named` or `jakarta.annotation.ManagedBean`, as the following example shows:

Java

```
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named("movieListener")  // @ManagedBean("movieListener") could be used as well
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Kotlin

```
import jakarta.inject.Inject
import jakarta.inject.Named

@Named("movieListener")  // @ManagedBean("movieListener") could be used as well
class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    // ...
}
```

It is very common to use `@Component` without specifying a name for the component. `@Named` can be used in a similar fashion, as the following example shows:

Java

```
import jakarta.inject.Inject;
import jakarta.inject.Named;

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Kotlin

```
import jakarta.inject.Inject
import jakarta.inject.Named

@Named
class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    // ...
}
```

When you use `@Named` or `@ManagedBean`, you can use component scanning in the exact same way as when you use Spring annotations, as the following example shows:

Java

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["org.example"])
class AppConfig  {
    // ...
}
```

In contrast to `@Component`, the JSR-330 `@Named` and the JSR-250 `@ManagedBean` annotations are not composable. You should use Spring's stereotype model for building custom component annotations.

Limitations of JSR-330 Standard Annotations

When you work with standard annotations, you should know that some significant features are not available, as the following table shows:

Table 6. Spring component model elements versus JSR-330 variants

Spring	**jakarta.inject.***	**jakarta.inject restrictions / comments**
@Autowired	@Inject	`@Inject` has no 'required' attribute. Can be used with Java 8's `Optional` instead.
@Component	@Named / @ManagedBean	JSR-330 does not provide a composable model, only a way to identify named components.

Spring	jakarta.inject.*	jakarta.inject restrictions / comments
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's `prototype`. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a `singleton` by default. In order to use a scope other than `singleton`, you should use Spring's `@Scope` annotation. `jakarta.inject` also provides a `jakarta.inject.Scope` annotation: however, this one is only intended to be used for creating custom annotations.
@Qualifier	@Qualifier / @Named	`jakarta.inject.Qualifier` is just a meta-annotation for building custom qualifiers. Concrete `String` qualifiers (like Spring's `@Qualifier` with a value) can be associated through `jakarta.inject.Named`.
@Value	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	`jakarta.inject.Provider` is a direct alternative to Spring's `ObjectFactory`, only with a shorter `get()` method name. It can also be used in combination with Spring's `@Autowired` or with non-annotated constructors and setter methods.

2.1.12. Java-based Container Configuration

This section covers how to use annotations in your Java code to configure the Spring container. It includes the following topics:

- Basic Concepts: `@Bean` and `@Configuration`
- Instantiating the Spring Container by Using `AnnotationConfigApplicationContext`
- Using the `@Bean` Annotation
- Using the `@Configuration` annotation
- Composing Java-based Configurations

- Bean Definition Profiles
- `PropertySource` Abstraction
- Using `@PropertySource`
- Placeholder Resolution in Statements

Basic Concepts: `@Bean` and `@Configuration`

The central artifacts in Spring's Java configuration support are `@Configuration`-annotated classes and `@Bean`-annotated methods.

The `@Bean` annotation is used to indicate that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring's `<beans/>` XML configuration, the `@Bean` annotation plays the same role as the `<bean/>` element. You can use `@Bean`-annotated methods with any Spring `@Component`. However, they are most often used with `@Configuration` beans.

Annotating a class with `@Configuration` indicates that its primary purpose is as a source of bean definitions. Furthermore, `@Configuration` classes let inter-bean dependencies be defined by calling other `@Bean` methods in the same class. The simplest possible `@Configuration` class reads as follows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun myService(): MyService {
        return MyServiceImpl()
    }
}
```

The preceding `AppConfig` class is equivalent to the following Spring `<beans/>` XML:

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

Full @Configuration vs “lite” @Bean mode?

When `@Bean` methods are declared within classes that are not annotated with `@Configuration`, they are referred to as being processed in a “lite” mode. Bean methods declared in a `@Component` or even in a plain old class are considered to be “lite”, with a different primary purpose of the containing class and a `@Bean` method being a sort of bonus there. For example, service components may expose management views to the container through an additional `@Bean` method on each applicable component class. In such scenarios, `@Bean` methods are a general-purpose factory method mechanism.

Unlike full `@Configuration`, lite `@Bean` methods cannot declare inter-bean dependencies. Instead, they operate on their containing component’s internal state and, optionally, on arguments that they may declare. Such a `@Bean` method should therefore not invoke other `@Bean` methods. Each such method is literally only a factory method for a particular bean reference, without any special runtime semantics. The positive side-effect here is that no CGLIB subclassing has to be applied at runtime, so there are no limitations in terms of class design (that is, the containing class may be `final` and so forth).

In common scenarios, `@Bean` methods are to be declared within `@Configuration` classes, ensuring that “full” mode is always used and that cross-method references therefore get redirected to the container’s lifecycle management. This prevents the same `@Bean` method from accidentally being invoked through a regular Java call, which helps to reduce subtle bugs that can be hard to track down when operating in “lite” mode.

The `@Bean` and `@Configuration` annotations are discussed in depth in the following sections. First, however, we cover the various ways of creating a spring container by using Java-based configuration.

Instantiating the Spring Container by Using `AnnotationConfigApplicationContext`

The following sections document Spring’s `AnnotationConfigApplicationContext`, introduced in Spring 3.0. This versatile `ApplicationContext` implementation is capable of accepting not only `@Configuration` classes as input but also plain `@Component` classes and classes annotated with JSR-330 metadata.

When `@Configuration` classes are provided as input, the `@Configuration` class itself is registered as a bean definition and all declared `@Bean` methods within the class are also registered as bean definitions.

When `@Component` and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as `@Autowired` or `@Inject` are used within those classes where necessary.

Simple Construction

In much the same way that Spring XML files are used as input when instantiating a `ClassPathXmlApplicationContext`, you can use `@Configuration` classes as input when instantiating an `AnnotationConfigApplicationContext`. This allows for completely XML-free usage of the Spring

container, as the following example shows:

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(AppConfig::class.java)
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

As mentioned earlier, `AnnotationConfigApplicationContext` is not limited to working only with `@Configuration` classes. Any `@Component` or JSR-330 annotated class may be supplied as input to the constructor, as the following example shows:

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class,
Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(MyServiceImpl::class.java,
Dependency1::class.java, Dependency2::class.java)
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

The preceding example assumes that `MyServiceImpl`, `Dependency1`, and `Dependency2` use Spring dependency injection annotations such as `@Autowired`.

Building the Container Programmatically by Using `register(Class<?>…)`

You can instantiate an `AnnotationConfigApplicationContext` by using a no-arg constructor and then configure it by using the `register()` method. This approach is particularly useful when programmatically building an `AnnotationConfigApplicationContext`. The following example shows how to do so:

Java

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext()
    ctx.register(AppConfig::class.java, OtherConfig::class.java)
    ctx.register(AdditionalConfig::class.java)
    ctx.refresh()
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

Enabling Component Scanning with `scan(String…)`

To enable component scanning, you can annotate your `@Configuration` class as follows:

Java

```
@Configuration
@ComponentScan(basePackages = "com.acme") ①
public class AppConfig  {
    // ...
}
```

① This annotation enables component scanning.

Kotlin

```
@Configuration
@ComponentScan(basePackages = ["com.acme"]) ①
class AppConfig  {
    // ...
}
```

① This annotation enables component scanning.

Experienced Spring users may be familiar with the XML declaration equivalent from Spring's `context:` namespace, shown in the following example:

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

In the preceding example, the `com.acme` package is scanned to look for any `@Component`-annotated classes, and those classes are registered as Spring bean definitions within the container. `AnnotationConfigApplicationContext` exposes the `scan(String…)` method to allow for the same component-scanning functionality, as the following example shows:

Java

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

Kotlin

```
fun main() {
    val ctx = AnnotationConfigApplicationContext()
    ctx.scan("com.acme")
    ctx.refresh()
    val myService = ctx.getBean<MyService>()
}
```

Remember that `@Configuration` classes are meta-annotated with `@Component`, so they are candidates for component-scanning. In the preceding example, assuming that `AppConfig` is declared within the `com.acme` package (or any package underneath), it is picked up during the call to `scan()`. Upon `refresh()`, all its `@Bean` methods are processed and registered as bean definitions within the container.

Support for Web Applications with `AnnotationConfigWebApplicationContext`

A `WebApplicationContext` variant of `AnnotationConfigApplicationContext` is available with `AnnotationConfigWebApplicationContext`. You can use this implementation when configuring the Spring `ContextLoaderListener` servlet listener, Spring MVC `DispatcherServlet`, and so forth. The following `web.xml` snippet configures a typical Spring MVC web application (note the use of the `contextClass` context-param and init-param):

```
<web-app>
    <!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-delimited
        fully-qualified @Configuration classes. Fully-qualified packages may also be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using ContextLoaderListener
-->
    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext
            instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more comma- or space-
delimited
            and fully-qualified @Configuration classes -->
```

```
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.acme.web.MvcConfig</param-value>
        </init-param>
    </servlet>

    <!-- map all requests for /app/* to the dispatcher servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>
</web-app>
```

For programmatic use cases, a `GenericWebApplicationContext` can be used as an alternative to `AnnotationConfigWebApplicationContext`. See the `GenericWebApplicationContext` javadoc for details.

Using the `@Bean` Annotation

`@Bean` is a method-level annotation and a direct analog of the XML `<bean/>` element. The annotation supports some of the attributes offered by `<bean/>`, such as:

- init-method
- destroy-method
- autowiring
- `name`.

You can use the `@Bean` annotation in a `@Configuration`-annotated or in a `@Component`-annotated class.

Declaring a Bean

To declare a bean, you can annotate a method with the `@Bean` annotation. You use this method to register a bean definition within an `ApplicationContext` of the type specified as the method's return value. By default, the bean name is the same as the method name. The following example shows a `@Bean` method declaration:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService() = TransferServiceImpl()
}
```

The preceding configuration is exactly equivalent to the following Spring XML:

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

Both declarations make a bean named `transferService` available in the `ApplicationContext`, bound to an object instance of type `TransferServiceImpl`, as the following text image shows:

```
transferService -> com.acme.TransferServiceImpl
```

You can also use default methods to define beans. This allows composition of bean configurations by implementing interfaces with bean definitions on default methods.

Java

```
public interface BaseConfig {

    @Bean
    default TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}

@Configuration
public class AppConfig implements BaseConfig {

}
```

You can also declare your `@Bean` method with an interface (or base class) return type, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl()
    }
}
```

However, this limits the visibility for advance type prediction to the specified interface type (`TransferService`). Then, with the full type (`TransferServiceImpl`) known to the container only once the affected singleton bean has been instantiated. Non-lazy singleton beans get instantiated according to their declaration order, so you may see different type matching results depending on when another component tries to match by a non-declared type (such as `@Autowired TransferServiceImpl`, which resolves only once the `transferService` bean has been instantiated).

If you consistently refer to your types by a declared service interface, your `@Bean` return types may safely join that design decision. However, for components that implement several interfaces or for components potentially referred to by their implementation type, it is safer to declare the most specific return type possible (at least as specific as required by the injection points that refer to your bean).

Bean Dependencies

A `@Bean`-annotated method can have an arbitrary number of parameters that describe the dependencies required to build that bean. For instance, if our `TransferService` requires an `AccountRepository`, we can materialize that dependency with a method parameter, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService(accountRepository: AccountRepository): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}
```

The resolution mechanism is pretty much identical to constructor-based dependency injection. See the relevant section for more details.

Receiving Lifecycle Callbacks

Any classes defined with the `@Bean` annotation support the regular lifecycle callbacks and can use the `@PostConstruct` and `@PreDestroy` annotations from JSR-250. See JSR-250 annotations for further details.

The regular Spring lifecycle callbacks are fully supported as well. If a bean implements `InitializingBean`, `DisposableBean`, or `Lifecycle`, their respective methods are called by the container.

The standard set of `*Aware` interfaces (such as BeanFactoryAware, BeanNameAware, MessageSourceAware, ApplicationContextAware, and so on) are also fully supported.

The `@Bean` annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's `init-method` and `destroy-method` attributes on the `bean` element, as the following example shows:

Java

```
public class BeanOne {

    public void init() {
        // initialization logic
    }
}

public class BeanTwo {

    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public BeanOne beanOne() {
        return new BeanOne();
    }

    @Bean(destroyMethod = "cleanup")
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

Kotlin

```
class BeanOne {

    fun init() {
        // initialization logic
    }
}

class BeanTwo {

    fun cleanup() {
        // destruction logic
    }
}

@Configuration
class AppConfig {

    @Bean(initMethod = "init")
    fun beanOne() = BeanOne()

    @Bean(destroyMethod = "cleanup")
    fun beanTwo() = BeanTwo()
}
```

By default, beans defined with Java configuration that have a public `close` or `shutdown` method are automatically enlisted with a destruction callback. If you have a public `close` or `shutdown` method and you do not wish for it to be called when the container shuts down, you can add `@Bean(destroyMethod = "")` to your bean definition to disable the default `(inferred)` mode.

You may want to do that by default for a resource that you acquire with JNDI, as its lifecycle is managed outside the application. In particular, make sure to always do it for a `DataSource`, as it is known to be problematic on Jakarta EE application servers.

The following example shows how to prevent an automatic destruction callback for a `DataSource`:

Java

```
@Bean(destroyMethod = "")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

Kotlin

```
@Bean(destroyMethod = "")
fun dataSource(): DataSource {
    return jndiTemplate.lookup("MyDS") as DataSource
}
```

Also, with `@Bean` methods, you typically use programmatic JNDI lookups, either by using Spring's `JndiTemplate` or `JndiLocatorDelegate` helpers or straight JNDI `InitialContext` usage but not the `JndiObjectFactoryBean` variant (which would force you to declare the return type as the `FactoryBean` type instead of the actual target type, making it harder to use for cross-reference calls in other `@Bean` methods that intend to refer to the provided resource here).

In the case of `BeanOne` from the example above the preceding note, it would be equally valid to call the `init()` method directly during construction, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        BeanOne beanOne = new BeanOne();
        beanOne.init();
        return beanOne;
    }

    // ...
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun beanOne() = BeanOne().apply {
        init()
    }

    // ...
}
```

When you work directly in Java, you can do anything you like with your objects and do not always need to rely on the container lifecycle.

Specifying Bean Scope

Spring includes the `@Scope` annotation so that you can specify the scope of a bean.

Using the `@Scope` Annotation

You can specify that your beans defined with the `@Bean` annotation should have a specific scope. You can use any of the standard scopes specified in the Bean Scopes section.

The default scope is `singleton`, but you can override this with the `@Scope` annotation, as the following example shows:

Java

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }
}
```

Kotlin

```
@Configuration
class MyConfiguration {

    @Bean
    @Scope("prototype")
    fun encryptor(): Encryptor {
        // ...
    }
}
```

`@Scope` and `scoped-proxy`

Spring offers a convenient way of working with scoped dependencies through scoped proxies. The easiest way to create such a proxy when using the XML configuration is the `<aop:scoped-proxy/>` element. Configuring your beans in Java with a `@Scope` annotation offers equivalent support with the `proxyMode` attribute. The default is `ScopedProxyMode.DEFAULT`, which typically indicates that no scoped proxy should be created unless a different default has been configured at the component-scan instruction level. You can specify `ScopedProxyMode.TARGET_CLASS`, `ScopedProxyMode.INTERFACES` or `ScopedProxyMode.NO`.

If you port the scoped proxy example from the XML reference documentation (see scoped proxies) to our `@Bean` using Java, it resembles the following:

Java

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
public UserPreferences userPreferences() {
    return new UserPreferences();
}

@Bean
public Service userService() {
    UserService service = new SimpleUserService();
    // a reference to the proxied userPreferences bean
    service.setUserPreferences(userPreferences());
    return service;
}
```

Kotlin

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
fun userPreferences() = UserPreferences()

@Bean
fun userService(): Service {
    return SimpleUserService().apply {
        // a reference to the proxied userPreferences bean
        setUserPreferences(userPreferences())
    }
}
```

Customizing Bean Naming

By default, configuration classes use a `@Bean` method's name as the name of the resulting bean. This functionality can be overridden, however, with the `name` attribute, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean("myThing")
    public Thing thing() {
        return new Thing();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean("myThing")
    fun thing() = Thing()
}
```

Bean Aliasing

As discussed in Naming Beans, it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing. The `name` attribute of the `@Bean` annotation accepts a String array for this purpose. The following example shows how to set a number of aliases for a bean:

Java

```
@Configuration
public class AppConfig {

    @Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean("dataSource", "subsystemA-dataSource", "subsystemB-dataSource")
    fun dataSource(): DataSource {
        // instantiate, configure and return DataSource bean...
    }
}
```

Bean Description

Sometimes, it is helpful to provide a more detailed textual description of a bean. This can be particularly useful when beans are exposed (perhaps through JMX) for monitoring purposes.

To add a description to a `@Bean`, you can use the `@Description` annotation, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Thing thing() {
        return new Thing();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    fun thing() = Thing()
}
```

Using the `@Configuration` annotation

`@Configuration` is a class-level annotation indicating that an object is a source of bean definitions. `@Configuration` classes declare beans through `@Bean`-annotated methods. Calls to `@Bean` methods on `@Configuration` classes can also be used to define inter-bean dependencies. See Basic Concepts: `@Bean` and `@Configuration` for a general introduction.

Injecting Inter-bean Dependencies

When beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another, as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        return new BeanOne(beanTwo());
    }

    @Bean
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun beanOne() = BeanOne(beanTwo())

    @Bean
    fun beanTwo() = BeanTwo()
}
```

In the preceding example, `beanOne` receives a reference to `beanTwo` through constructor injection.

This method of declaring inter-bean dependencies works only when the `@Bean` method is declared within a `@Configuration` class. You cannot declare inter-bean dependencies by using plain `@Component` classes.

Lookup Method Injection

As noted earlier, lookup method injection is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern. The following example shows how to use lookup method injection:

Java

```
public abstract class CommandManager {
    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

Kotlin

```
abstract class CommandManager {
    fun process(commandState: Any): Any {
        // grab a new instance of the appropriate Command interface
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState)
        return command.execute()
    }

    // okay... but where is the implementation of this method?
    protected abstract fun createCommand(): Command
}
```

By using Java configuration, you can create a subclass of `CommandManager` where the abstract `createCommand()` method is overridden in such a way that it looks up a new (prototype) command object. The following example shows how to do so:

Java

```
@Bean
@Scope("prototype")
public AsyncCommand asyncCommand() {
    AsyncCommand command = new AsyncCommand();
    // inject dependencies here as required
    return command;
}

@Bean
public CommandManager commandManager() {
    // return new anonymous implementation of CommandManager with createCommand()
    // overridden to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
        }
    }
}
```

Kotlin

```
@Bean
@Scope("prototype")
fun asyncCommand(): AsyncCommand {
    val command = AsyncCommand()
    // inject dependencies here as required
    return command
}

@Bean
fun commandManager(): CommandManager {
    // return new anonymous implementation of CommandManager with createCommand()
    // overridden to return a new prototype Command object
    return object : CommandManager() {
        override fun createCommand(): Command {
            return asyncCommand()
        }
    }
}
```

Further Information About How Java-based Configuration Works Internally

Consider the following example, which shows a `@Bean` annotated method being called twice:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun clientService1(): ClientService {
        return ClientServiceImpl().apply {
            clientDao = clientDao()
        }
    }

    @Bean
    fun clientService2(): ClientService {
        return ClientServiceImpl().apply {
            clientDao = clientDao()
        }
    }

    @Bean
    fun clientDao(): ClientDao {
        return ClientDaoImpl()
    }
}
```

`clientDao()` has been called once in `clientService1()` and once in `clientService2()`. Since this method creates a new instance of `ClientDaoImpl` and returns it, you would normally expect to have two instances (one for each service). That definitely would be problematic: In Spring, instantiated beans have a `singleton` scope by default. This is where the magic comes in: All `@Configuration` classes are subclassed at startup-time with `CGLIB`. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

The behavior could be different according to the scope of your bean. We are talking about singletons here.

As of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under `org.springframework.cglib` and included directly within the spring-core JAR.

There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time. In particular, configuration classes must not be final. However, as of 4.3, any constructors are allowed on configuration classes, including the use of `@Autowired` or a single non-default constructor declaration for default injection.

If you prefer to avoid any CGLIB-imposed limitations, consider declaring your `@Bean` methods on non-`@Configuration` classes (for example, on plain `@Component` classes instead). Cross-method calls between `@Bean` methods are not then intercepted, so you have to exclusively rely on dependency injection at the constructor or method level there.

Composing Java-based Configurations

Spring's Java-based configuration feature lets you compose annotations, which can reduce the complexity of your configuration.

Using the `@Import` Annotation

Much as the `<import/>` element is used within Spring XML files to aid in modularizing configurations, the `@Import` annotation allows for loading `@Bean` definitions from another configuration class, as the following example shows:

Java

```
@Configuration
public class ConfigA {

    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }
}
```

Kotlin

```
@Configuration
class ConfigA {

    @Bean
    fun a() = A()
}

@Configuration
@Import(ConfigA::class)
class ConfigB {

    @Bean
    fun b() = B()
}
```

Now, rather than needing to specify both `ConfigA.class` and `ConfigB.class` when instantiating the context, only `ConfigB` needs to be supplied explicitly, as the following example shows:

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

    // now both beans A and B will be available...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(ConfigB::class.java)

    // now both beans A and B will be available...
    val a = ctx.getBean<A>()
    val b = ctx.getBean<B>()
}
```

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring you to remember a potentially large number of `@Configuration` classes during construction.

As of Spring Framework 4.2, `@Import` also supports references to regular component classes, analogous to the `AnnotationConfigApplicationContext.register` method. This is particularly useful if you want to avoid component scanning, by using a few configuration classes as entry points to explicitly define all your components.

Injecting Dependencies on Imported `@Bean` Definitions

The preceding example works but is simplistic. In most practical scenarios, beans have dependencies on one another across configuration classes. When using XML, this is not an issue, because no compiler is involved, and you can declare `ref="someBean"` and trust Spring to work it out during container initialization. When using `@Configuration` classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

Fortunately, solving this problem is simple. As we already discussed, a `@Bean` method can have an arbitrary number of parameters that describe the bean dependencies. Consider the following more real-world scenario with several `@Configuration` classes, each depending on beans declared in the others:

Java

```
@Configuration
public class ServiceConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}

@Configuration
public class RepositoryConfig {

    @Bean
    public AccountRepository accountRepository(DataSource dataSource) {
        return new JdbcAccountRepository(dataSource);
    }
}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }
}

public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Bean
    fun transferService(accountRepository: AccountRepository): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}

@Configuration
class RepositoryConfig {

    @Bean
    fun accountRepository(dataSource: DataSource): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }
}

@Configuration
@Import(ServiceConfig::class, RepositoryConfig::class)
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return new DataSource
    }
}


fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    // everything wires up across configuration classes...
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

There is another way to achieve the same result. Remember that `@Configuration` classes are ultimately only another bean in the container: This means that they can take advantage of `@Autowired` and `@Value` injection and other features the same as any other bean.

Make sure that the dependencies you inject that way are of the simplest kind only. `@Configuration` classes are processed quite early during the initialization of the context, and forcing a dependency to be injected this way may lead to unexpected early initialization. Whenever possible, resort to parameter-based injection, as in the preceding example.

Also, be particularly careful with `BeanPostProcessor` and `BeanFactoryPostProcessor` definitions through `@Bean`. Those should usually be declared as `static @Bean` methods, not triggering the instantiation of their containing configuration class. Otherwise, `@Autowired` and `@Value` may not work on the configuration class itself, since it is possible to create it as a bean instance earlier than `AutowiredAnnotationBeanPostProcessor`.

The following example shows how one bean can be autowired to another bean:

Java

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }
}

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    public RepositoryConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }
}

public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Autowired
    lateinit var accountRepository: AccountRepository

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}

@Configuration
class RepositoryConfig(private val dataSource: DataSource) {

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }
}

@Configuration
@Import(ServiceConfig::class, RepositoryConfig::class)
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return new DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    // everything wires up across configuration classes...
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

Constructor injection in `@Configuration` classes is only supported as of Spring Framework 4.3. Note also that there is no need to specify `@Autowired` if the target bean defines only one constructor.

Fully-qualifying imported beans for ease of navigation

In the preceding scenario, using `@Autowired` works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat

ambiguous. For example, as a developer looking at `ServiceConfig`, how do you know exactly where the `@Autowired AccountRepository` bean is declared? It is not explicit in the code, and this may be just fine. Remember that the Spring Tools for Eclipse provides tooling that can render graphs showing how everything is wired, which may be all you need. Also, your Java IDE can easily find all declarations and uses of the `AccountRepository` type and quickly show you the location of `@Bean` methods that return that type.

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one `@Configuration` class to another, consider autowiring the configuration classes themselves. The following example shows how to do so:

Java

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        // navigate 'through' the config class to the @Bean method!
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}
```

Kotlin

```
@Configuration
class ServiceConfig {

    @Autowired
    private lateinit var repositoryConfig: RepositoryConfig

    @Bean
    fun transferService(): TransferService {
        // navigate 'through' the config class to the @Bean method!
        return TransferServiceImpl(repositoryConfig.accountRepository())
    }
}
```

In the preceding situation, where `AccountRepository` is defined is completely explicit. However, `ServiceConfig` is now tightly coupled to `RepositoryConfig`. That is the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based `@Configuration` classes. Consider the following example:

Java

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}

@Configuration
public interface RepositoryConfig {

    @Bean
    AccountRepository accountRepository();
}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig {

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }
}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class})  // import the concrete
config!
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new
AnnotationConfigApplicationContext(SystemTestConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Autowired
    private lateinit var repositoryConfig: RepositoryConfig

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl(repositoryConfig.accountRepository())
    }
}

@Configuration
interface RepositoryConfig {

    @Bean
    fun accountRepository(): AccountRepository
}

@Configuration
class DefaultRepositoryConfig : RepositoryConfig {

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(...)
    }
}

@Configuration
@Import(ServiceConfig::class, DefaultRepositoryConfig::class)  // import the concrete
config!
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return DataSource
    }

}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

Now `ServiceConfig` is loosely coupled with respect to the concrete `DefaultRepositoryConfig`, and built-in IDE tooling is still useful: You can easily get a type hierarchy of `RepositoryConfig` implementations. In this way, navigating `@Configuration` classes and their dependencies becomes no different than the usual process of navigating interface-based code.

If you want to influence the startup creation order of certain beans, consider declaring some of them as `@Lazy` (for creation on first access instead of on startup) or as `@DependsOn` certain other beans (making sure that specific other beans are created before the current bean, beyond what the latter's direct dependencies imply).

Conditionally Include `@Configuration` Classes or `@Bean` Methods

It is often useful to conditionally enable or disable a complete `@Configuration` class or even individual `@Bean` methods, based on some arbitrary system state. One common example of this is to use the `@Profile` annotation to activate beans only when a specific profile has been enabled in the Spring `Environment` (see Bean Definition Profiles for details).

The `@Profile` annotation is actually implemented by using a much more flexible annotation called `@Conditional`. The `@Conditional` annotation indicates specific `org.springframework.context.annotation.Condition` implementations that should be consulted before a `@Bean` is registered.

Implementations of the `Condition` interface provide a `matches(…)` method that returns `true` or `false`. For example, the following listing shows the actual `Condition` implementation used for `@Profile`:

Java

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    // Read the @Profile annotation attributes
    MultiValueMap<String, Object> attrs =
metadata.getAllAnnotationAttributes(Profile.class.getName());
    if (attrs != null) {
        for (Object value : attrs.get("value")) {
            if (context.getEnvironment().acceptsProfiles(((String[]) value))) {
                return true;
            }
        }
        return false;
    }
    return true;
}
```

Kotlin

```
override fun matches(context: ConditionContext, metadata: AnnotatedTypeMetadata):
Boolean {
    // Read the @Profile annotation attributes
    val attrs = metadata.getAllAnnotationAttributes(Profile::class.java.name)
    if (attrs != null) {
        for (value in attrs["value"]!!) {
            if (context.environment.acceptsProfiles(Profiles.of(*value as
Array<String>))) {
                return true
            }
        }
        return false
    }
    return true
}
```

See the `@Conditional` javadoc for more detail.

Combining Java and XML Configuration

Spring's `@Configuration` class support does not aim to be a 100% complete replacement for Spring XML. Some facilities, such as Spring XML namespaces, remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an "XML-centric" way by using, for example, `ClassPathXmlApplicationContext`, or instantiate it in a "Java-centric" way by using `AnnotationConfigApplicationContext` and the `@ImportResource` annotation to import XML as needed.

XML-centric Use of `@Configuration` Classes

It may be preferable to bootstrap the Spring container from XML and include `@Configuration` classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it is easier to create `@Configuration` classes on an as-needed basis and include them from the existing XML files. Later in this section, we cover the options for using `@Configuration` classes in this kind of "XML-centric" situation.

Declaring `@Configuration` classes as plain Spring `<bean/>` elements

Remember that `@Configuration` classes are ultimately bean definitions in the container. In this series examples, we create a `@Configuration` class named `AppConfig` and include it within `system-test-config.xml` as a `<bean/>` definition. Because `<context:annotation-config/>` is switched on, the container recognizes the `@Configuration` annotation and processes the `@Bean` methods declared in `AppConfig` properly.

The following example shows an ordinary configuration class in Java:

Java

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Autowired
    private lateinit var dataSource: DataSource

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }

    @Bean
    fun transferService() = TransferService(accountRepository())
}
```

The following example shows part of a sample `system-test-config.xml` file:

```
<beans>
    <!-- enable processing of annotations such as @Autowired and @Configuration -->
    <context:annotation-config/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="com.acme.AppConfig"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

The following example shows a possible `jdbc.properties` file:

```
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

Kotlin

```
fun main() {
    val ctx = ClassPathXmlApplicationContext("classpath:/com/acme/system-test-
config.xml")
    val transferService = ctx.getBean<TransferService>()
    // ...
}
```

In `system-test-config.xml` file, the `AppConfig <bean/>` does not declare an `id` element. While it would be acceptable to do so, it is unnecessary, given that no other bean ever refers to it, and it is unlikely to be explicitly fetched from the container by name. Similarly, the `DataSource` bean is only ever autowired by type, so an explicit bean `id` is not strictly required.

Using <context:component-scan/> to pick up `@Configuration` classes

Because `@Configuration` is meta-annotated with `@Component`, `@Configuration`-annotated classes are

automatically candidates for component scanning. Using the same scenario as described in the previous example, we can redefine `system-test-config.xml` to take advantage of component-scanning. Note that, in this case, we need not explicitly declare `<context:annotation-config/>`, because `<context:component-scan/>` enables the same functionality.

The following example shows the modified `system-test-config.xml` file:

```
<beans>
    <!-- picks up and registers AppConfig as a bean definition -->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

`@Configuration` Class-centric Use of XML with `@ImportResource`

In applications where `@Configuration` classes are the primary mechanism for configuring the container, it is still likely necessary to use at least some XML. In these scenarios, you can use `@ImportResource` and define only as much XML as you need. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum. The following example (which includes a configuration class, an XML file that defines a bean, a properties file, and the `main` class) shows how to use the `@ImportResource` annotation to achieve "Java-centric" configuration that uses XML as needed:

Java

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
}
```

Kotlin

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
class AppConfig {

    @Value("\${jdbc.url}")
    private lateinit var url: String

    @Value("\${jdbc.username}")
    private lateinit var username: String

    @Value("\${jdbc.password}")
    private lateinit var password: String

    @Bean
    fun dataSource(): DataSource {
        return DriverManagerDataSource(url, username, password)
    }
}
```

```
properties-config.xml
<beans>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>
</beans>
```

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(AppConfig::class.java)
    val transferService = ctx.getBean<TransferService>()
    // ...
}
```

2.1.13. Environment Abstraction

The `Environment` interface is an abstraction integrated in the container that models two key aspects of the application environment: profiles and properties.

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active. Beans may be assigned to a profile whether defined in XML or with annotations. The role of the `Environment` object with relation to profiles is in determining which profiles (if any) are currently active, and which profiles (if any) should be active by default.

Properties play an important role in almost all applications and may originate from a variety of sources: properties files, JVM system properties, system environment variables, JNDI, servlet context parameters, ad-hoc `Properties` objects, `Map` objects, and so on. The role of the `Environment` object with relation to properties is to provide the user with a convenient service interface for configuring property sources and resolving properties from them.

Bean Definition Profiles

Bean definition profiles provide a mechanism in the core container that allows for registration of different beans in different environments. The word, “environment,” can mean different things to different users, and this feature can help with many use cases, including:

- Working against an in-memory datasource in development versus looking up that same datasource from JNDI when in QA or production.
- Registering monitoring infrastructure only when deploying an application into a performance environment.
- Registering customized implementations of beans for customer A versus customer B deployments.

Consider the first use case in a practical application that requires a `DataSource`. In a test environment, the configuration might resemble the following:

Java

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

Kotlin

```
@Bean
fun dataSource(): DataSource {
    return EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("my-schema.sql")
            .addScript("my-test-data.sql")
            .build()
}
```

Now consider how this application can be deployed into a QA or production environment, assuming that the datasource for the application is registered with the production application server's JNDI directory. Our `dataSource` bean now looks like the following listing:

Java

```
@Bean(destroyMethod = "")
public DataSource dataSource() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
}
```

Kotlin

```
@Bean(destroyMethod = "")
fun dataSource(): DataSource {
    val ctx = InitialContext()
    return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource
}
```

The problem is how to switch between using these two variations based on the current environment. Over time, Spring users have devised a number of ways to get this done, usually relying on a combination of system environment variables and XML `<import/>` statements containing `${placeholder}` tokens that resolve to the correct configuration file path depending on the value of an environment variable. Bean definition profiles is a core container feature that provides a solution to this problem.

If we generalize the use case shown in the preceding example of environment-specific bean definitions, we end up with the need to register certain bean definitions in certain contexts but not in others. You could say that you want to register a certain profile of bean definitions in situation A and a different profile in situation B. We start by updating our configuration to reflect this need.

Using `@Profile`

The `@Profile` annotation lets you indicate that a component is eligible for registration when one or more specified profiles are active. Using our preceding example, we can rewrite the `dataSource` configuration as follows:

Java

```
@Configuration
@Profile("development")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

Kotlin

```
@Configuration
@Profile("development")
class StandaloneDataConfig {

    @Bean
    fun dataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/bank/config/sql/schema.sql")
                .addScript("classpath:com/bank/config/sql/test-data.sql")
                .build()
    }
}
```

Java

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean(destroyMethod = "") ①
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

① `@Bean(destroyMethod = "")` disables default destroy method inference.

Kotlin

```
@Configuration
@Profile("production")
class JndiDataConfig {

    @Bean(destroyMethod = "") ①
    fun dataSource(): DataSource {
        val ctx = InitialContext()
        return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource
    }
}
```

① `@Bean(destroyMethod = "")` disables default destroy method inference.

As mentioned earlier, with `@Bean` methods, you typically choose to use programmatic JNDI lookups, by using either Spring's `JndiTemplate`/`JndiLocatorDelegate` helpers or the straight JNDI `InitialContext` usage shown earlier but not the `JndiObjectFactoryBean` variant, which would force you to declare the return type as the `FactoryBean` type.

The profile string may contain a simple profile name (for example, `production`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed (for example, `production & us-east`). The following operators are supported in profile expressions:

- `!`: A logical `NOT` of the profile
- `&`: A logical `AND` of the profiles
- `|`: A logical `OR` of the profiles

You cannot mix the `&` and `|` operators without using parentheses. For example, `production & us-east | eu-central` is not a valid expression. It must be expressed as `production & (us-east | eu-central)`.

You can use `@Profile` as a meta-annotation for the purpose of creating a custom composed

annotation. The following example defines a custom `@Production` annotation that you can use as a drop-in replacement for `@Profile("production")`:

Java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

Kotlin

```
@Target(AnnotationTarget.CLASS)
@Retention(AnnotationRetention.RUNTIME)
@Profile("production")
annotation class Production
```

If a `@Configuration` class is marked with `@Profile`, all of the `@Bean` methods and `@Import` annotations associated with that class are bypassed unless one or more of the specified profiles are active. If a `@Component` or `@Configuration` class is marked with `@Profile({"p1", "p2"})`, that class is not registered or processed unless profiles 'p1' or 'p2' have been activated. If a given profile is prefixed with the NOT operator (`!`), the annotated element is registered only if the profile is not active. For example, given `@Profile({"p1", "!p2"})`, registration will occur if profile 'p1' is active or if profile 'p2' is not active.

`@Profile` can also be declared at the method level to include only one particular bean of a configuration class (for example, for alternative variants of a particular bean), as the following example shows:

Java

```
@Configuration
public class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    public DataSource standaloneDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }

    @Bean("dataSource")
    @Profile("production") ②
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

① The `standaloneDataSource` method is available only in the `development` profile.

② The `jndiDataSource` method is available only in the `production` profile.

Kotlin

```
@Configuration
class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    fun standaloneDataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/bank/config/sql/schema.sql")
                .addScript("classpath:com/bank/config/sql/test-data.sql")
                .build()
    }

    @Bean("dataSource")
    @Profile("production") ②
    fun jndiDataSource() =
        InitialContext().lookup("java:comp/env/jdbc/datasource") as DataSource
}
```

① The `standaloneDataSource` method is available only in the `development` profile.

② The `jndiDataSource` method is available only in the `production` profile.

With `@Profile` on `@Bean` methods, a special scenario may apply: In the case of overloaded `@Bean` methods of the same Java method name (analogous to constructor overloading), a `@Profile` condition needs to be consistently declared on all overloaded methods. If the conditions are inconsistent, only the condition on the first declaration among the overloaded methods matters. Therefore, `@Profile` can not be used to select an overloaded method with a particular argument signature over another. Resolution between all factory methods for the same bean follows Spring's constructor resolution algorithm at creation time.

If you want to define alternative beans with different profile conditions, use distinct Java method names that point to the same bean name by using the `@Bean` name attribute, as shown in the preceding example. If the argument signatures are all the same (for example, all of the variants have no-arg factory methods), this is the only way to represent such an arrangement in a valid Java class in the first place (since there can only be one method of a particular name and argument signature).

XML Bean Definition Profiles

The XML counterpart is the `profile` attribute of the `<beans>` element. Our preceding sample configuration can be rewritten in two XML files, as follows:

```
<beans profile="development"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="...">

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

```
<beans profile="production"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

It is also possible to avoid that split and nest `<beans/>` elements within the same file, as the following example shows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <!-- other bean definitions -->

    <beans profile="development">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
    </beans>
</beans>
```

The `spring-bean.xsd` has been constrained to allow such elements only as the last ones in the file. This should help provide flexibility without incurring clutter in the XML files.

The XML counterpart does not support the profile expressions described earlier. It is possible, however, to negate a profile by using the `!` operator. It is also possible to apply a logical “and” by nesting the profiles, as the following example shows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <!-- other bean definitions -->

    <beans profile="production">
        <beans profile="us-east">
            <jee:jndi-lookup id="dataSource" jndi-
name="java:comp/env/jdbc/datasource"/>
        </beans>
    </beans>
</beans>
```

In the preceding example, the `dataSource` bean is exposed if both the `production` and `us-east` profiles are active.

Activating a Profile

Now that we have updated our configuration, we still need to instruct Spring which profile is active. If we started our sample application right now, we would see a `NoSuchBeanDefinitionException` thrown, because the container could not find the Spring bean named `dataSource`.

Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the `Environment` API which is available through an `ApplicationContext`. The following example shows how to do so:

Java

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

Kotlin

```
val ctx = AnnotationConfigApplicationContext().apply {
    environment.setActiveProfiles("development")
    register(SomeConfig::class.java, StandaloneDataConfig::class.java,
JndiDataConfig::class.java)
    refresh()
}
```

In addition, you can also declaratively activate profiles through the `spring.profiles.active` property, which may be specified through system environment variables, JVM system properties, servlet context parameters in `web.xml`, or even as an entry in JNDI (see `PropertySource` Abstraction). In integration tests, active profiles can be declared by using the `@ActiveProfiles` annotation in the `spring-test` module (see context configuration with environment profiles).

Note that profiles are not an “either-or” proposition. You can activate multiple profiles at once. Programmatically, you can provide multiple profile names to the `setActiveProfiles()` method, which accepts `String…` varargs. The following example activates multiple profiles:

Java

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

Kotlin

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2")
```

Declaratively, `spring.profiles.active` may accept a comma-separated list of profile names, as the following example shows:

```
-Dspring.profiles.active="profile1,profile2"
```

Default Profile

The default profile represents the profile that is enabled by default. Consider the following example:

Java

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

Kotlin

```
@Configuration
@Profile("default")
class DefaultDataConfig {

    @Bean
    fun dataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/bank/config/sql/schema.sql")
                .build()
    }
}
```

If no profile is active, the `dataSource` is created. You can see this as a way to provide a default definition for one or more beans. If any profile is enabled, the default profile does not apply.

You can change the name of the default profile by using `setDefaultProfiles()` on the `Environment` or, declaratively, by using the `spring.profiles.default` property.

`PropertySource` Abstraction

Spring's `Environment` abstraction provides search operations over a configurable hierarchy of property sources. Consider the following listing:

Java

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsMyProperty = env.containsProperty("my-property");
System.out.println("Does my environment contain the 'my-property' property? " +
containsMyProperty);
```

Kotlin

```
val ctx = GenericApplicationContext()
val env = ctx.environment
val containsMyProperty = env.containsProperty("my-property")
println("Does my environment contain the 'my-property' property? $containsMyProperty")
```

In the preceding snippet, we see a high-level way of asking Spring whether the `my-property` property is defined for the current environment. To answer this question, the `Environment` object performs a search over a set of `PropertySource` objects. A `PropertySource` is a simple abstraction over any source of key-value pairs, and Spring's `StandardEnvironment` is configured with two PropertySource objects — one representing the set of JVM system properties (`System.getProperties()`) and one representing the set of system environment variables (`System.getenv()`).

These default property sources are present for `StandardEnvironment`, for use in standalone applications. `StandardServletEnvironment` is populated with additional default property sources including servlet config, servlet context parameters, and a `JndiPropertySource` if JNDI is available.

Concretely, when you use the `StandardEnvironment`, the call to `env.containsProperty("my-property")` returns true if a `my-property` system property or `my-property` environment variable is present at runtime.

The search performed is hierarchical. By default, system properties have precedence over environment variables. So, if the `my-property` property happens to be set in both places during a call to `env.getProperty("my-property")`, the system property value "wins" and is returned. Note that property values are not merged but rather completely overridden by a preceding entry.

For a common `StandardServletEnvironment`, the full hierarchy is as follows, with the highest-precedence entries at the top:

1. ServletConfig parameters (if applicable — for example, in case of a `DispatcherServlet` context)
2. ServletContext parameters (web.xml context-param entries)
3. JNDI environment variables (`java:comp/env/` entries)
4. JVM system properties (`-D` command-line arguments)
5. JVM system environment (operating system environment variables)

Most importantly, the entire mechanism is configurable. Perhaps you have a custom source of properties that you want to integrate into this search. To do so, implement and instantiate your own `PropertySource` and add it to the set of `PropertySources` for the current `Environment`. The following example shows how to do so:

Java

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

Kotlin

```
val ctx = GenericApplicationContext()
val sources = ctx.environment.propertySources
sources.addFirst(MyPropertySource())
```

In the preceding code, `MyPropertySource` has been added with highest precedence in the search. If it contains a `my-property` property, the property is detected and returned, in favor of any `my-property` property in any other `PropertySource`. The `MutablePropertySources` API exposes a number of methods that allow for precise manipulation of the set of property sources.

Using `@PropertySource`

The `@PropertySource` annotation provides a convenient and declarative mechanism for adding a `PropertySource` to Spring's `Environment`.

Given a file called `app.properties` that contains the key-value pair `testbean.name=myTestBean`, the following `@Configuration` class uses `@PropertySource` in such a way that a call to `testBean.getName()` returns `myTestBean`:

Java

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
public class AppConfig {

    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

Kotlin

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
class AppConfig {

    @Autowired
    private lateinit var env: Environment

    @Bean
    fun testBean() = TestBean().apply {
        name = env.getProperty("testbean.name")!!
    }
}
```

Any `${…}` placeholders present in a `@PropertySource` resource location are resolved against the set of property sources already registered against the environment, as the following example shows:

Java

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")
public class AppConfig {

    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

Kotlin

```
@Configuration
@PropertySource("classpath:/com/\${my.placeholder:default/path}/app.properties")
class AppConfig {

    @Autowired
    private lateinit var env: Environment

    @Bean
    fun testBean() = TestBean().apply {
        name = env.getProperty("testbean.name")!!
    }
}
```

Assuming that `my.placeholder` is present in one of the property sources already registered (for example, system properties or environment variables), the placeholder is resolved to the corresponding value. If not, then `default/path` is used as a default. If no default is specified and a property cannot be resolved, an `IllegalArgumentException` is thrown.

The `@PropertySource` annotation is repeatable, according to Java 8 conventions. However, all such `@PropertySource` annotations need to be declared at the same level, either directly on the configuration class or as meta-annotations within the same custom annotation. Mixing direct annotations and meta-annotations is not recommended, since direct annotations effectively override meta-annotations.

Placeholder Resolution in Statements

Historically, the value of placeholders in elements could be resolved only against JVM system properties or environment variables. This is no longer the case. Because the `Environment` abstraction is integrated throughout the container, it is easy to route resolution of placeholders through it. This means that you may configure the resolution process in any way you like. You can change the

precedence of searching through system properties and environment variables or remove them entirely. You can also add your own property sources to the mix, as appropriate.

Concretely, the following statement works regardless of where the `customer` property is defined, as long as it is available in the `Environment`:

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

2.1.14. Registering a `LoadTimeWeaver`

The `LoadTimeWeaver` is used by Spring to dynamically transform classes as they are loaded into the Java virtual machine (JVM).

To enable load-time weaving, you can add the `@EnableLoadTimeWeaving` to one of your `@Configuration` classes, as the following example shows:

Java

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {
}
```

Kotlin

```
@Configuration
@EnableLoadTimeWeaving
class AppConfig
```

Alternatively, for XML configuration, you can use the `context:load-time-weaver` element:

```
<beans>
    <context:load-time-weaver/>
</beans>
```

Once configured for the `ApplicationContext`, any bean within that `ApplicationContext` may implement `LoadTimeWeaverAware`, thereby receiving a reference to the load-time weaver instance. This is particularly useful in combination with Spring's JPA support where load-time weaving may be necessary for JPA class transformation. Consult the `LocalContainerEntityManagerFactoryBean` javadoc for more detail. For more on AspectJ load-time weaving, see Load-time Weaving with AspectJ in the Spring Framework.

2.1.15. Additional Capabilities of the `ApplicationContext`

As discussed in the chapter introduction, the `org.springframework.beans.factory` package provides basic functionality for managing and manipulating beans, including in a programmatic way. The `org.springframework.context` package adds the `ApplicationContext` interface, which extends the `BeanFactory` interface, in addition to extending other interfaces to provide additional functionality in a more application framework-oriented style. Many people use the `ApplicationContext` in a completely declarative fashion, not even creating it programmatically, but instead relying on support classes such as `ContextLoader` to automatically instantiate an `ApplicationContext` as part of the normal startup process of a Jakarta EE web application.

To enhance `BeanFactory` functionality in a more framework-oriented style, the context package also provides the following functionality:

- Access to messages in i18n-style, through the `MessageSource` interface.
- Access to resources, such as URLs and files, through the `ResourceLoader` interface.
- Event publication, namely to beans that implement the `ApplicationListener` interface, through the use of the `ApplicationEventPublisher` interface.
- Loading of multiple (hierarchical) contexts, letting each be focused on one particular layer, such as the web layer of an application, through the `HierarchicalBeanFactory` interface.

Internationalization using `MessageSource`

The `ApplicationContext` interface extends an interface called `MessageSource` and, therefore, provides internationalization ("i18n") functionality. Spring also provides the `HierarchicalMessageSource` interface, which can resolve messages hierarchically. Together, these interfaces provide the foundation upon which Spring effects message resolution. The methods defined on these interfaces include:

- `String getMessage(String code, Object[] args, String default, Locale loc)`: The basic method used to retrieve a message from the `MessageSource`. When no message is found for the specified locale, the default message is used. Any arguments passed in become replacement values, using the `MessageFormat` functionality provided by the standard library.
- `String getMessage(String code, Object[] args, Locale loc)`: Essentially the same as the previous method but with one difference: No default message can be specified. If the message cannot be found, a `NoSuchMessageException` is thrown.
- `String getMessage(MessageSourceResolvable resolvable, Locale locale)`: All properties used in the preceding methods are also wrapped in a class named `MessageSourceResolvable`, which you can use with this method.

When an `ApplicationContext` is loaded, it automatically searches for a `MessageSource` bean defined in the context. The bean must have the name `messageSource`. If such a bean is found, all calls to the preceding methods are delegated to the message source. If no message source is found, the `ApplicationContext` attempts to find a parent containing a bean with the same name. If it does, it uses that bean as the `MessageSource`. If the `ApplicationContext` cannot find any source for messages, an empty `DelegatingMessageSource` is instantiated in order to be able to accept calls to the methods defined above.

Spring provides three `MessageSource` implementations, `ResourceBundleMessageSource`, `ReloadableResourceBundleMessageSource` and `StaticMessageSource`. All of them implement `HierarchicalMessageSource` in order to do nested messaging. The `StaticMessageSource` is rarely used but provides programmatic ways to add messages to the source. The following example shows `ResourceBundleMessageSource`:

```
<beans>
    <bean id="messageSource"
            class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>format</value>
                <value>exceptions</value>
                <value>windows</value>
            </list>
        </property>
    </bean>
</beans>
```

The example assumes that you have three resource bundles called `format`, `exceptions` and `windows` defined in your classpath. Any request to resolve a message is handled in the JDK-standard way of resolving messages through `ResourceBundle` objects. For the purposes of the example, assume the contents of two of the above resource bundle files are as follows:

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The {0} argument is required.
```

The next example shows a program to run the `MessageSource` functionality. Remember that all `ApplicationContext` implementations are also `MessageSource` implementations and so can be cast to the `MessageSource` interface.

Java

```
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("message", null, "Default", Locale.ENGLISH);
    System.out.println(message);
}
```

Kotlin

```
fun main() {
    val resources = ClassPathXmlApplicationContext("beans.xml")
    val message = resources.getMessage("message", null, "Default", Locale.ENGLISH)
    println(message)
}
```

The resulting output from the above program is as follows:

```
Alligators rock!
```

To summarize, the `MessageSource` is defined in a file called `beans.xml`, which exists at the root of your classpath. The `messageSource` bean definition refers to a number of resource bundles through its `basenames` property. The three files that are passed in the list to the `basenames` property exist as files at the root of your classpath and are called `format.properties`, `exceptions.properties`, and `windows.properties`, respectively.

The next example shows arguments passed to the message lookup. These arguments are converted into `String` objects and inserted into placeholders in the lookup message.

```
<beans>

    <!-- this MessageSource is being used in a web application -->
    <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="exceptions"/>
    </bean>

    <!-- lets inject the above MessageSource into this POJO -->
    <bean id="example" class="com.something.Example">
        <property name="messages" ref="messageSource"/>
    </bean>

</beans>
```

Java

```
public class Example {

    private MessageSource messages;

    public void setMessages(MessageSource messages) {
        this.messages = messages;
    }

    public void execute() {
        String message = this.messages.getMessage("argument.required",
            new Object [] {"userDao"}, "Required", Locale.ENGLISH);
        System.out.println(message);
    }
}
```

Kotlin

```
    class Example {

    lateinit var messages: MessageSource

    fun execute() {
        val message = messages.getMessage("argument.required",
                arrayOf("userDao"), "Required", Locale.ENGLISH)
        println(message)
    }
}
```

The resulting output from the invocation of the `execute()` method is as follows:

```
The userDao argument is required.
```

With regard to internationalization ("i18n"), Spring's various `MessageSource` implementations follow the same locale resolution and fallback rules as the standard JDK `ResourceBundle`. In short, and continuing with the example `messageSource` defined previously, if you want to resolve messages against the British (`en-GB`) locale, you would create files called `format_en_GB.properties`, `exceptions_en_GB.properties`, and `windows_en_GB.properties`, respectively.

Typically, locale resolution is managed by the surrounding environment of the application. In the following example, the locale against which (British) messages are resolved is specified manually:

```
# in exceptions_en_GB.properties
argument.required=Ebagum lad, the ''{0}'' argument is required, I say, required.
```

Java

```
public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", Locale.UK);
    System.out.println(message);
}
```

Kotlin

```
fun main() {
    val resources = ClassPathXmlApplicationContext("beans.xml")
    val message = resources.getMessage("argument.required",
            arrayOf("userDao"), "Required", Locale.UK)
    println(message)
}
```

The resulting output from the running of the above program is as follows:

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

You can also use the `MessageSourceAware` interface to acquire a reference to any `MessageSource` that has been defined. Any bean that is defined in an `ApplicationContext` that implements the `MessageSourceAware` interface is injected with the application context's `MessageSource` when the bean is created and configured.

Because Spring's `MessageSource` is based on Java's `ResourceBundle`, it does not merge bundles with the same base name, but will only use the first bundle found. Subsequent message bundles with the same base name are ignored.

As an alternative to `ResourceBundleMessageSource`, Spring provides a `ReloadableResourceBundleMessageSource` class. This variant supports the same bundle file format but is more flexible than the standard JDK based `ResourceBundleMessageSource` implementation. In particular, it allows for reading files from any Spring resource location (not only from the classpath) and supports hot reloading of bundle property files (while efficiently caching them in between). See the `ReloadableResourceBundleMessageSource` javadoc for details.

Standard and Custom Events

Event handling in the `ApplicationContext` is provided through the `ApplicationEvent` class and the `ApplicationListener` interface. If a bean that implements the `ApplicationListener` interface is deployed into the context, every time an `ApplicationEvent` gets published to the `ApplicationContext`, that bean is notified. Essentially, this is the standard Observer design pattern.

As of Spring 4.2, the event infrastructure has been significantly improved and offers an annotation-based model as well as the ability to publish any arbitrary event (that is, an object that does not necessarily extend from `ApplicationEvent`). When such an object is published, we wrap it in an event for you.

The following table describes the standard events that Spring provides:

Table 7. Built-in Events

Event	Explanation
`ContextRefreshedEvent`	Published when the `ApplicationContext` is initialized or refreshed (for example, by using the `refresh()` method on the `ConfigurableApplicationContext` interface). Here, “initialized” means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the `ApplicationContext` object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen `ApplicationContext` actually supports such “hot” refreshes. For example, `XmlWebApplicationContext` supports hot refreshes, but `GenericApplicationContext` does not.
`ContextStartedEvent`	Published when the `ApplicationContext` is started by using the `start()` method on the `ConfigurableApplicationContext` interface. Here, “started” means that all `Lifecycle` beans receive an explicit start signal. Typically, this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart (for example, components that have not already started on initialization).
`ContextStoppedEvent`	Published when the `ApplicationContext` is stopped by using the `stop()` method on the `ConfigurableApplicationContext` interface. Here, “stopped” means that all `Lifecycle` beans receive an explicit stop signal. A stopped context may be restarted through a `start()` call.
`ContextClosedEvent`	Published when the `ApplicationContext` is being closed by using the `close()` method on the `ConfigurableApplicationContext` interface or via a JVM shutdown hook. Here, "closed" means that all singleton beans will be destroyed. Once the context is closed, it reaches its end of life and cannot be refreshed or restarted.
`RequestHandledEvent`	A web-specific event telling all beans that an HTTP request has been serviced. This event is published after the request is complete. This event is only applicable to web applications that use Spring’s `DispatcherServlet`.
`ServletRequestHandledEvent`	A subclass of `RequestHandledEvent` that adds Servlet-specific context information.

You can also create and publish your own custom events. The following example shows a simple class that extends Spring’s `ApplicationEvent` base class:

Java

```
public class BlockedListEvent extends ApplicationEvent {

    private final String address;
    private final String content;

    public BlockedListEvent(Object source, String address, String content) {
        super(source);
        this.address = address;
        this.content = content;
    }

    // accessor and other methods...
}
```

Kotlin

```
class BlockedListEvent(source: Any,
                       val address: String,
                       val content: String) : ApplicationEvent(source)
```

To publish a custom `ApplicationEvent`, call the `publishEvent()` method on an `ApplicationEventPublisher`. Typically, this is done by creating a class that implements `ApplicationEventPublisherAware` and registering it as a Spring bean. The following example shows such a class:

Java

```
public class EmailService implements ApplicationEventPublisherAware {

    private List<String> blockedList;
    private ApplicationEventPublisher publisher;

    public void setBlockedList(List<String> blockedList) {
        this.blockedList = blockedList;
    }

    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
        this.publisher = publisher;
    }

    public void sendEmail(String address, String content) {
        if (blockedList.contains(address)) {
            publisher.publishEvent(new BlockedListEvent(this, address, content));
            return;
        }
        // send email...
    }
}
```

Kotlin

```
class EmailService : ApplicationEventPublisherAware {

    private lateinit var blockedList: List<String>
    private lateinit var publisher: ApplicationEventPublisher

    fun setBlockedList(blockedList: List<String>) {
        this.blockedList = blockedList
    }

    override fun setApplicationEventPublisher(publisher: ApplicationEventPublisher) {
        this.publisher = publisher
    }

    fun sendEmail(address: String, content: String) {
        if (blockedList!!.contains(address)) {
            publisher!!.publishEvent(BlockedListEvent(this, address, content))
            return
        }
        // send email...
    }
}
```

At configuration time, the Spring container detects that `EmailService` implements `ApplicationEventPublisherAware` and automatically calls `setApplicationEventPublisher()`. In reality,

the parameter passed in is the Spring container itself. You are interacting with the application context through its `ApplicationEventPublisher` interface.

To receive the custom `ApplicationEvent`, you can create a class that implements `ApplicationListener` and register it as a Spring bean. The following example shows such a class:

Java

```
public class BlockedListNotifier implements ApplicationListener<BlockedListEvent> {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(BlockedListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}
```

Kotlin

```
class BlockedListNotifier : ApplicationListener<BlockedListEvent> {

    lateinit var notificationAddress: String

    override fun onApplicationEvent(event: BlockedListEvent) {
        // notify appropriate parties via notificationAddress...
    }
}
```

Notice that `ApplicationListener` is generically parameterized with the type of your custom event (`BlockedListEvent` in the preceding example). This means that the `onApplicationEvent()` method can remain type-safe, avoiding any need for downcasting. You can register as many event listeners as you wish, but note that, by default, event listeners receive events synchronously. This means that the `publishEvent()` method blocks until all listeners have finished processing the event. One advantage of this synchronous and single-threaded approach is that, when a listener receives an event, it operates inside the transaction context of the publisher if a transaction context is available. If another strategy for event publication becomes necessary, see the javadoc for Spring's `ApplicationEventMulticaster` interface and `SimpleApplicationEventMulticaster` implementation for configuration options.

The following example shows the bean definitions used to register and configure each of the classes above:

```
<bean id="emailService" class="example.EmailService">
    <property name="blockedList">
        <list>
            <value>known.spammer@example.org</value>
            <value>known.hacker@example.org</value>
            <value>john.doe@example.org</value>
        </list>
    </property>
</bean>

<bean id="blockedListNotifier" class="example.BlockedListNotifier">
    <property name="notificationAddress" value="blockedlist@example.org"/>
</bean>
```

Putting it all together, when the `sendEmail()` method of the `emailService` bean is called, if there are any email messages that should be blocked, a custom event of type `BlockedListEvent` is published. The `blockedListNotifier` bean is registered as an `ApplicationListener` and receives the `BlockedListEvent`, at which point it can notify appropriate parties.

Spring's eventing mechanism is designed for simple communication between Spring beans within the same application context. However, for more sophisticated enterprise integration needs, the separately maintained Spring Integration project provides complete support for building lightweight, pattern-oriented, event-driven architectures that build upon the well-known Spring programming model.

Annotation-based Event Listeners

You can register an event listener on any method of a managed bean by using the `@EventListener` annotation. The `BlockedListNotifier` can be rewritten as follows:

Java

```
public class BlockedListNotifier {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    @EventListener
    public void processBlockedListEvent(BlockedListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}
```

Kotlin

```
class BlockedListNotifier {

    lateinit var notificationAddress: String

    @EventListener
    fun processBlockedListEvent(event: BlockedListEvent) {
        // notify appropriate parties via notificationAddress...
    }
}
```

The method signature once again declares the event type to which it listens, but, this time, with a flexible name and without implementing a specific listener interface. The event type can also be narrowed through generics as long as the actual event type resolves your generic parameter in its implementation hierarchy.

If your method should listen to several events or if you want to define it with no parameter at all, the event types can also be specified on the annotation itself. The following example shows how to do so:

Java

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})
public void handleContextStart() {
    // ...
}
```

Kotlin

```
@EventListener(ContextStartedEvent::class, ContextRefreshedEvent::class)
fun handleContextStart() {
    // ...
}
```

It is also possible to add additional runtime filtering by using the `condition` attribute of the annotation that defines a `SpEL` expression, which should match to actually invoke the method for a particular event.

The following example shows how our notifier can be rewritten to be invoked only if the `content` attribute of the event is equal to `my-event`:

Java

```
@EventListener(condition = "#blEvent.content == 'my-event'")
public void processBlockedListEvent(BlockedListEvent blEvent) {
    // notify appropriate parties via notificationAddress...
}
```

Kotlin

```
@EventListener(condition = "#blEvent.content == 'my-event'")
fun processBlockedListEvent(blEvent: BlockedListEvent) {
    // notify appropriate parties via notificationAddress...
}
```

Each SpEL expression evaluates against a dedicated context. The following table lists the items made available to the context so that you can use them for conditional event processing:

Table 8. Event SpEL available metadata

Name	Location	Description	Example
Event	root object	The actual `ApplicationEvent`.	`#root.event` or `event`
Arguments array	root object	The arguments (as an object array) used to invoke the method.	`#root.args` or `args`; `args[0]` to access the first argument, etc.
Argument name	evaluation context	The name of any of the method arguments. If, for some reason, the names are not available (for example, because there is no debug information in the compiled byte code), individual arguments are also available using the `#a<#arg>` syntax where `<#arg>` stands for the argument index (starting from 0).	`#blEvent` or `#a0` (you can also use `#p0` or `#p<#arg>` parameter notation as an alias)

Note that `#root.event` gives you access to the underlying event, even if your method signature actually refers to an arbitrary object that was published.

If you need to publish an event as the result of processing another event, you can change the method signature to return the event that should be published, as the following example shows:

Java

```
@EventListener
public ListUpdateEvent handleBlockedListEvent(BlockedListEvent event) {
    // notify appropriate parties via notificationAddress and
    // then publish a ListUpdateEvent...
}
```

Kotlin

```
@EventListener
fun handleBlockedListEvent(event: BlockedListEvent): ListUpdateEvent {
    // notify appropriate parties via notificationAddress and
    // then publish a ListUpdateEvent...
}
```

This feature is not supported for asynchronous listeners.

The `handleBlockedListEvent()` method publishes a new `ListUpdateEvent` for every `BlockedListEvent` that it handles. If you need to publish several events, you can return a `Collection` or an array of events instead.

Asynchronous Listeners

If you want a particular listener to process events asynchronously, you can reuse the regular `@Async` support. The following example shows how to do so:

Java

```
@EventListener
@Async
public void processBlockedListEvent(BlockedListEvent event) {
    // BlockedListEvent is processed in a separate thread
}
```

Kotlin

```
@EventListener
@Async
fun processBlockedListEvent(event: BlockedListEvent) {
    // BlockedListEvent is processed in a separate thread
}
```

Be aware of the following limitations when using asynchronous events:

- If an asynchronous event listener throws an `Exception`, it is not propagated to the caller. See `AsyncUncaughtExceptionHandler` for more details.
- Asynchronous event listener methods cannot publish a subsequent event by returning a value. If you need to publish another event as the result of the processing, inject an `ApplicationEventPublisher` to publish the event manually.

Ordering Listeners

If you need one listener to be invoked before another one, you can add the `@Order` annotation to the method declaration, as the following example shows:

Java

```
@EventListener
@Order(42)
public void processBlockedListEvent(BlockedListEvent event) {
    // notify appropriate parties via notificationAddress...
}
```

Kotlin

```
@EventListener
@Order(42)
fun processBlockedListEvent(event: BlockedListEvent) {
    // notify appropriate parties via notificationAddress...
}
```

Generic Events

You can also use generics to further define the structure of your event. Consider using an `EntityCreatedEvent<T>` where `T` is the type of the actual entity that got created. For example, you can create the following listener definition to receive only `EntityCreatedEvent` for a `Person`:

Java

```
@EventListener
public void onPersonCreated(EntityCreatedEvent<Person> event) {
    // ...
}
```

Kotlin

```
@EventListener
fun onPersonCreated(event: EntityCreatedEvent<Person>) {
    // ...
}
```

Due to type erasure, this works only if the event that is fired resolves the generic parameters on which the event listener filters (that is, something like `class PersonCreatedEvent extends EntityCreatedEvent<Person> { … }`).

In certain circumstances, this may become quite tedious if all events follow the same structure (as should be the case for the event in the preceding example). In such a case, you can implement `ResolvableTypeProvider` to guide the framework beyond what the runtime environment provides. The following event shows how to do so:

Java

```
public class EntityCreatedEvent<T> extends ApplicationEvent implements
ResolvableTypeProvider {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(),
ResolvableType.forInstance(getSource()));
    }
}
```

Kotlin

```
class EntityCreatedEvent<T>(entity: T) : ApplicationEvent(entity),
ResolvableTypeProvider {

    override fun getResolvableType(): ResolvableType? {
        return ResolvableType.forClassWithGenerics(javaClass,
ResolvableType.forInstance(getSource()))
    }
}
```

This works not only for `ApplicationEvent` but any arbitrary object that you send as an event.

Convenient Access to Low-level Resources

For optimal usage and understanding of application contexts, you should familiarize yourself with Spring's `Resource` abstraction, as described in Resources.

An application context is a `ResourceLoader`, which can be used to load `Resource` objects. A `Resource` is essentially a more feature rich version of the JDK `java.net.URL` class. In fact, the implementations of the `Resource` wrap an instance of `java.net.URL`, where appropriate. A `Resource` can obtain low-level resources from almost any location in a transparent fashion, including from the classpath, a filesystem location, anywhere describable with a standard URL, and some other variations. If the resource location string is a simple path without any special prefixes, where those resources come from is specific and appropriate to the actual application context type.

You can configure a bean deployed into the application context to implement the special callback interface, `ResourceLoaderAware`, to be automatically called back at initialization time with the application context itself passed in as the `ResourceLoader`. You can also expose properties of type `Resource`, to be used to access static resources. They are injected into it like any other properties. You can specify those `Resource` properties as simple `String` paths and rely on automatic conversion from those text strings to actual `Resource` objects when the bean is deployed.

The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings and, in simple form, are treated appropriately according to the specific context implementation. For example `ClassPathXmlApplicationContext` treats a simple location path as a classpath location. You can also use location paths (resource strings) with special prefixes to force loading of definitions from the classpath or a URL, regardless of the actual context type.

Application Startup Tracking

The `ApplicationContext` manages the lifecycle of Spring applications and provides a rich programming model around components. As a result, complex applications can have equally complex component graphs and startup phases.

Tracking the application startup steps with specific metrics can help understand where time is being spent during the startup phase, but it can also be used as a way to better understand the context lifecycle as a whole.

The `AbstractApplicationContext` (and its subclasses) is instrumented with an `ApplicationStartup`, which collects `StartupStep` data about various startup phases:

- application context lifecycle (base packages scanning, config classes management)
- beans lifecycle (instantiation, smart initialization, post processing)
- application events processing

Here is an example of instrumentation in the `AnnotationConfigApplicationContext`:

Java

```
// create a startup step and start recording
StartupStep scanPackages = this.getApplicationStartup().start("spring.context.base-
packages.scan");
// add tagging information to the current step
scanPackages.tag("packages", () -> Arrays.toString(basePackages));
// perform the actual phase we're instrumenting
this.scanner.scan(basePackages);
// end the current step
scanPackages.end();
```

Kotlin

```
// create a startup step and start recording
val scanPackages = this.getApplicationStartup().start("spring.context.base-
packages.scan")
// add tagging information to the current step
scanPackages.tag("packages", () -> Arrays.toString(basePackages))
// perform the actual phase we're instrumenting
this.scanner.scan(basePackages)
// end the current step
scanPackages.end()
```

The application context is already instrumented with multiple steps. Once recorded, these startup steps can be collected, displayed and analyzed with specific tools. For a complete list of existing startup steps, you can check out the dedicated appendix section.

The default `ApplicationStartup` implementation is a no-op variant, for minimal overhead. This means no metrics will be collected during application startup by default. Spring Framework ships with an implementation for tracking startup steps with Java Flight Recorder: `FlightRecorderApplicationStartup`. To use this variant, you must configure an instance of it to the `ApplicationContext` as soon as it's been created.

Developers can also use the `ApplicationStartup` infrastructure if they're providing their own `AbstractApplicationContext` subclass, or if they wish to collect more precise data.

`ApplicationStartup` is meant to be only used during application startup and for the core container; this is by no means a replacement for Java profilers or metrics libraries like Micrometer.

To start collecting custom `StartupStep`, components can either get the `ApplicationStartup` instance from the application context directly, make their component implement `ApplicationStartupAware`, or ask for the `ApplicationStartup` type on any injection point.

Developers should not use the `"spring.*"` namespace when creating custom startup steps. This namespace is reserved for internal Spring usage and is subject to change.

Convenient ApplicationContext Instantiation for Web Applications

You can create `ApplicationContext` instances declaratively by using, for example, a `ContextLoader`. Of course, you can also create `ApplicationContext` instances programmatically by using one of the `ApplicationContext` implementations.

You can register an `ApplicationContext` by using the `ContextLoaderListener`, as the following example shows:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

The listener inspects the `contextConfigLocation` parameter. If the parameter does not exist, the listener uses `/WEB-INF/applicationContext.xml` as a default. When the parameter does exist, the listener separates the `String` by using predefined delimiters (comma, semicolon, and whitespace) and uses the values as locations where application contexts are searched. Ant-style path patterns

are supported as well. Examples are `/WEB-INF/*Context.xml` (for all files with names that end with `Context.xml` and that reside in the `WEB-INF` directory) and `/WEB-INF/**/*Context.xml` (for all such files in any subdirectory of `WEB-INF`).

Deploying a Spring `ApplicationContext` as a Jakarta EE RAR File

It is possible to deploy a Spring `ApplicationContext` as a RAR file, encapsulating the context and all of its required bean classes and library JARs in a Jakarta EE RAR deployment unit. This is the equivalent of bootstrapping a stand-alone `ApplicationContext` (only hosted in Jakarta EE environment) being able to access the Jakarta EE servers facilities. RAR deployment is a more natural alternative to a scenario of deploying a headless WAR file — in effect, a WAR file without any HTTP entry points that is used only for bootstrapping a Spring `ApplicationContext` in a Jakarta EE environment.

RAR deployment is ideal for application contexts that do not need HTTP entry points but rather consist only of message endpoints and scheduled jobs. Beans in such a context can use application server resources such as the JTA transaction manager and JNDI-bound JDBC `DataSource` instances and JMS `ConnectionFactory` instances and can also register with the platform's JMX server — all through Spring's standard transaction management and JNDI and JMX support facilities. Application components can also interact with the application server's JCA `WorkManager` through Spring's `TaskExecutor` abstraction.

See the javadoc of the `SpringContextResourceAdapter` class for the configuration details involved in RAR deployment.

For a simple deployment of a Spring ApplicationContext as a Jakarta EE RAR file:

1. Package all application classes into a RAR file (which is a standard JAR file with a different file extension).
2. Add all required library JARs into the root of the RAR archive.
3. Add a `META-INF/ra.xml` deployment descriptor (as shown in the javadoc for `SpringContextResourceAdapter`) and the corresponding Spring XML bean definition file(s) (typically `META-INF/applicationContext.xml`).
4. Drop the resulting RAR file into your application server's deployment directory.

Such RAR deployment units are usually self-contained. They do not expose components to the outside world, not even to other modules of the same application. Interaction with a RAR-based `ApplicationContext` usually occurs through JMS destinations that it shares with other modules. A RAR-based `ApplicationContext` may also, for example, schedule some jobs or react to new files in the file system (or the like). If it needs to allow synchronous access from the outside, it could (for example) export RMI endpoints, which may be used by other application modules on the same machine.

2.1.16. The `BeanFactory` API

The `BeanFactory` API provides the underlying basis for Spring's IoC functionality. Its specific contracts are mostly used in integration with other parts of Spring and related third-party

frameworks, and its `DefaultListableBeanFactory` implementation is a key delegate within the higher-level `GenericApplicationContext` container.

`BeanFactory` and related interfaces (such as `BeanFactoryAware`, `InitializingBean`, `DisposableBean`) are important integration points for other framework components. By not requiring any annotations or even reflection, they allow for very efficient interaction between the container and its components. Application-level beans may use the same callback interfaces but typically prefer declarative dependency injection instead, either through annotations or through programmatic configuration.

Note that the core `BeanFactory` API level and its `DefaultListableBeanFactory` implementation do not make assumptions about the configuration format or any component annotations to be used. All of these flavors come in through extensions (such as `XmlBeanDefinitionReader` and `AutowiredAnnotationBeanPostProcessor`) and operate on shared `BeanDefinition` objects as a core metadata representation. This is the essence of what makes Spring's container so flexible and extensible.

`BeanFactory` or `ApplicationContext`?

This section explains the differences between the `BeanFactory` and `ApplicationContext` container levels and the implications on bootstrapping.

You should use an `ApplicationContext` unless you have a good reason for not doing so, with `GenericApplicationContext` and its subclass `AnnotationConfigApplicationContext` as the common implementations for custom bootstrapping. These are the primary entry points to Spring's core container for all common purposes: loading of configuration files, triggering a classpath scan, programmatically registering bean definitions and annotated classes, and (as of 5.0) registering functional bean definitions.

Because an `ApplicationContext` includes all the functionality of a `BeanFactory`, it is generally recommended over a plain `BeanFactory`, except for scenarios where full control over bean processing is needed. Within an `ApplicationContext` (such as the `GenericApplicationContext` implementation), several kinds of beans are detected by convention (that is, by bean name or by bean type — in particular, post-processors), while a plain `DefaultListableBeanFactory` is agnostic about any special beans.

For many extended container features, such as annotation processing and AOP proxying, the `BeanPostProcessor` extension point is essential. If you use only a plain `DefaultListableBeanFactory`, such post-processors do not get detected and activated by default. This situation could be confusing, because nothing is actually wrong with your bean configuration. Rather, in such a scenario, the container needs to be fully bootstrapped through additional setup.

The following table lists features provided by the `BeanFactory` and `ApplicationContext` interfaces and implementations.

Table 9. Feature Matrix

Feature	`BeanFactory`	`ApplicationContext`
Bean instantiation/wiring	Yes	Yes
Integrated lifecycle management	No	Yes

Feature	BeanFactory	ApplicationContext
Automatic BeanPostProcessor registration	No	Yes
Automatic BeanFactoryPostProcessor registration	No	Yes
Convenient MessageSource access (for internationalization)	No	Yes
Built-in ApplicationEvent publication mechanism	No	Yes

To explicitly register a bean post-processor with a DefaultListableBeanFactory, you need to programmatically call addBeanPostProcessor, as the following example shows:

Java

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(new AutowiredAnnotationBeanPostProcessor());
factory.addBeanPostProcessor(new MyBeanPostProcessor());

// now start using the factory
```

Kotlin

```
val factory = DefaultListableBeanFactory()
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(AutowiredAnnotationBeanPostProcessor())
factory.addBeanPostProcessor(MyBeanPostProcessor())

// now start using the factory
```

To apply a BeanFactoryPostProcessor to a plain DefaultListableBeanFactory, you need to call its postProcessBeanFactory method, as the following example shows:

Java

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertySourcesPlaceholderConfigurer cfg = new PropertySourcesPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

Kotlin

```
val factory = DefaultListableBeanFactory()
val reader = XmlBeanDefinitionReader(factory)
reader.loadBeanDefinitions(FileSystemResource("beans.xml"))

// bring in some property values from a Properties file
val cfg = PropertySourcesPlaceholderConfigurer()
cfg.setLocation(FileSystemResource("jdbc.properties"))

// now actually do the replacement
cfg.postProcessBeanFactory(factory)
```

In both cases, the explicit registration steps are inconvenient, which is why the various `ApplicationContext` variants are preferred over a plain `DefaultListableBeanFactory` in Spring-backed applications, especially when relying on `BeanFactoryPostProcessor` and `BeanPostProcessor` instances for extended container functionality in a typical enterprise setup.

An `AnnotationConfigApplicationContext` has all common annotation post-processors registered and may bring in additional processors underneath the covers through configuration annotations, such as `@EnableTransactionManagement`. At the abstraction level of Spring's annotation-based configuration model, the notion of bean post-processors becomes a mere internal container detail.

2.2. Resources

This chapter covers how Spring handles resources and how you can work with resources in Spring. It includes the following topics:

- Introduction
- The `Resource` Interface
- Built-in `Resource` Implementations
- The `ResourceLoader` Interface

- The `ResourcePatternResolver` Interface
- The `ResourceLoaderAware` Interface
- Resources as Dependencies
- Application Contexts and Resource Paths

2.2.1. Introduction

Java's standard `java.net.URL` class and standard handlers for various URL prefixes, unfortunately, are not quite adequate enough for all access to low-level resources. For example, there is no standardized `URL` implementation that may be used to access a resource that needs to be obtained from the classpath or relative to a `ServletContext`. While it is possible to register new handlers for specialized `URL` prefixes (similar to existing handlers for prefixes such as `http:`), this is generally quite complicated, and the `URL` interface still lacks some desirable functionality, such as a method to check for the existence of the resource being pointed to.

2.2.2. The `Resource` Interface

Spring's `Resource` interface located in the `org.springframework.core.io.` package is meant to be a more capable interface for abstracting access to low-level resources. The following listing provides an overview of the `Resource` interface. See the `Resource` javadoc for further details.

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isReadable();

    boolean isOpen();

    boolean isFile();

    URL getURL() throws IOException;

    URI getURI() throws IOException;

    File getFile() throws IOException;

    ReadableByteChannel readableChannel() throws IOException;

    long contentLength() throws IOException;

    long lastModified() throws IOException;

    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();
}
```

As the definition of the `Resource` interface shows, it extends the `InputStreamSource` interface. The following listing shows the definition of the `InputStreamSource` interface:

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;
}
```

Some of the most important methods from the `Resource` interface are:

- `getInputStream()`: Locates and opens the resource, returning an `InputStream` for reading from the resource. It is expected that each invocation returns a fresh `InputStream`. It is the responsibility of the caller to close the stream.
- `exists()`: Returns a `boolean` indicating whether this resource actually exists in physical form.
- `isOpen()`: Returns a `boolean` indicating whether this resource represents a handle with an open stream. If `true`, the `InputStream` cannot be read multiple times and must be read once only and then closed to avoid resource leaks. Returns `false` for all usual resource implementations, with the exception of `InputStreamResource`.

- `getDescription()`: Returns a description for this resource, to be used for error output when working with the resource. This is often the fully qualified file name or the actual URL of the resource.

Other methods let you obtain an actual `URL` or `File` object representing the resource (if the underlying implementation is compatible and supports that functionality).

Some implementations of the `Resource` interface also implement the extended `WritableResource` interface for a resource that supports writing to it.

Spring itself uses the `Resource` abstraction extensively, as an argument type in many method signatures when a resource is needed. Other methods in some Spring APIs (such as the constructors to various `ApplicationContext` implementations) take a `String` which in unadorned or simple form is used to create a `Resource` appropriate to that context implementation or, via special prefixes on the `String` path, let the caller specify that a specific `Resource` implementation must be created and used.

While the `Resource` interface is used a lot with Spring and by Spring, it is actually very convenient to use as a general utility class by itself in your own code, for access to resources, even when your code does not know or care about any other parts of Spring. While this couples your code to Spring, it really only couples it to this small set of utility classes, which serves as a more capable replacement for `URL` and can be considered equivalent to any other library you would use for this purpose.

The `Resource` abstraction does not replace functionality. It wraps it where possible. For example, a `UrlResource` wraps a URL and uses the wrapped `URL` to do its work.

2.2.3. Built-in `Resource` Implementations

Spring includes several built-in `Resource` implementations:

- `UrlResource`
- `ClassPathResource`
- `FileSystemResource`
- `PathResource`
- `ServletContextResource`
- `InputStreamResource`
- `ByteArrayResource`

For a complete list of `Resource` implementations available in Spring, consult the "All Known Implementing Classes" section of the `Resource` javadoc.

`UrlResource`

`UrlResource` wraps a `java.net.URL` and can be used to access any object that is normally accessible with a URL, such as files, an HTTPS target, an FTP target, and others. All URLs have a standardized `String` representation, such that appropriate standardized prefixes are used to indicate one URL type from another. This includes `file:` for accessing filesystem paths, `https:` for accessing resources

through the HTTPS protocol, `ftp:` for accessing resources through FTP, and others.

A `UrlResource` is created by Java code by explicitly using the `UrlResource` constructor but is often created implicitly when you call an API method that takes a `String` argument meant to represent a path. For the latter case, a JavaBeans `PropertyEditor` ultimately decides which type of `Resource` to create. If the path string contains a well-known (to property editor, that is) prefix (such as `classpath:`), it creates an appropriate specialized `Resource` for that prefix. However, if it does not recognize the prefix, it assumes the string is a standard URL string and creates a `UrlResource`.

`ClassPathResource`

This class represents a resource that should be obtained from the classpath. It uses either the thread context class loader, a given class loader, or a given class for loading resources.

This `Resource` implementation supports resolution as a `java.io.File` if the class path resource resides in the file system but not for classpath resources that reside in a jar and have not been expanded (by the servlet engine or whatever the environment is) to the filesystem. To address this, the various `Resource` implementations always support resolution as a `java.net.URL`.

A `ClassPathResource` is created by Java code by explicitly using the `ClassPathResource` constructor but is often created implicitly when you call an API method that takes a `String` argument meant to represent a path. For the latter case, a JavaBeans `PropertyEditor` recognizes the special prefix, `classpath:`, on the string path and creates a `ClassPathResource` in that case.

`FileSystemResource`

This is a `Resource` implementation for `java.io.File` handles. It also supports `java.nio.file.Path` handles, applying Spring's standard String-based path transformations but performing all operations via the `java.nio.file.Files` API. For pure `java.nio.path.Path` based support use a `PathResource` instead. `FileSystemResource` supports resolution as a `File` and as a `URL`.

`PathResource`

This is a `Resource` implementation for `java.nio.file.Path` handles, performing all operations and transformations via the `Path` API. It supports resolution as a `File` and as a `URL` and also implements the extended `WritableResource` interface. `PathResource` is effectively a pure `java.nio.path.Path` based alternative to `FileSystemResource` with different `createRelative` behavior.

`ServletContextResource`

This is a `Resource` implementation for `ServletContext` resources that interprets relative paths within the relevant web application's root directory.

It always supports stream access and URL access but allows `java.io.File` access only when the web application archive is expanded and the resource is physically on the filesystem. Whether or not it is expanded and on the filesystem or accessed directly from the JAR or somewhere else like a database (which is conceivable) is actually dependent on the Servlet container.

`InputStreamResource`

An `InputStreamResource` is a `Resource` implementation for a given `InputStream`. It should be used only

if no specific `Resource` implementation is applicable. In particular, prefer `ByteArrayResource` or any of the file-based `Resource` implementations where possible.

In contrast to other `Resource` implementations, this is a descriptor for an already-opened resource. Therefore, it returns `true` from `isOpen()`. Do not use it if you need to keep the resource descriptor somewhere or if you need to read a stream multiple times.

`ByteArrayResource`

This is a `Resource` implementation for a given byte array. It creates a `ByteArrayInputStream` for the given byte array.

It is useful for loading content from any given byte array without having to resort to a single-use `InputStreamResource`.

2.2.4. The `ResourceLoader` Interface

The `ResourceLoader` interface is meant to be implemented by objects that can return (that is, load) `Resource` instances. The following listing shows the `ResourceLoader` interface definition:

```
public interface ResourceLoader {

    Resource getResource(String location);

    ClassLoader getClassLoader();
}
```

All application contexts implement the `ResourceLoader` interface. Therefore, all application contexts may be used to obtain `Resource` instances.

When you call `getResource()` on a specific application context, and the location path specified doesn't have a specific prefix, you get back a `Resource` type that is appropriate to that particular application context. For example, assume the following snippet of code was run against a `ClassPathXmlApplicationContext` instance:

Java

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("some/resource/path/myTemplate.txt")
```

Against a `ClassPathXmlApplicationContext`, that code returns a `ClassPathResource`. If the same method were run against a `FileSystemXmlApplicationContext` instance, it would return a `FileSystemResource`. For a `WebApplicationContext`, it would return a `ServletContextResource`. It would similarly return appropriate objects for each context.

As a result, you can load resources in a fashion appropriate to the particular application context.

On the other hand, you may also force `ClassPathResource` to be used, regardless of the application context type, by specifying the special `classpath:` prefix, as the following example shows:

Java

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("classpath:some/resource/path/myTemplate.txt")
```

Similarly, you can force a `UrlResource` to be used by specifying any of the standard `java.net.URL` prefixes. The following examples use the `file` and `https` prefixes:

Java

```
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("file:///some/resource/path/myTemplate.txt")
```

Java

```
Resource template =
ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt")
```

The following table summarizes the strategy for converting `String` objects to `Resource` objects:

Table 10. Resource strings

Prefix	Example	Explanation
classpath:	`classpath:com/myapp/config.xml`	Loaded from the classpath.
file:	`file:///data/config.xml`	Loaded as a `URL` from the filesystem. See also `FileSystemResource` Caveats.
https:	`https://myserver/logo.png`	Loaded as a `URL`.
(none)	`/data/config.xml`	Depends on the underlying `ApplicationContext`.

2.2.5. The `ResourcePatternResolver` Interface

The `ResourcePatternResolver` interface is an extension to the `ResourceLoader` interface which defines a strategy for resolving a location pattern (for example, an Ant-style path pattern) into `Resource` objects.

```
public interface ResourcePatternResolver extends ResourceLoader {

    String CLASSPATH_ALL_URL_PREFIX = "classpath*:";

    Resource[] getResources(String locationPattern) throws IOException;
}
```

As can be seen above, this interface also defines a special `classpath*:` resource prefix for all matching resources from the class path. Note that the resource location is expected to be a path without placeholders in this case — for example, `classpath*:/config/beans.xml`. JAR files or different directories in the class path can contain multiple files with the same path and the same name. See Wildcards in Application Context Constructor Resource Paths and its subsections for further details on wildcard support with the `classpath*:` resource prefix.

A passed-in `ResourceLoader` (for example, one supplied via `ResourceLoaderAware` semantics) can be checked whether it implements this extended interface too.

`PathMatchingResourcePatternResolver` is a standalone implementation that is usable outside an `ApplicationContext` and is also used by `ResourceArrayPropertyEditor` for populating `Resource[]` bean properties. `PathMatchingResourcePatternResolver` is able to resolve a specified resource location path into one or more matching `Resource` objects. The source path may be a simple path which has a one-to-one mapping to a target `Resource`, or alternatively may contain the special `classpath*:` prefix and/or internal Ant-style regular expressions (matched using Spring's `org.springframework.util.AntPathMatcher` utility). Both of the latter are effectively wildcards.

The default `ResourceLoader` in any standard `ApplicationContext` is in fact an instance of `PathMatchingResourcePatternResolver` which implements the `ResourcePatternResolver` interface. The same is true for the `ApplicationContext` instance itself which also implements the `ResourcePatternResolver` interface and delegates to the default `PathMatchingResourcePatternResolver`.

2.2.6. The `ResourceLoaderAware` Interface

The `ResourceLoaderAware` interface is a special callback interface which identifies components that expect to be provided a `ResourceLoader` reference. The following listing shows the definition of the `ResourceLoaderAware` interface:

```
public interface ResourceLoaderAware {

    void setResourceLoader(ResourceLoader resourceLoader);
}
```

When a class implements `ResourceLoaderAware` and is deployed into an application context (as a Spring-managed bean), it is recognized as `ResourceLoaderAware` by the application context. The application context then invokes `setResourceLoader(ResourceLoader)`, supplying itself as the argument (remember, all application contexts in Spring implement the `ResourceLoader` interface).

Since an `ApplicationContext` is a `ResourceLoader`, the bean could also implement the `ApplicationContextAware` interface and use the supplied application context directly to load resources. However, in general, it is better to use the specialized `ResourceLoader` interface if that is all you need. The code would be coupled only to the resource loading interface (which can be considered a utility interface) and not to the whole Spring `ApplicationContext` interface.

In application components, you may also rely upon autowiring of the `ResourceLoader` as an alternative to implementing the `ResourceLoaderAware` interface. The *traditional* `constructor` and `byType` autowiring modes (as described in Autowiring Collaborators) are capable of providing a `ResourceLoader` for either a constructor argument or a setter method parameter, respectively. For more flexibility (including the ability to autowire fields and multiple parameter methods), consider using the annotation-based autowiring features. In that case, the `ResourceLoader` is autowired into a field, constructor argument, or method parameter that expects the `ResourceLoader` type as long as the field, constructor, or method in question carries the `@Autowired` annotation. For more information, see Using `@Autowired`.

To load one or more `Resource` objects for a resource path that contains wildcards or makes use of the special `classpath*:` resource prefix, consider having an instance of `ResourcePatternResolver` autowired into your application components instead of `ResourceLoader`.

2.2.7. Resources as Dependencies

If the bean itself is going to determine and supply the resource path through some sort of dynamic process, it probably makes sense for the bean to use the `ResourceLoader` or `ResourcePatternResolver` interface to load resources. For example, consider the loading of a template of some sort, where the specific resource that is needed depends on the role of the user. If the resources are static, it makes sense to eliminate the use of the `ResourceLoader` interface (or `ResourcePatternResolver` interface) completely, have the bean expose the `Resource` properties it needs, and expect them to be injected into it.

What makes it trivial to then inject these properties is that all application contexts register and use a special JavaBeans `PropertyEditor`, which can convert `String` paths to `Resource` objects. For example, the following `MyBean` class has a `template` property of type `Resource`.

Java

```
package example;

public class MyBean {

    private Resource template;

    public setTemplate(Resource template) {
        this.template = template;
    }

    // ...
}
```

Kotlin

```
class MyBean(var template: Resource)
```

In an XML configuration file, the `template` property can be configured with a simple string for that resource, as the following example shows:

```
<bean id="myBean" class="example.MyBean">
    <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

Note that the resource path has no prefix. Consequently, because the application context itself is going to be used as the `ResourceLoader`, the resource is loaded through a `ClassPathResource`, a `FileSystemResource`, or a `ServletContextResource`, depending on the exact type of the application context.

If you need to force a specific `Resource` type to be used, you can use a prefix. The following two examples show how to force a `ClassPathResource` and a `UrlResource` (the latter being used to access a file in the filesystem):

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:///some/resource/path/myTemplate.txt"/>
```

If the `MyBean` class is refactored for use with annotation-driven configuration, the path to `myTemplate.txt` can be stored under a key named `template.path` — for example, in a properties file made available to the Spring `Environment` (see Environment Abstraction). The template path can then be referenced via the `@Value` annotation using a property placeholder (see Using `@Value`). Spring will retrieve the value of the template path as a string, and a special `PropertyEditor` will convert the string to a `Resource` object to be injected into the `MyBean` constructor. The following

example demonstrates how to achieve this.

Java

```
@Component
public class MyBean {

    private final Resource template;

    public MyBean(@Value("${template.path}") Resource template) {
        this.template = template;
    }

    // ...
}
```

Kotlin

```
@Component
class MyBean(@Value("\${template.path}") private val template: Resource)
```

If we want to support multiple templates discovered under the same path in multiple locations in the classpath — for example, in multiple jars in the classpath — we can use the special `classpath*:` prefix and wildcarding to define a `templates.path` key as `classpath*:/config/templates/*.txt`. If we redefine the `MyBean` class as follows, Spring will convert the template path pattern into an array of `Resource` objects that can be injected into the `MyBean` constructor.

Java

```
@Component
public class MyBean {

    private final Resource[] templates;

    public MyBean(@Value("${templates.path}") Resource[] templates) {
        this.templates = templates;
    }

    // ...
}
```

Kotlin

```
@Component
class MyBean(@Value("\${templates.path}") private val templates: Resource[])
```

2.2.8. Application Contexts and Resource Paths

This section covers how to create application contexts with resources, including shortcuts that work with XML, how to use wildcards, and other details.

Constructing Application Contexts

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location paths of the resources, such as XML files that make up the definition of the context.

When such a location path does not have a prefix, the specific `Resource` type built from that path and used to load the bean definitions depends on and is appropriate to the specific application context. For example, consider the following example, which creates a `ClassPathXmlApplicationContext`:

Java

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

Kotlin

```
val ctx = ClassPathXmlApplicationContext("conf/appContext.xml")
```

The bean definitions are loaded from the classpath, because a `ClassPathResource` is used. However, consider the following example, which creates a `FileSystemXmlApplicationContext`:

Java

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("conf/appContext.xml")
```

Now the bean definitions are loaded from a filesystem location (in this case, relative to the current working directory).

Note that the use of the special `classpath` prefix or a standard URL prefix on the location path overrides the default type of `Resource` created to load the bean definitions. Consider the following example:

Java

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("classpath:conf/appContext.xml")
```

Using `FileSystemXmlApplicationContext` loads the bean definitions from the classpath. However, it is still a `FileSystemXmlApplicationContext`. If it is subsequently used as a `ResourceLoader`, any unprefixed paths are still treated as filesystem paths.

Constructing `ClassPathXmlApplicationContext` Instances — Shortcuts

The `ClassPathXmlApplicationContext` exposes a number of constructors to enable convenient instantiation. The basic idea is that you can supply merely a string array that contains only the filenames of the XML files themselves (without the leading path information) and also supply a `Class`. The `ClassPathXmlApplicationContext` then derives the path information from the supplied class.

Consider the following directory layout:

```
com/
  example/
    services.xml
    repositories.xml
    MessengerService.class
```

The following example shows how a `ClassPathXmlApplicationContext` instance composed of the beans defined in files named `services.xml` and `repositories.xml` (which are on the classpath) can be instantiated:

Java

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "repositories.xml"}, MessengerService.class);
```

Kotlin

```
val ctx = ClassPathXmlApplicationContext(arrayOf("services.xml", "repositories.xml"),
MessengerService::class.java)
```

See the `ClassPathXmlApplicationContext` javadoc for details on the various constructors.

Wildcards in Application Context Constructor Resource Paths

The resource paths in application context constructor values may be simple paths (as shown earlier), each of which has a one-to-one mapping to a target `Resource` or, alternately, may contain the special `classpath*:` prefix or internal Ant-style patterns (matched by using Spring's `PathMatcher` utility). Both of the latter are effectively wildcards.

One use for this mechanism is when you need to do component-style application assembly. All

components can *publish* context definition fragments to a well-known location path, and, when the final application context is created using the same path prefixed with `classpath*:`, all component fragments are automatically picked up.

Note that this wildcarding is specific to the use of resource paths in application context constructors (or when you use the `PathMatcher` utility class hierarchy directly) and is resolved at construction time. It has nothing to do with the `Resource` type itself. You cannot use the `classpath*:` prefix to construct an actual `Resource`, as a resource points to just one resource at a time.

Ant-style Patterns

Path locations can contain Ant-style patterns, as the following example shows:

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

When the path location contains an Ant-style pattern, the resolver follows a more complex procedure to try to resolve the wildcard. It produces a `Resource` for the path up to the last non-wildcard segment and obtains a URL from it. If this URL is not a `jar:` URL or container-specific variant (such as `zip:` in WebLogic, `wsjar` in WebSphere, and so on), a `java.io.File` is obtained from it and used to resolve the wildcard by traversing the filesystem. In the case of a jar URL, the resolver either gets a `java.net.JarURLConnection` from it or manually parses the jar URL and then traverses the contents of the jar file to resolve the wildcards.

Implications on Portability

If the specified path is already a `file` URL (either implicitly because the base `ResourceLoader` is a filesystem one or explicitly), wildcarding is guaranteed to work in a completely portable fashion.

If the specified path is a `classpath` location, the resolver must obtain the last non-wildcard path segment URL by making a `Classloader.getResource()` call. Since this is just a node of the path (not the file at the end), it is actually undefined (in the `ClassLoader` javadoc) exactly what sort of a URL is returned in this case. In practice, it is always a `java.io.File` representing the directory (where the classpath resource resolves to a filesystem location) or a jar URL of some sort (where the classpath resource resolves to a jar location). Still, there is a portability concern on this operation.

If a jar URL is obtained for the last non-wildcard segment, the resolver must be able to get a `java.net.JarURLConnection` from it or manually parse the jar URL, to be able to walk the contents of the jar and resolve the wildcard. This does work in most environments but fails in others, and we strongly recommend that the wildcard resolution of resources coming from jars be thoroughly tested in your specific environment before you rely on it.

The `classpath*:` Prefix

When constructing an XML-based application context, a location string may use the special `classpath*:` prefix, as the following example shows:

Java

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

Kotlin

```
val ctx = ClassPathXmlApplicationContext("classpath*:conf/appContext.xml")
```

This special prefix specifies that all classpath resources that match the given name must be obtained (internally, this essentially happens through a call to `ClassLoader.getResources(…)`) and then merged to form the final application context definition.

The wildcard classpath relies on the `getResources()` method of the underlying `ClassLoader`. As most application servers nowadays supply their own `ClassLoader` implementation, the behavior might differ, especially when dealing with jar files. A simple test to check if `classpath*` works is to use the `ClassLoader` to load a file from within a jar on the classpath: `getClass().getClassLoader().getResources("<someFileInsideTheJar>")`. Try this test with files that have the same name but reside in two different locations — for example, files with the same name and same path but in different jars on the classpath. In case an inappropriate result is returned, check the application server documentation for settings that might affect the `ClassLoader` behavior.

You can also combine the `classpath*:` prefix with a `PathMatcher` pattern in the rest of the location path (for example, `classpath*:META-INF/*-beans.xml`). In this case, the resolution strategy is fairly simple: A `ClassLoader.getResources()` call is used on the last non-wildcard path segment to get all the matching resources in the class loader hierarchy and then, off each resource, the same `PathMatcher` resolution strategy described earlier is used for the wildcard subpath.

Other Notes Relating to Wildcards

Note that `classpath*:`, when combined with Ant-style patterns, only works reliably with at least one root directory before the pattern starts, unless the actual target files reside in the file system. This means that a pattern such as `classpath*:*.xml` might not retrieve files from the root of jar files but rather only from the root of expanded directories.

Spring's ability to retrieve classpath entries originates from the JDK's `ClassLoader.getResources()` method, which only returns file system locations for an empty string (indicating potential roots to search). Spring evaluates `URLClassLoader` runtime configuration and the `java.class.path` manifest in jar files as well, but this is not guaranteed to lead to portable behavior.

The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, do not activate the `files-only` switch of the JAR task. Also, classpath directories may not get exposed based on security policies in some environments — for example, stand-alone applications on JDK 1.7.0_45 and higher (which requires 'Trusted-Library' to be set up in your manifests. See https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources).

On JDK 9's module path (Jigsaw), Spring's classpath scanning generally works as expected. Putting resources into a dedicated directory is highly recommendable here as well, avoiding the aforementioned portability problems with searching the jar file root level.

Ant-style patterns with `classpath:` resources are not guaranteed to find matching resources if the root package to search is available in multiple classpath locations. Consider the following example of a resource location:

```
com/mycompany/package1/service-context.xml
```

Now consider an Ant-style path that someone might use to try to find that file:

```
classpath:com/mycompany/**/service-context.xml
```

Such a resource may exist in only one location in the classpath, but when a path such as the preceding example is used to try to resolve it, the resolver works off the (first) URL returned by `getResource("com/mycompany");`. If this base package node exists in multiple `ClassLoader` locations, the desired resource may not exist in the first location found. Therefore, in such cases you should prefer using `classpath*:` with the same Ant-style pattern, which searches all classpath locations that contain the `com.mycompany` base package: `classpath*:com/mycompany/**/service-context.xml`.

`FileSystemResource` Caveats

A `FileSystemResource` that is not attached to a `FileSystemApplicationContext` (that is, when a `FileSystemApplicationContext` is not the actual `ResourceLoader`) treats absolute and relative paths as you would expect. Relative paths are relative to the current working directory, while absolute paths are relative to the root of the filesystem.

For backwards compatibility (historical) reasons however, this changes when the `FileSystemApplicationContext` is the `ResourceLoader`. The `FileSystemApplicationContext` forces all attached `FileSystemResource` instances to treat all location paths as relative, whether they start with a leading slash or not. In practice, this means the following examples are equivalent:

Java

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/context.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("conf/context.xml")
```

Java

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("/conf/context.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("/conf/context.xml")
```

The following examples are also equivalent (even though it would make sense for them to be different, as one case is relative and the other absolute):

Java

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

Kotlin

```
val ctx: FileSystemXmlApplicationContext = ...
ctx.getResource("some/resource/path/myTemplate.txt")
```

Java

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

Kotlin

```
val ctx: FileSystemXmlApplicationContext = ...
ctx.getResource("/some/resource/path/myTemplate.txt")
```

In practice, if you need true absolute filesystem paths, you should avoid using absolute paths with `FileSystemResource` or `FileSystemXmlApplicationContext` and force the use of a `UrlResource` by using the `file:` URL prefix. The following examples show how to do so:

Java

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

Kotlin

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:///some/resource/path/myTemplate.txt")
```

Java

```
// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:///conf/context.xml");
```

Kotlin

```
// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
val ctx = FileSystemXmlApplicationContext("file:///conf/context.xml")
```

2.3. Validation, Data Binding, and Type Conversion

There are pros and cons for considering validation as business logic, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Specifically, validation should not be tied to the web tier and should be easy to localize, and it should be possible to plug in any available validator. Considering these concerns, Spring provides a `Validator` contract that is both basic and eminently usable in every layer of an application.

Data binding is useful for letting user input be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the aptly named `DataBinder` to do exactly that. The `Validator` and the `DataBinder` make up the `validation` package, which is primarily used in but not limited to the web layer.

The `BeanWrapper` is a fundamental concept in the Spring Framework and is used in a lot of places. However, you probably do not need to use the `BeanWrapper` directly. Because this is reference documentation, however, we felt that some explanation might be in order. We explain the `BeanWrapper` in this chapter, since, if you are going to use it at all, you are most likely do so when trying to bind data to objects.

Spring's `DataBinder` and the lower-level `BeanWrapper` both use `PropertyEditorSupport` implementations to parse and format property values. The `PropertyEditor` and `PropertyEditorSupport` types are part of the JavaBeans specification and are also explained in this chapter. Spring 3 introduced a `core.convert` package that provides a general type conversion facility, as well as a higher-level "format" package for formatting UI field values. You can use these packages as simpler alternatives to `PropertyEditorSupport` implementations. They are also discussed in this chapter.

Spring supports Java Bean Validation through setup infrastructure and an adaptor to Spring's own `Validator` contract. Applications can enable Bean Validation once globally, as described in Java Bean Validation, and use it exclusively for all validation needs. In the web layer, applications can further register controller-local Spring `Validator` instances per `DataBinder`, as described in Configuring a

`DataBinder`, which can be useful for plugging in custom validation logic.

2.3.1. Validation by Using Spring's Validator Interface

Spring features a `Validator` interface that you can use to validate objects. The `Validator` interface works by using an `Errors` object so that, while validating, validators can report validation failures to the `Errors` object.

Consider the following example of a small data object:

Java

```
public class Person {

    private String name;
    private int age;

    // the usual getters and setters...
}
```

Kotlin

```
class Person(val name: String, val age: Int)
```

The next example provides validation behavior for the `Person` class by implementing the following two methods of the `org.springframework.validation.Validator` interface:

- `supports(Class)`: Can this `Validator` validate instances of the supplied `Class`?
- `validate(Object, org.springframework.validation.Errors)`: Validates the given object and, in case of validation errors, registers those with the given `Errors` object.

Implementing a `Validator` is fairly straightforward, especially when you know of the `ValidationUtils` helper class that the Spring Framework also provides. The following example implements `Validator` for `Person` instances:

Java

```
public class PersonValidator implements Validator {

    /**
     * This Validator validates only Person instances
     */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

Kotlin

```
class PersonValidator : Validator {

    /**
     * This Validator validates only Person instances
     */
    override fun supports(clazz: Class<*>): Boolean {
        return Person::class.java == clazz
    }

    override fun validate(obj: Any, e: Errors) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty")
        val p = obj as Person
        if (p.age < 0) {
            e.rejectValue("age", "negativevalue")
        } else if (p.age > 110) {
            e.rejectValue("age", "too.darn.old")
        }
    }
}
```

The `static rejectIfEmpty(..)` method on the `ValidationUtils` class is used to reject the `name` property if it is `null` or the empty string. Have a look at the `ValidationUtils` javadoc to see what functionality it provides besides the example shown previously.

While it is certainly possible to implement a single `Validator` class to validate each of the nested objects in a rich object, it may be better to encapsulate the validation logic for each nested class of

object in its own `Validator` implementation. A simple example of a “rich” object would be a `Customer` that is composed of two `String` properties (a first and a second name) and a complex `Address` object. `Address` objects may be used independently of `Customer` objects, so a distinct `AddressValidator` has been implemented. If you want your `CustomerValidator` to reuse the logic contained within the `AddressValidator` class without resorting to copy-and-paste, you can dependency-inject or instantiate an `AddressValidator` within your `CustomerValidator`, as the following example shows:

Java

```
public class CustomerValidator implements Validator {

    private final Validator addressValidator;

    public CustomerValidator(Validator addressValidator) {
        if (addressValidator == null) {
            throw new IllegalArgumentException("The supplied [Validator] is " +
                "required and must not be null.");
        }
        if (!addressValidator.supports(Address.class)) {
            throw new IllegalArgumentException("The supplied [Validator] must " +
                "support the validation of [Address] instances.");
        }
        this.addressValidator = addressValidator;
    }

    /**
     * This Validator validates Customer instances, and any subclasses of Customer too
     */
    public boolean supports(Class clazz) {
        return Customer.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
"field.required");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname",
"field.required");
        Customer customer = (Customer) target;
        try {
            errors.pushNestedPath("address");
            ValidationUtils.invokeValidator(this.addressValidator,
customer.getAddress(), errors);
        } finally {
            errors.popNestedPath();
        }
    }
}
```

Kotlin

```
class CustomerValidator(private val addressValidator: Validator) : Validator {

    init {
        if (addressValidator == null) {
            throw IllegalArgumentException("The supplied [Validator] is required and
must not be null.")
        }
        if (!addressValidator.supports(Address::class.java)) {
            throw IllegalArgumentException("The supplied [Validator] must support the
validation of [Address] instances.")
        }
    }

    /*
    * This Validator validates Customer instances, and any subclasses of Customer too
    */
    override fun supports(clazz: Class<>): Boolean {
        return Customer::class.java.isAssignableFrom(clazz)
    }

    override fun validate(target: Any, errors: Errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
"field.required")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required")
        val customer = target as Customer
        try {
            errors.pushNestedPath("address")
            ValidationUtils.invokeValidator(this.addressValidator, customer.address,
errors)
        } finally {
            errors.popNestedPath()
        }
    }
}
```

Validation errors are reported to the `Errors` object passed to the validator. In the case of Spring Web MVC, you can use the `<spring:bind/>` tag to inspect the error messages, but you can also inspect the `Errors` object yourself. More information about the methods it offers can be found in the javadoc.

2.3.2. Resolving Codes to Error Messages

We covered databinding and validation. This section covers outputting messages that correspond to validation errors. In the example shown in the preceding section, we rejected the `name` and `age` fields. If we want to output the error messages by using a `MessageSource`, we can do so using the error code we provide when rejecting the field ('name' and 'age' in this case). When you call (either directly, or indirectly, by using, for example, the `ValidationUtils` class) `rejectValue` or one of the other `reject` methods from the `Errors` interface, the underlying implementation not only registers the code you passed in but also registers a number of additional error codes. The

`MessageCodesResolver` determines which error codes the `Errors` interface registers. By default, the `DefaultMessageCodesResolver` is used, which (for example) not only registers a message with the code you gave but also registers messages that include the field name you passed to the reject method. So, if you reject a field by using `rejectValue("age", "too.darn.old")`, apart from the `too.darn.old` code, Spring also registers `too.darn.old.age` and `too.darn.old.age.int` (the first includes the field name and the second includes the type of the field). This is done as a convenience to aid developers when targeting error messages.

More information on the `MessageCodesResolver` and the default strategy can be found in the javadoc of `MessageCodesResolver` and `DefaultMessageCodesResolver`, respectively.

2.3.3. Bean Manipulation and the `BeanWrapper`

The `org.springframework.beans` package adheres to the JavaBeans standard. A JavaBean is a class with a default no-argument constructor and that follows a naming convention where (for example) a property named `bingoMadness` would have a setter method `setBingoMadness(..)` and a getter method `getBingoMadness()`. For more information about JavaBeans and the specification, see javabeans.

One quite important class in the beans package is the `BeanWrapper` interface and its corresponding implementation (`BeanWrapperImpl`). As quoted from the javadoc, the `BeanWrapper` offers functionality to set and get property values (individually or in bulk), get property descriptors, and query properties to determine if they are readable or writable. Also, the `BeanWrapper` offers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. The `BeanWrapper` also supports the ability to add standard JavaBeans `PropertyChangeListeners` and `VetoableChangeListeners`, without the need for supporting code in the target class. Last but not least, the `BeanWrapper` provides support for setting indexed properties. The `BeanWrapper` usually is not used by application code directly but is used by the `DataBinder` and the `BeanFactory`.

The way the `BeanWrapper` works is partly indicated by its name: it wraps a bean to perform actions on that bean, such as setting and retrieving properties.

Setting and Getting Basic and Nested Properties

Setting and getting properties is done through the `setPropertyValue` and `getPropertyValue` overloaded method variants of `BeanWrapper`. See their Javadoc for details. The below table shows some examples of these conventions:

Table 11. Examples of properties

Expression	Explanation
`name`	Indicates the property `name` that corresponds to the `getName()` or `isName()` and `setName(..)` methods.
`account.name`	Indicates the nested property `name` of the property `account` that corresponds to (for example) the `getAccount().setName()` or `getAccount().getName()` methods.

Expression	Explanation
`account[2]`	Indicates the *third* element of the indexed property `account`. Indexed properties can be of type `array`, `list`, or other naturally ordered collection.
`account[COMPANYNAME]`	Indicates the value of the map entry indexed by the `COMPANYNAME` key of the `account` `Map` property.

(This next section is not vitally important to you if you do not plan to work with the `BeanWrapper` directly. If you use only the `DataBinder` and the `BeanFactory` and their default implementations, you should skip ahead to the section on `PropertyEditors`.)

The following two example classes use the `BeanWrapper` to get and set properties:

Java

```
public class Company {

    private String name;
    private Employee managingDirector;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Employee getManagingDirector() {
        return this.managingDirector;
    }

    public void setManagingDirector(Employee managingDirector) {
        this.managingDirector = managingDirector;
    }
}
```

Kotlin

```
class Company {
    var name: String? = null
    var managingDirector: Employee? = null
}
```

Java

```
public class Employee {

    private String name;

    private float salary;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public float getSalary() {
        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

Kotlin

```
class Employee {
    var name: String? = null
    var salary: Float? = null
}
```

The following code snippets show some examples of how to retrieve and manipulate some of the properties of instantiated `Company`s and `Employee`s:

Java

```
BeanWrapper company = new BeanWrapperImpl(new Company());
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue value = new PropertyValue("name", "Some Company Inc.");
company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:
BeanWrapper jim = new BeanWrapperImpl(new Employee());
jim.setPropertyValue("name", "Jim Stravinsky");
company.setPropertyValue("managingDirector", jim.getWrappedInstance());

// retrieving the salary of the managingDirector through the company
Float salary = (Float) company.getPropertyValue("managingDirector.salary");
```

Kotlin

```
val company = BeanWrapperImpl(Company())
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.")
// ... can also be done like this:
val value = PropertyValue("name", "Some Company Inc.")
company.setPropertyValue(value)

// ok, let's create the director and tie it to the company:
val jim = BeanWrapperImpl(Employee())
jim.setPropertyValue("name", "Jim Stravinsky")
company.setPropertyValue("managingDirector", jim.wrappedInstance)

// retrieving the salary of the managingDirector through the company
val salary = company.getPropertyValue("managingDirector.salary") as Float?
```

Built-in `PropertyEditor` Implementations

Spring uses the concept of a `PropertyEditor` to effect the conversion between an `Object` and a `String`. It can be handy to represent properties in a different way than the object itself. For example, a `Date` can be represented in a human readable way (as the `String`: `'2007-14-09'`), while we can still convert the human readable form back to the original date (or, even better, convert any date entered in a human readable form back to `Date` objects). This behavior can be achieved by registering custom editors of type `java.beans.PropertyEditor`. Registering custom editors on a `BeanWrapper` or, alternatively, in a specific IoC container (as mentioned in the previous chapter), gives it the knowledge of how to convert properties to the desired type. For more about `PropertyEditor`, see the javadoc of the `java.beans` package from Oracle.

A couple of examples where property editing is used in Spring:

- Setting properties on beans is done by using `PropertyEditor` implementations. When you use

`String` as the value of a property of some bean that you declare in an XML file, Spring (if the setter of the corresponding property has a `Class` parameter) uses `ClassEditor` to try to resolve the parameter to a `Class` object.

- Parsing HTTP request parameters in Spring's MVC framework is done by using all kinds of `PropertyEditor` implementations that you can manually bind in all subclasses of the `CommandController`.

Spring has a number of built-in `PropertyEditor` implementations to make life easy. They are all located in the `org.springframework.beans.propertyeditors` package. Most, (but not all, as indicated in the following table) are, by default, registered by `BeanWrapperImpl`. Where the property editor is configurable in some fashion, you can still register your own variant to override the default one. The following table describes the various `PropertyEditor` implementations that Spring provides:

Table 12. Built-in `PropertyEditor` Implementations

Class	Explanation
`ByteArrayPropertyEditor`	Editor for byte arrays. Converts strings to their corresponding byte representations. Registered by default by `BeanWrapperImpl`.
`ClassEditor`	Parses Strings that represent classes to actual classes and vice-versa. When a class is not found, an `IllegalArgumentException` is thrown. By default, registered by `BeanWrapperImpl`.
`CustomBooleanEditor`	Customizable property editor for `Boolean` properties. By default, registered by `BeanWrapperImpl` but can be overridden by registering a custom instance of it as a custom editor.
`CustomCollectionEditor`	Property editor for collections, converting any source `Collection` to a given target `Collection` type.
`CustomDateEditor`	Customizable property editor for `java.util.Date`, supporting a custom `DateFormat`. NOT registered by default. Must be user-registered with the appropriate format as needed.
`CustomNumberEditor`	Customizable property editor for any `Number` subclass, such as `Integer`, `Long`, `Float`, or `Double`. By default, registered by `BeanWrapperImpl` but can be overridden by registering a custom instance of it as a custom editor.
`FileEditor`	Resolves strings to `java.io.File` objects. By default, registered by `BeanWrapperImpl`.
`InputStreamEditor`	One-way property editor that can take a string and produce (through an intermediate `ResourceEditor` and `Resource`) an `InputStream` so that `InputStream` properties may be directly set as strings. Note that the default usage does not close the `InputStream` for you. By default, registered by `BeanWrapperImpl`.
`LocaleEditor`	Can resolve strings to `Locale` objects and vice-versa (the string format is `[language]_[country]_[variant]`, same as the `toString()` method of `Locale`). Also accepts spaces as separators, as an alternative to underscores. By default, registered by `BeanWrapperImpl`.

Class	Explanation
`PatternEditor`	Can resolve strings to `java.util.regex.Pattern` objects and vice-versa.
`PropertiesEditor`	Can convert strings (formatted with the format defined in the javadoc of the `java.util.Properties` class) to `Properties` objects. By default, registered by `BeanWrapperImpl`.
`StringTrimmerEditor`	Property editor that trims strings. Optionally allows transforming an empty string into a `null` value. NOT registered by default — must be user-registered.
`URLEditor`	Can resolve a string representation of a URL to an actual `URL` object. By default, registered by `BeanWrapperImpl`.

Spring uses the `java.beans.PropertyEditorManager` to set the search path for property editors that might be needed. The search path also includes `sun.bean.editors`, which includes `PropertyEditor` implementations for types such as `Font`, `Color`, and most of the primitive types. Note also that the standard JavaBeans infrastructure automatically discovers `PropertyEditor` classes (without you having to register them explicitly) if they are in the same package as the class they handle and have the same name as that class, with `Editor` appended. For example, one could have the following class and package structure, which would be sufficient for the `SomethingEditor` class to be recognized and used as the `PropertyEditor` for `Something`-typed properties.

```
com
  chank
    pop
      Something
      SomethingEditor // the PropertyEditor for the Something class
```

Note that you can also use the standard `BeanInfo` JavaBeans mechanism here as well (described to some extent here). The following example uses the `BeanInfo` mechanism to explicitly register one or more `PropertyEditor` instances with the properties of an associated class:

```
com
  chank
    pop
      Something
      SomethingBeanInfo // the BeanInfo for the Something class
```

The following Java source code for the referenced `SomethingBeanInfo` class associates a `CustomNumberEditor` with the `age` property of the `Something` class:

Java

```
public class SomethingBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class,
true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age",
Something.class) {
                @Override
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                }
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

Kotlin

```
class SomethingBeanInfo : SimpleBeanInfo() {

    override fun getPropertyDescriptors(): Array<PropertyDescriptor> {
        try {
            val numberPE = CustomNumberEditor(Int::class.java, true)
            val ageDescriptor = object : PropertyDescriptor("age",
Something::class.java) {
                override fun createPropertyEditor(bean: Any): PropertyEditor {
                    return numberPE
                }
            }
            return arrayOf(ageDescriptor)
        } catch (ex: IntrospectionException) {
            throw Error(ex.toString())
        }

    }
}
```

Registering Additional Custom `PropertyEditor` Implementations

When setting bean properties as string values, a Spring IoC container ultimately uses standard JavaBeans `PropertyEditor` implementations to convert these strings to the complex type of the property. Spring pre-registers a number of custom `PropertyEditor` implementations (for example, to convert a class name expressed as a string into a `Class` object). Additionally, Java's standard

JavaBeans `PropertyEditor` lookup mechanism lets a `PropertyEditor` for a class be named appropriately and placed in the same package as the class for which it provides support, so that it can be found automatically.

If there is a need to register other custom `PropertyEditors`, several mechanisms are available. The most manual approach, which is not normally convenient or recommended, is to use the `registerCustomEditor()` method of the `ConfigurableBeanFactory` interface, assuming you have a `BeanFactory` reference. Another (slightly more convenient) mechanism is to use a special bean factory post-processor called `CustomEditorConfigurer`. Although you can use bean factory post-processors with `BeanFactory` implementations, the `CustomEditorConfigurer` has a nested property setup, so we strongly recommend that you use it with the `ApplicationContext`, where you can deploy it in similar fashion to any other bean and where it can be automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use of a `BeanWrapper` to handle property conversions. The standard property editors that the `BeanWrapper` registers are listed in the previous section. Additionally, `ApplicationContext`s also override or add additional editors to handle resource lookups in a manner appropriate to the specific application context type.

Standard JavaBeans `PropertyEditor` instances are used to convert property values expressed as strings to the actual complex type of the property. You can use `CustomEditorConfigurer`, a bean factory post-processor, to conveniently add support for additional `PropertyEditor` instances to an `ApplicationContext`.

Consider the following example, which defines a user class called `ExoticType` and another class called `DependsOnExoticType`, which needs `ExoticType` set as a property:

Java

```
package example;

public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

Kotlin

```
package example

class ExoticType(val name: String)

class DependsOnExoticType {

    var type: ExoticType? = null
}
```

When things are properly set up, we want to be able to assign the type property as a string, which a `PropertyEditor` converts into an actual `ExoticType` instance. The following bean definition shows how to set up this relationship:

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

The `PropertyEditor` implementation could look similar to the following:

Java

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

Kotlin

```
// converts string representation to ExoticType object
package example

import java.beans.PropertyEditorSupport

class ExoticTypeEditor : PropertyEditorSupport() {

    override fun setAsText(text: String) {
        value = ExoticType(text.toUpperCase())
    }
}
```

Finally, the following example shows how to use `CustomEditorConfigurer` to register the new

`PropertyEditor` with the `ApplicationContext`, which will then be able to use it as needed:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
        </map>
    </property>
</bean>
```

Using `PropertyEditorRegistrar`

Another mechanism for registering property editors with the Spring container is to create and use a `PropertyEditorRegistrar`. This interface is particularly useful when you need to use the same set of property editors in several different situations. You can write a corresponding registrar and reuse it in each case. `PropertyEditorRegistrar` instances work in conjunction with an interface called `PropertyEditorRegistry`, an interface that is implemented by the Spring `BeanWrapper` (and `DataBinder`). `PropertyEditorRegistrar` instances are particularly convenient when used in conjunction with `CustomEditorConfigurer` (described here), which exposes a property called `setPropertyEditorRegistrars(..)`. `PropertyEditorRegistrar` instances added to a `CustomEditorConfigurer` in this fashion can easily be shared with `DataBinder` and Spring MVC controllers. Furthermore, it avoids the need for synchronization on custom editors: A `PropertyEditorRegistrar` is expected to create fresh `PropertyEditor` instances for each bean creation attempt.

The following example shows how to create your own `PropertyEditorRegistrar` implementation:

Java

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

Kotlin

```
package com.foo.editors.spring

import org.springframework.beans.PropertyEditorRegistrar
import org.springframework.beans.PropertyEditorRegistry

class CustomPropertyEditorRegistrar : PropertyEditorRegistrar {

    override fun registerCustomEditors(registry: PropertyEditorRegistry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType::class.java, ExoticTypeEditor())

        // you could register as many custom property editors as are required here...
    }
}
```

See also the `org.springframework.beans.support.ResourceEditorRegistrar` for an example `PropertyEditorRegistrar` implementation. Notice how in its implementation of the `registerCustomEditors(..)` method, it creates new instances of each property editor.

The next example shows how to configure a `CustomEditorConfigurer` and inject an instance of our `CustomPropertyEditorRegistrar` into it:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar"
    class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

Finally (and in a bit of a departure from the focus of this chapter) for those of you using Spring's MVC web framework, using a `PropertyEditorRegistrar` in conjunction with data-binding web controllers can be very convenient. The following example uses a `PropertyEditorRegistrar` in the implementation of an `@InitBinder` method:

Java

```
@Controller
public class RegisterUserController {

    private final PropertyEditorRegistrar customPropertyEditorRegistrar;

    RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {
        this.customPropertyEditorRegistrar = propertyEditorRegistrar;
    }

    @InitBinder
    void initBinder(WebDataBinder binder) {
        this.customPropertyEditorRegistrar.registerCustomEditors(binder);
    }

    // other methods related to registering a User
}
```

Kotlin

```
@Controller
class RegisterUserController(
    private val customPropertyEditorRegistrar: PropertyEditorRegistrar) {

    @InitBinder
    fun initBinder(binder: WebDataBinder) {
        this.customPropertyEditorRegistrar.registerCustomEditors(binder)
    }

    // other methods related to registering a User
}
```

This style of `PropertyEditor` registration can lead to concise code (the implementation of the `@InitBinder` method is only one line long) and lets common `PropertyEditor` registration code be encapsulated in a class and then shared amongst as many controllers as needed.

2.3.4. Spring Type Conversion

Spring 3 introduced a `core.convert` package that provides a general type conversion system. The system defines an SPI to implement type conversion logic and an API to perform type conversions at runtime. Within a Spring container, you can use this system as an alternative to `PropertyEditor` implementations to convert externalized bean property value strings to the required property types. You can also use the public API anywhere in your application where type conversion is needed.

Converter SPI

The SPI to implement type conversion logic is simple and strongly typed, as the following interface

definition shows:

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);
}
```

To create your own converter, implement the `Converter` interface and parameterize `S` as the type you are converting from and `T` as the type you are converting to. You can also transparently apply such a converter if a collection or array of `S` needs to be converted to an array or collection of `T`, provided that a delegating array or collection converter has been registered as well (which `DefaultConversionService` does by default).

For each call to `convert(S)`, the source argument is guaranteed to not be null. Your `Converter` may throw any unchecked exception if conversion fails. Specifically, it should throw an `IllegalArgumentException` to report an invalid source value. Take care to ensure that your `Converter` implementation is thread-safe.

Several converter implementations are provided in the `core.convert.support` package as a convenience. These include converters from strings to numbers and other common types. The following listing shows the `StringToInteger` class, which is a typical `Converter` implementation:

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }
}
```

Using `ConverterFactory`

When you need to centralize the conversion logic for an entire class hierarchy (for example, when converting from `String` to `Enum` objects), you can implement `ConverterFactory`, as the following example shows:

```
package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

Parameterize S to be the type you are converting from and R to be the base type defining the *range*

of classes you can convert to. Then implement `getConverter(Class<T>)`, where T is a subclass of R.

Consider the `StringToEnumConverterFactory` as an example:

```
package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new StringToEnumConverter(targetType);
    }

    private final class StringToEnumConverter<T extends Enum> implements
Converter<String, T> {

        private Class<T> enumType;

        public StringToEnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }

        public T convert(String source) {
            return (T) Enum.valueOf(this.enumType, source.trim());
        }
    }
}
```

Using `GenericConverter`

When you require a sophisticated `Converter` implementation, consider using the `GenericConverter` interface. With a more flexible but less strongly typed signature than `Converter`, a `GenericConverter` supports converting between multiple source and target types. In addition, a `GenericConverter` makes available source and target field context that you can use when you implement your conversion logic. Such context lets a type conversion be driven by a field annotation or by generic information declared on a field signature. The following listing shows the interface definition of `GenericConverter`:

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor
targetType);
}
```

To implement a `GenericConverter`, have `getConvertibleTypes()` return the supported source→target

type pairs. Then implement `convert(Object, TypeDescriptor, TypeDescriptor)` to contain your conversion logic. The source `TypeDescriptor` provides access to the source field that holds the value being converted. The target `TypeDescriptor` provides access to the target field where the converted value is to be set.

A good example of a `GenericConverter` is a converter that converts between a Java array and a collection. Such an `ArrayToCollectionConverter` introspects the field that declares the target collection type to resolve the collection's element type. This lets each element in the source array be converted to the collection element type before the collection is set on the target field.

Because `GenericConverter` is a more complex SPI interface, you should use it only when you need it. Favor `Converter` or `ConverterFactory` for basic type conversion needs.

Using `ConditionalGenericConverter`

Sometimes, you want a `Converter` to run only if a specific condition holds true. For example, you might want to run a `Converter` only if a specific annotation is present on the target field, or you might want to run a `Converter` only if a specific method (such as a `static valueOf` method) is defined on the target class. `ConditionalGenericConverter` is the union of the `GenericConverter` and `ConditionalConverter` interfaces that lets you define such custom matching criteria:

```
public interface ConditionalConverter {

    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}

public interface ConditionalGenericConverter extends GenericConverter,
ConditionalConverter {
}
```

A good example of a `ConditionalGenericConverter` is an `IdToEntityConverter` that converts between a persistent entity identifier and an entity reference. Such an `IdToEntityConverter` might match only if the target entity type declares a static finder method (for example, `findAccount(Long)`). You might perform such a finder method check in the implementation of `matches(TypeDescriptor, TypeDescriptor)`.

The `ConversionService` API

`ConversionService` defines a unified API for executing type conversion logic at runtime. Converters are often run behind the following facade interface:

```
package org.springframework.core.convert;

public interface ConversionService {

    boolean canConvert(Class<?> sourceType, Class<?> targetType);

    <T> T convert(Object source, Class<T> targetType);

    boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor
targetType);
}
```

Most `ConversionService` implementations also implement `ConverterRegistry`, which provides an SPI for registering converters. Internally, a `ConversionService` implementation delegates to its registered converters to carry out type conversion logic.

A robust `ConversionService` implementation is provided in the `core.convert.support` package. `GenericConversionService` is the general-purpose implementation suitable for use in most environments. `ConversionServiceFactory` provides a convenient factory for creating common `ConversionService` configurations.

Configuring a `ConversionService`

A `ConversionService` is a stateless object designed to be instantiated at application startup and then shared between multiple threads. In a Spring application, you typically configure a `ConversionService` instance for each Spring container (or `ApplicationContext`). Spring picks up that `ConversionService` and uses it whenever a type conversion needs to be performed by the framework. You can also inject this `ConversionService` into any of your beans and invoke it directly.

If no `ConversionService` is registered with Spring, the original `PropertyEditor`-based system is used.

To register a default `ConversionService` with Spring, add the following bean definition with an `id` of `conversionService`:

```
<bean id="conversionService"
    class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

A default `ConversionService` can convert between strings, numbers, enums, collections, maps, and other common types. To supplement or override the default converters with your own custom converters, set the `converters` property. Property values can implement any of the `Converter`, `ConverterFactory`, or `GenericConverter` interfaces.

```
<bean id="conversionService"
        class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
        <set>
            <bean class="example.MyCustomConverter"/>
        </set>
    </property>
</bean>
```

It is also common to use a `ConversionService` within a Spring MVC application. See Conversion and Formatting in the Spring MVC chapter.

In certain situations, you may wish to apply formatting during conversion. See The `FormatterRegistry` SPI for details on using `FormattingConversionServiceFactoryBean`.

Using a `ConversionService` Programmatically

To work with a `ConversionService` instance programmatically, you can inject a reference to it like you would for any other bean. The following example shows how to do so:

Java

```
@Service
public class MyService {

    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

Kotlin

```
@Service
class MyService(private val conversionService: ConversionService) {

    fun doIt() {
        conversionService.convert(...)
    }
}
```

For most use cases, you can use the `convert` method that specifies the `targetType`, but it does not work with more complex types, such as a collection of a parameterized element. For example, if you want to convert a `List` of `Integer` to a `List` of `String` programmatically, you need to provide a formal definition of the source and target types.

Fortunately, `TypeDescriptor` provides various options to make doing so straightforward, as the following example shows:

Java

```
DefaultConversionService cs = new DefaultConversionService();

List<Integer> input = ...
cs.convert(input,
    TypeDescriptor.forObject(input), // List<Integer> type descriptor
    TypeDescriptor.collection(List.class, TypeDescriptor.valueOf(String.class)));
```

Kotlin

```
val cs = DefaultConversionService()

val input: List<Integer> = ...
cs.convert(input,
        TypeDescriptor.forObject(input), // List<Integer> type descriptor
        TypeDescriptor.collection(List::class.java,
TypeDescriptor.valueOf(String::class.java)))
```

Note that `DefaultConversionService` automatically registers converters that are appropriate for most environments. This includes collection converters, scalar converters, and basic `Object`-to-`String` converters. You can register the same converters with any `ConverterRegistry` by using the static `addDefaultConverters` method on the `DefaultConversionService` class.

Converters for value types are reused for arrays and collections, so there is no need to create a specific converter to convert from a `Collection` of `S` to a `Collection` of `T`, assuming that standard collection handling is appropriate.

2.3.5. Spring Field Formatting

As discussed in the previous section, `core.convert` is a general-purpose type conversion system. It provides a unified `ConversionService` API as well as a strongly typed `Converter` SPI for implementing conversion logic from one type to another. A Spring container uses this system to bind bean property values. In addition, both the Spring Expression Language (SpEL) and `DataBinder` use this system to bind field values. For example, when SpEL needs to coerce a `Short` to a `Long` to complete an `expression.setValue(Object bean, Object value)` attempt, the `core.convert` system performs the coercion.

Now consider the type conversion requirements of a typical client environment, such as a web or desktop application. In such environments, you typically convert from `String` to support the client postback process, as well as back to `String` to support the view rendering process. In addition, you often need to localize `String` values. The more general `core.convert` `Converter` SPI does not address such formatting requirements directly. To directly address them, Spring 3 introduced a convenient `Formatter` SPI that provides a simple and robust alternative to `PropertyEditor` implementations for client environments.

In general, you can use the `Converter` SPI when you need to implement general-purpose type conversion logic — for example, for converting between a `java.util.Date` and a `Long`. You can use the `Formatter` SPI when you work in a client environment (such as a web application) and need to parse and print localized field values. The `ConversionService` provides a unified type conversion API for both SPIs.

The `Formatter` SPI

The `Formatter` SPI to implement field formatting logic is simple and strongly typed. The following listing shows the `Formatter` interface definition:

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

`Formatter` extends from the `Printer` and `Parser` building-block interfaces. The following listing shows the definitions of those two interfaces:

```
public interface Printer<T> {

    String print(T fieldValue, Locale locale);
}
```

```
import java.text.ParseException;

public interface Parser<T> {

    T parse(String clientValue, Locale locale) throws ParseException;
}
```

To create your own `Formatter`, implement the `Formatter` interface shown earlier. Parameterize `T` to be the type of object you wish to format — for example, `java.util.Date`. Implement the `print()` operation to print an instance of `T` for display in the client locale. Implement the `parse()` operation to parse an instance of `T` from the formatted representation returned from the client locale. Your `Formatter` should throw a `ParseException` or an `IllegalArgumentException` if a parse attempt fails. Take care to ensure that your `Formatter` implementation is thread-safe.

The `format` subpackages provide several `Formatter` implementations as a convenience. The `number` package provides `NumberStyleFormatter`, `CurrencyStyleFormatter`, and `PercentStyleFormatter` to format `Number` objects that use a `java.text.NumberFormat`. The `datetime` package provides a `DateFormatter` to format `java.util.Date` objects with a `java.text.DateFormat`.

The following `DateFormatter` is an example `Formatter` implementation:

Java

```
package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {

    private String pattern;

    public DateFormatter(String pattern) {
        this.pattern = pattern;
    }

    public String print(Date date, Locale locale) {
        if (date == null) {
            return "";
        }
        return getDateFormat(locale).format(date);
    }

    public Date parse(String formatted, Locale locale) throws ParseException {
        if (formatted.length() == 0) {
            return null;
        }
        return getDateFormat(locale).parse(formatted);
    }

    protected DateFormat getDateFormat(Locale locale) {
        DateFormat dateFormat = new SimpleDateFormat(this.pattern, locale);
        dateFormat.setLenient(false);
        return dateFormat;
    }
}
```

Kotlin

```
class DateFormatter(private val pattern: String) : Formatter<Date> {

    override fun print(date: Date, locale: Locale)
            = getDateFormat(locale).format(date)

    @Throws(ParseException::class)
    override fun parse(formatted: String, locale: Locale)
            = getDateFormat(locale).parse(formatted)

    protected fun getDateFormat(locale: Locale): DateFormat {
        val dateFormat = SimpleDateFormat(this.pattern, locale)
        dateFormat.isLenient = false
        return dateFormat
    }
}
```

The Spring team welcomes community-driven `Formatter` contributions. See GitHub Issues to contribute.

Annotation-driven Formatting

Field formatting can be configured by field type or annotation. To bind an annotation to a `Formatter`, implement `AnnotationFormatterFactory`. The following listing shows the definition of the `AnnotationFormatterFactory` interface:

```
package org.springframework.format;

public interface AnnotationFormatterFactory<A extends Annotation> {

    Set<Class<?>> getFieldTypes();

    Printer<?> getPrinter(A annotation, Class<?> fieldType);

    Parser<?> getParser(A annotation, Class<?> fieldType);
}
```

To create an implementation:

1. Parameterize A to be the field `annotationType` with which you wish to associate formatting logic — for example `org.springframework.format.annotation.DateTimeFormat`.
2. Have `getFieldTypes()` return the types of fields on which the annotation can be used.
3. Have `getPrinter()` return a `Printer` to print the value of an annotated field.
4. Have `getParser()` return a `Parser` to parse a `clientValue` for an annotated field.

The following example `AnnotationFormatterFactory` implementation binds the `@NumberFormat` annotation to a formatter to let a number style or pattern be specified:

Java

```
public final class NumberFormatAnnotationFormatterFactory
        implements AnnotationFormatterFactory<NumberFormat> {

    public Set<Class<?>> getFieldTypes() {
        return new HashSet<Class<?>>(asList(new Class<?>[] {
            Short.class, Integer.class, Long.class, Float.class,
            Double.class, BigDecimal.class, BigInteger.class }));
    }

    public Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    public Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {
        return configureFormatterFrom(annotation, fieldType);
    }

    private Formatter<Number> configureFormatterFrom(NumberFormat annotation, Class<?>
fieldType) {
        if (!annotation.pattern().isEmpty()) {
            return new NumberStyleFormatter(annotation.pattern());
        } else {
            Style style = annotation.style();
            if (style == Style.PERCENT) {
                return new PercentStyleFormatter();
            } else if (style == Style.CURRENCY) {
                return new CurrencyStyleFormatter();
            } else {
                return new NumberStyleFormatter();
            }
        }
    }
}
```

Kotlin

```
class NumberFormatAnnotationFormatterFactory :
AnnotationFormatterFactory<NumberFormat> {

    override fun getFieldTypes(): Set<Class<*>> {
        return setOf(Short::class.java, Int::class.java, Long::class.java,
Float::class.java, Double::class.java, BigDecimal::class.java, BigInteger::class.java)
    }

    override fun getPrinter(annotation: NumberFormat, fieldType: Class<*>):
Printer<Number> {
        return configureFormatterFrom(annotation, fieldType)
    }

    override fun getParser(annotation: NumberFormat, fieldType: Class<*>):
Parser<Number> {
        return configureFormatterFrom(annotation, fieldType)
    }

    private fun configureFormatterFrom(annotation: NumberFormat, fieldType: Class<*>):
Formatter<Number> {
        return if (annotation.pattern.isNotEmpty()) {
            NumberStyleFormatter(annotation.pattern)
        } else {
            val style = annotation.style
            when {
                style === NumberFormat.Style.PERCENT -> PercentStyleFormatter()
                style === NumberFormat.Style.CURRENCY -> CurrencyStyleFormatter()
                else -> NumberStyleFormatter()
            }
        }
    }
}
```

To trigger formatting, you can annotate fields with @NumberFormat, as the following example shows:

Java

```
public class MyModel {

    @NumberFormat(style=Style.CURRENCY)
    private BigDecimal decimal;
}
```

Kotlin

```
class MyModel(
    @field:NumberFormat(style = Style.CURRENCY) private val decimal: BigDecimal
)
```

Format Annotation API

A portable format annotation API exists in the `org.springframework.format.annotation` package. You can use `@NumberFormat` to format `Number` fields such as `Double` and `Long`, and `@DateTimeFormat` to format `java.util.Date`, `java.util.Calendar`, `Long` (for millisecond timestamps) as well as JSR-310 `java.time`.

The following example uses `@DateTimeFormat` to format a `java.util.Date` as an ISO Date (yyyy-MM-dd):

Java

```
public class MyModel {

    @DateTimeFormat(iso=ISO.DATE)
    private Date date;
}
```

Kotlin

```
class MyModel(
    @DateTimeFormat(iso=ISO.DATE) private val date: Date
)
```

The `FormatterRegistry` SPI

The `FormatterRegistry` is an SPI for registering formatters and converters. `FormattingConversionService` is an implementation of `FormatterRegistry` suitable for most environments. You can programmatically or declaratively configure this variant as a Spring bean, e.g. by using `FormattingConversionServiceFactoryBean`. Because this implementation also implements `ConversionService`, you can directly configure it for use with Spring's `DataBinder` and the Spring Expression Language (SpEL).

The following listing shows the `FormatterRegistry` SPI:

```
package org.springframework.format;

public interface FormatterRegistry extends ConverterRegistry {

    void addPrinter(Printer<?> printer);

    void addParser(Parser<?> parser);

    void addFormatter(Formatter<?> formatter);

    void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

    void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?>
parser);

    void addFormatterForFieldAnnotation(AnnotationFormatterFactory<? extends
Annotation> annotationFormatterFactory);
}
```

As shown in the preceding listing, you can register formatters by field type or by annotation.

The `FormatterRegistry` SPI lets you configure formatting rules centrally, instead of duplicating such configuration across your controllers. For example, you might want to enforce that all date fields are formatted a certain way or that fields with a specific annotation are formatted in a certain way. With a shared `FormatterRegistry`, you define these rules once, and they are applied whenever formatting is needed.

The `FormatterRegistrar` SPI

`FormatterRegistrar` is an SPI for registering formatters and converters through the FormatterRegistry. The following listing shows its interface definition:

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);
}
```

A `FormatterRegistrar` is useful when registering multiple related converters and formatters for a given formatting category, such as date formatting. It can also be useful where declarative registration is insufficient — for example, when a formatter needs to be indexed under a specific field type different from its own `<T>` or when registering a `Printer`/`Parser` pair. The next section provides more information on converter and formatter registration.

Configuring Formatting in Spring MVC

See Conversion and Formatting in the Spring MVC chapter.

2.3.6. Configuring a Global Date and Time Format

By default, date and time fields not annotated with `@DateTimeFormat` are converted from strings by using the `DateFormat.SHORT` style. If you prefer, you can change this by defining your own global format.

To do that, ensure that Spring does not register default formatters. Instead, register formatters manually with the help of:

- `org.springframework.format.datetime.standard.DateTimeFormatterRegistrar`
- `org.springframework.format.datetime.DateFormatterRegistrar`

For example, the following Java configuration registers a global `yyyyMMdd` format:

Java

```
@Configuration
public class AppConfig {

    @Bean
    public FormattingConversionService conversionService() {

        // Use the DefaultFormattingConversionService but do not register defaults
        DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService(false);

        // Ensure @NumberFormat is still supported
        conversionService.addFormatterForFieldAnnotation(new
NumberFormatAnnotationFormatterFactory());

        // Register JSR-310 date conversion with a specific global format
        DateTimeFormatterRegistrar registrar = new DateTimeFormatterRegistrar();
        registrar.setDateFormatter(DateTimeFormatter.ofPattern("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        // Register date conversion with a specific global format
        DateFormatterRegistrar registrar = new DateFormatterRegistrar();
        registrar.setFormatter(new DateFormatter("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        return conversionService;
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun conversionService(): FormattingConversionService {
        // Use the DefaultFormattingConversionService but do not register defaults
        return DefaultFormattingConversionService(false).apply {

            // Ensure @NumberFormat is still supported
            addFormatterForFieldAnnotation(NumberFormatAnnotationFormatterFactory())

            // Register JSR-310 date conversion with a specific global format
            val registrar = DateTimeFormatterRegistrar()
            registrar.setDateFormatter(DateTimeFormatter.ofPattern("yyyyMMdd"))
            registrar.registerFormatters(this)

            // Register date conversion with a specific global format
            val registrar = DateFormatterRegistrar()
            registrar.setFormatter(DateFormatter("yyyyMMdd"))
            registrar.registerFormatters(this)
        }
    }
}
```

If you prefer XML-based configuration, you can use a `FormattingConversionServiceFactoryBean`. The following example shows how to do so:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="conversionService"
class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="registerDefaultFormatters" value="false" />
        <property name="formatters">
            <set>
                <bean
class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean
class="org.springframework.format.datetime.standard.DateTimeFormatterRegistrar">
                    <property name="dateFormatter">
                        <bean
class="org.springframework.format.datetime.standard.DateTimeFormatterFactoryBean">
                            <property name="pattern" value="yyyyMMdd"/>
                        </bean>
                    </property>
                </bean>
            </set>
        </property>
    </bean>
</beans>
```

Note there are extra considerations when configuring date and time formats in web applications. Please see WebMVC Conversion and Formatting or WebFlux Conversion and Formatting.

2.3.7. Java Bean Validation

The Spring Framework provides support for the Java Bean Validation API.

Overview of Bean Validation

Bean Validation provides a common way of validation through constraint declaration and metadata for Java applications. To use it, you annotate domain model properties with declarative validation constraints which are then enforced by the runtime. There are built-in constraints, and you can also define your own custom constraints.

Consider the following example, which shows a simple `PersonForm` model with two properties:

Java

```
public class PersonForm {
    private String name;
    private int age;
}
```

Kotlin

```
class PersonForm(
        private val name: String,
        private val age: Int
)
```

Bean Validation lets you declare constraints as the following example shows:

Java

```
public class PersonForm {

    @NotNull
    @Size(max=64)
    private String name;

    @Min(0)
    private int age;
}
```

Kotlin

```
class PersonForm(
    @get:NotNull @get:Size(max=64)
    private val name: String,
    @get:Min(0)
    private val age: Int
)
```

A Bean Validation validator then validates instances of this class based on the declared constraints. See Bean Validation for general information about the API. See the Hibernate Validator documentation for specific constraints. To learn how to set up a bean validation provider as a Spring bean, keep reading.

Configuring a Bean Validation Provider

Spring provides full support for the Bean Validation API including the bootstrapping of a Bean Validation provider as a Spring bean. This lets you inject a `jakarta.validation.ValidatorFactory` or `jakarta.validation.Validator` wherever validation is needed in your application.

You can use the `LocalValidatorFactoryBean` to configure a default Validator as a Spring bean, as the following example shows:

Java

```
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration
public class AppConfig {

    @Bean
    public LocalValidatorFactoryBean validator() {
        return new LocalValidatorFactoryBean();
    }
}
```

XML

```
<bean id="validator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

The basic configuration in the preceding example triggers bean validation to initialize by using its default bootstrap mechanism. A Bean Validation provider, such as the Hibernate Validator, is expected to be present in the classpath and is automatically detected.

Injecting a Validator

`LocalValidatorFactoryBean` implements both `jakarta.validation.ValidatorFactory` and `jakarta.validation.Validator`, as well as Spring's `org.springframework.validation.Validator`. You can inject a reference to either of these interfaces into beans that need to invoke validation logic.

You can inject a reference to `jakarta.validation.Validator` if you prefer to work with the Bean Validation API directly, as the following example shows:

Java

```
import jakarta.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
}
```

Kotlin

```
import jakarta.validation.Validator;

@Service
class MyService(@Autowired private val validator: Validator)
```

You can inject a reference to `org.springframework.validation.Validator` if your bean requires the Spring Validation API, as the following example shows:

Java

```
import org.springframework.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
}
```

Kotlin

```
import org.springframework.validation.Validator

@Service
class MyService(@Autowired private val validator: Validator)
```

Configuring Custom Constraints

Each bean validation constraint consists of two parts:

- A `@Constraint` annotation that declares the constraint and its configurable properties.
- An implementation of the `jakarta.validation.ConstraintValidator` interface that implements the constraint's behavior.

To associate a declaration with an implementation, each `@Constraint` annotation references a corresponding `ConstraintValidator` implementation class. At runtime, a `ConstraintValidatorFactory` instantiates the referenced implementation when the constraint annotation is encountered in your domain model.

By default, the `LocalValidatorFactoryBean` configures a `SpringConstraintValidatorFactory` that uses Spring to create `ConstraintValidator` instances. This lets your custom `ConstraintValidators` benefit from dependency injection like any other Spring bean.

The following example shows a custom `@Constraint` declaration followed by an associated `ConstraintValidator` implementation that uses Spring for dependency injection:

Java

```
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy=MyConstraintValidator.class)
public @interface MyConstraint {
}
```

Kotlin

```
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.FIELD)
@Retention(AnnotationRetention.RUNTIME)
@Constraint(validatedBy = MyConstraintValidator::class)
annotation class MyConstraint
```

Java

```
import jakarta.validation.ConstraintValidator;

public class MyConstraintValidator implements ConstraintValidator {

    @Autowired;
    private Foo aDependency;

    // ...
}
```

Kotlin

```
import jakarta.validation.ConstraintValidator

class MyConstraintValidator(private val aDependency: Foo) : ConstraintValidator {

    // ...
}
```

As the preceding example shows, a `ConstraintValidator` implementation can have its dependencies `@Autowired` as any other Spring bean.

Spring-driven Method Validation

You can integrate the method validation feature supported by Bean Validation 1.1 (and, as a custom extension, also by Hibernate Validator 4.3) into a Spring context through a `MethodValidationPostProcessor` bean definition:

Java

```
import org.springframework.validation.beanvalidation.MethodValidationPostProcessor;

@Configuration
public class AppConfig {

    @Bean
    public MethodValidationPostProcessor validationPostProcessor() {
        return new MethodValidationPostProcessor();
    }
}
```

XML

```
<bean
class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>
```

To be eligible for Spring-driven method validation, all target classes need to be annotated with Spring's `@Validated` annotation, which can optionally also declare the validation groups to use. See `MethodValidationPostProcessor` for setup details with the Hibernate Validator and Bean Validation 1.1 providers.

Method validation relies on AOP Proxies around the target classes, either JDK dynamic proxies for methods on interfaces or CGLIB proxies. There are certain limitations with the use of proxies, some of which are described in Understanding AOP Proxies. In addition remember to always use methods and accessors on proxied classes; direct field access will not work.

Additional Configuration Options

The default `LocalValidatorFactoryBean` configuration suffices for most cases. There are a number of configuration options for various Bean Validation constructs, from message interpolation to traversal resolution. See the `LocalValidatorFactoryBean` javadoc for more information on these options.

Configuring a `DataBinder`

Since Spring 3, you can configure a `DataBinder` instance with a `Validator`. Once configured, you can invoke the `Validator` by calling `binder.validate()`. Any validation `Errors` are automatically added to the binder's `BindingResult`.

The following example shows how to use a `DataBinder` programmatically to invoke validation logic after binding to a target object:

Java

```
Foo target = new Foo();
DataBinder binder = new DataBinder(target);
binder.setValidator(new FooValidator());

// bind to the target object
binder.bind(propertyValues);

// validate the target object
binder.validate();

// get BindingResult that includes any validation errors
BindingResult results = binder.getBindingResult();
```

Kotlin

```
val target = Foo()
val binder = DataBinder(target)
binder.validator = FooValidator()

// bind to the target object
binder.bind(propertyValues)

// validate the target object
binder.validate()

// get BindingResult that includes any validation errors
val results = binder.bindingResult
```

You can also configure a `DataBinder` with multiple `Validator` instances through `dataBinder.addValidators` and `dataBinder.replaceValidators`. This is useful when combining globally configured bean validation with a Spring `Validator` configured locally on a DataBinder instance. See Spring MVC Validation Configuration.

Spring MVC 3 Validation

See Validation in the Spring MVC chapter.

2.4. Spring Expression Language (SpEL)

The Spring Expression Language ("SpEL" for short) is a powerful expression language that supports querying and manipulating an object graph at runtime. The language syntax is similar to Unified EL but offers additional features, most notably method invocation and basic string templating functionality.

While there are several other Java expression languages available — OGNL, MVEL, and JBoss EL, to name a few — the Spring Expression Language was created to provide the Spring community with a single well supported expression language that can be used across all the products in the Spring

portfolio. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the Spring Tools for Eclipse. That said, SpEL is based on a technology-agnostic API that lets other expression language implementations be integrated, should the need arise.

While SpEL serves as the foundation for expression evaluation within the Spring portfolio, it is not directly tied to Spring and can be used independently. To be self contained, many of the examples in this chapter use SpEL as if it were an independent expression language. This requires creating a few bootstrapping infrastructure classes, such as the parser. Most Spring users need not deal with this infrastructure and can, instead, author only expression strings for evaluation. An example of this typical use is the integration of SpEL into creating XML or annotation-based bean definitions, as shown in Expression support for defining bean definitions.

This chapter covers the features of the expression language, its API, and its language syntax. In several places, `Inventor` and `Society` classes are used as the target objects for expression evaluation. These class declarations and the data used to populate them are listed at the end of the chapter.

The expression language supports the following functionality:

- Literal expressions
- Boolean and relational operators
- Regular expressions
- Class expressions
- Accessing properties, arrays, lists, and maps
- Method invocation
- Relational operators
- Assignment
- Calling constructors
- Bean references
- Array construction
- Inline lists
- Inline maps
- Ternary operator
- Variables
- User-defined functions
- Collection projection
- Collection selection
- Templated expressions

2.4.1. Evaluation

This section introduces the simple use of SpEL interfaces and its expression language. The complete

language reference can be found in Language Reference.

The following code introduces the SpEL API to evaluate the literal string expression, `Hello World`.

Java

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'"); ①
String message = (String) exp.getValue();
```

① The value of the message variable is `'Hello World'`.

Kotlin

```
val parser = SpelExpressionParser()
val exp = parser.parseExpression("'Hello World'") ①
val message = exp.value as String
```

① The value of the message variable is `'Hello World'`.

The SpEL classes and interfaces you are most likely to use are located in the `org.springframework.expression` package and its sub-packages, such as `spel.support`.

The `ExpressionParser` interface is responsible for parsing an expression string. In the preceding example, the expression string is a string literal denoted by the surrounding single quotation marks. The `Expression` interface is responsible for evaluating the previously defined expression string. Two exceptions that can be thrown, `ParseException` and `EvaluationException`, when calling `parser.parseExpression` and `exp.getValue`, respectively.

SpEL supports a wide range of features, such as calling methods, accessing properties, and calling constructors.

In the following example of method invocation, we call the `concat` method on the string literal:

Java

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!')"); ①
String message = (String) exp.getValue();
```

① The value of `message` is now 'Hello World!'.

Kotlin

```
val parser = SpelExpressionParser()
val exp = parser.parseExpression("'Hello World'.concat('!')") ①
val message = exp.value as String
```

① The value of `message` is now 'Hello World!'.

The following example of calling a JavaBean property calls the `String` property `Bytes`:

Java

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes"); ①
byte[] bytes = (byte[]) exp.getValue();
```

① This line converts the literal to a byte array.

Kotlin

```
val parser = SpelExpressionParser()

// invokes 'getBytes()'
val exp = parser.parseExpression("'Hello World'.bytes") ①
val bytes = exp.value as ByteArray
```

① This line converts the literal to a byte array.

SpEL also supports nested properties by using the standard dot notation (such as `prop1.prop2.prop3`) and also the corresponding setting of property values. Public fields may also be accessed.

The following example shows how to use dot notation to get the length of a literal:

Java

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length"); ①
int length = (Integer) exp.getValue();
```

① `'Hello World'.bytes.length` gives the length of the literal.

Kotlin

```
val parser = SpelExpressionParser()

// invokes 'getBytes().length'
val exp = parser.parseExpression("'Hello World'.bytes.length") ①
val length = exp.value as Int
```

① `'Hello World'.bytes.length` gives the length of the literal.

The String's constructor can be called instead of using a string literal, as the following example shows:

Java

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()"); ①
String message = exp.getValue(String.class);
```

① Construct a new `String` from the literal and make it be upper case.

Kotlin

```
val parser = SpelExpressionParser()
val exp = parser.parseExpression("new String('hello world').toUpperCase()")  ①
val message = exp.getValue(String::class.java)
```

① Construct a new `String` from the literal and make it be upper case.

Note the use of the generic method: `public <T> T getValue(Class<T> desiredResultType)`. Using this method removes the need to cast the value of the expression to the desired result type. An `EvaluationException` is thrown if the value cannot be cast to the type `T` or converted by using the registered type converter.

The more common usage of SpEL is to provide an expression string that is evaluated against a specific object instance (called the root object). The following example shows how to retrieve the `name` property from an instance of the `Inventor` class or create a boolean condition:

Java

```
// Create and set a calendar
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("name"); // Parse name as an expression
String name = (String) exp.getValue(tesla);
// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(tesla, Boolean.class);
// result == true
```

Kotlin

```
// Create and set a calendar
val c = GregorianCalendar()
c.set(1856, 7, 9)

// The constructor arguments are name, birthday, and nationality.
val tesla = Inventor("Nikola Tesla", c.time, "Serbian")

val parser = SpelExpressionParser()

var exp = parser.parseExpression("name") // Parse name as an expression
val name = exp.getValue(tesla) as String
// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'")
val result = exp.getValue(tesla, Boolean::class.java)
// result == true
```

Understanding `EvaluationContext`

The `EvaluationContext` interface is used when evaluating an expression to resolve properties, methods, or fields and to help perform type conversion. Spring provides two implementations.

- `SimpleEvaluationContext`: Exposes a subset of essential SpEL language features and configuration options, for categories of expressions that do not require the full extent of the SpEL language syntax and should be meaningfully restricted. Examples include but are not limited to data binding expressions and property-based filters.
- `StandardEvaluationContext`: Exposes the full set of SpEL language features and configuration options. You can use it to specify a default root object and to configure every available evaluation-related strategy.

`SimpleEvaluationContext` is designed to support only a subset of the SpEL language syntax. It excludes Java type references, constructors, and bean references. It also requires you to explicitly choose the level of support for properties and methods in expressions. By default, the `create()` static factory method enables only read access to properties. You can also obtain a builder to configure the exact level of support needed, targeting one or some combination of the following:

- Custom `PropertyAccessor` only (no reflection)
- Data binding properties for read-only access
- Data binding properties for read and write

Type Conversion

By default, SpEL uses the conversion service available in Spring core (`org.springframework.core.convert.ConversionService`). This conversion service comes with many built-in converters for common conversions but is also fully extensible so that you can add custom conversions between types. Additionally, it is generics-aware. This means that, when you work with

generic types in expressions, SpEL attempts conversions to maintain type correctness for any objects it encounters.

What does this mean in practice? Suppose assignment, using `setValue()`, is being used to set a `List` property. The type of the property is actually `List<Boolean>`. SpEL recognizes that the elements of the list need to be converted to `Boolean` before being placed in it. The following example shows how to do so:

Java

```
class Simple {
    public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();
simple.booleanList.add(true);

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

// "false" is passed in here as a String. SpEL and the conversion service
// will recognize that it needs to be a Boolean and convert it accordingly.
parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

// b is false
Boolean b = simple.booleanList.get(0);
```

Kotlin

```
class Simple {
    var booleanList: MutableList<Boolean> = ArrayList()
}

val simple = Simple()
simple.booleanList.add(true)

val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

// "false" is passed in here as a String. SpEL and the conversion service
// will recognize that it needs to be a Boolean and convert it accordingly.
parser.parseExpression("booleanList[0]").setValue(context, simple, "false")

// b is false
val b = simple.booleanList[0]
```

Parser Configuration

It is possible to configure the SpEL expression parser by using a parser configuration object (`org.springframework.expression.spel.SpelParserConfiguration`). The configuration object controls the behavior of some of the expression components. For example, if you index into an array or

collection and the element at the specified index is `null`, SpEL can automatically create the element. This is useful when using expressions made up of a chain of property references. If you index into an array or list and specify an index that is beyond the end of the current size of the array or list, SpEL can automatically grow the array or list to accommodate that index. In order to add an element at the specified index, SpEL will try to create the element using the element type's default constructor before setting the specified value. If the element type does not have a default constructor, `null` will be added to the array or list. If there is no built-in or custom converter that knows how to set the value, `null` will remain in the array or list at the specified index. The following example demonstrates how to automatically grow the list:

Java

```
class Demo {
    public List<String> list;
}

// Turn on:
// - auto null reference initialization
// - auto collection growing
SpelParserConfiguration config = new SpelParserConfiguration(true, true);

ExpressionParser parser = new SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = new Demo();

Object o = expression.getValue(demo);

// demo.list will now be a real collection of 4 entries
// Each entry is a new empty String
```

Kotlin

```
class Demo {
    var list: List<String>? = null
}

// Turn on:
// - auto null reference initialization
// - auto collection growing
val config = SpelParserConfiguration(true, true)

val parser = SpelExpressionParser(config)

val expression = parser.parseExpression("list[3]")

val demo = Demo()

val o = expression.getValue(demo)

// demo.list will now be a real collection of 4 entries
// Each entry is a new empty String
```

SpEL Compilation

Spring Framework 4.1 includes a basic expression compiler. Expressions are usually interpreted, which provides a lot of dynamic flexibility during evaluation but does not provide optimum performance. For occasional expression usage, this is fine, but, when used by other components such as Spring Integration, performance can be very important, and there is no real need for the dynamism.

The SpEL compiler is intended to address this need. During evaluation, the compiler generates a Java class that embodies the expression behavior at runtime and uses that class to achieve much faster expression evaluation. Due to the lack of typing around expressions, the compiler uses information gathered during the interpreted evaluations of an expression when performing compilation. For example, it does not know the type of a property reference purely from the expression, but during the first interpreted evaluation, it finds out what it is. Of course, basing compilation on such derived information can cause trouble later if the types of the various expression elements change over time. For this reason, compilation is best suited to expressions whose type information is not going to change on repeated evaluations.

Consider the following basic expression:

```
someArray[0].someProperty.someOtherProperty < 0.1
```

Because the preceding expression involves array access, some property de-referencing, and numeric operations, the performance gain can be very noticeable. In an example micro benchmark run of 50000 iterations, it took 75ms to evaluate by using the interpreter and only 3ms using the compiled version of the expression.

Compiler Configuration

The compiler is not turned on by default, but you can turn it on in either of two different ways. You can turn it on by using the parser configuration process (discussed earlier) or by using a Spring property when SpEL usage is embedded inside another component. This section discusses both of these options.

The compiler can operate in one of three modes, which are captured in the `org.springframework.expression.spel.SpelCompilerMode` enum. The modes are as follows:

- `OFF` (default): The compiler is switched off.
- `IMMEDIATE`: In immediate mode, the expressions are compiled as soon as possible. This is typically after the first interpreted evaluation. If the compiled expression fails (typically due to a type changing, as described earlier), the caller of the expression evaluation receives an exception.
- `MIXED`: In mixed mode, the expressions silently switch between interpreted and compiled mode over time. After some number of interpreted runs, they switch to compiled form and, if something goes wrong with the compiled form (such as a type changing, as described earlier), the expression automatically switches back to interpreted form again. Sometime later, it may generate another compiled form and switch to it. Basically, the exception that the user gets in `IMMEDIATE` mode is instead handled internally.

`IMMEDIATE` mode exists because `MIXED` mode could cause issues for expressions that have side effects. If a compiled expression blows up after partially succeeding, it may have already done something that has affected the state of the system. If this has happened, the caller may not want it to silently re-run in interpreted mode, since part of the expression may be running twice.

After selecting a mode, use the `SpelParserConfiguration` to configure the parser. The following example shows how to do so:

Java

```
SpelParserConfiguration config = new
SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,
        this.getClass().getClassLoader());

SpelExpressionParser parser = new SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = new MyMessage();

Object payload = expr.getValue(message);
```

Kotlin

```
val config = SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,
        this.javaClass.classLoader)

val parser = SpelExpressionParser(config)

val expr = parser.parseExpression("payload")

val message = MyMessage()

val payload = expr.getValue(message)
```

When you specify the compiler mode, you can also specify a classloader (passing null is allowed). Compiled expressions are defined in a child classloader created under any that is supplied. It is important to ensure that, if a classloader is specified, it can see all the types involved in the expression evaluation process. If you do not specify a classloader, a default classloader is used (typically the context classloader for the thread that is running during expression evaluation).

The second way to configure the compiler is for use when SpEL is embedded inside some other component and it may not be possible to configure it through a configuration object. In these cases, it is possible to set the `spring.expression.compiler.mode` property via a JVM system property (or via the `SpringProperties` mechanism) to one of the `SpelCompilerMode` enum values (`off`, `immediate`, or `mixed`).

Compiler Limitations

Since Spring Framework 4.1, the basic compilation framework is in place. However, the framework does not yet support compiling every kind of expression. The initial focus has been on the common expressions that are likely to be used in performance-critical contexts. The following kinds of expression cannot be compiled at the moment:

- Expressions involving assignment
- Expressions relying on the conversion service
- Expressions using custom resolvers or accessors
- Expressions using selection or projection

More types of expressions will be compilable in the future.

2.4.2. Expressions in Bean Definitions

You can use SpEL expressions with XML-based or annotation-based configuration metadata for defining `BeanDefinition` instances. In both cases, the syntax to define the expression is of the form `#{ <expression string> }`.

XML Configuration

A property or constructor argument value can be set by using expressions, as the following

example shows:

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>
```

All beans in the application context are available as predefined variables with their common bean name. This includes standard context beans such as `environment` (of type `org.springframework.core.env.Environment`) as well as `systemProperties` and `systemEnvironment` (of type `Map<String, Object>`) for access to the runtime environment.

The following example shows access to the `systemProperties` bean as a SpEL variable:

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

    <!-- other properties -->
</bean>
```

Note that you do not have to prefix the predefined variable with the `#` symbol here.

You can also refer to other bean properties by name, as the following example shows:

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

    <!-- other properties -->
</bean>
```

Annotation Configuration

To specify a default value, you can place the `@Value` annotation on fields, methods, and method or constructor parameters.

The following example sets the default value of a field:

Java

```
public class FieldValueTestBean {

    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }
}
```

Kotlin

```
class FieldValueTestBean {

    @Value("#{ systemProperties['user.region'] }")
    var defaultLocale: String? = null
}
```

The following example shows the equivalent but on a property setter method:

Java

```
public class PropertyValueTestBean {

    private String defaultLocale;

    @Value("#{ systemProperties['user.region'] }")
    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }
}
```

Kotlin

```
class PropertyValueTestBean {

    @Value("#{ systemProperties['user.region'] }")
    var defaultLocale: String? = null
}
```

Autowired methods and constructors can also use the `@Value` annotation, as the following examples show:

Java

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;
    private String defaultLocale;

    @Autowired
    public void configure(MovieFinder movieFinder,
            @Value("#{ systemProperties['user.region'] }") String defaultLocale) {
        this.movieFinder = movieFinder;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

Kotlin

```
class SimpleMovieLister {

    private lateinit var movieFinder: MovieFinder
    private lateinit var defaultLocale: String

    @Autowired
    fun configure(movieFinder: MovieFinder,
                  @Value("#{ systemProperties['user.region'] }") defaultLocale: String)
{
        this.movieFinder = movieFinder
        this.defaultLocale = defaultLocale
    }

    // ...
}
```

Java

```
public class MovieRecommender {

    private String defaultLocale;

    private CustomerPreferenceDao customerPreferenceDao;

    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao,
            @Value("#{systemProperties['user.country']}") String defaultLocale) {
        this.customerPreferenceDao = customerPreferenceDao;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender(private val customerPreferenceDao: CustomerPreferenceDao,
            @Value("#{systemProperties['user.country']}") private val defaultLocale:
String) {
    // ...
}
```

2.4.3. Language Reference

This section describes how the Spring Expression Language works. It covers the following topics:

- Literal Expressions
- Properties, Arrays, Lists, Maps, and Indexers
- Inline Lists
- Inline Maps
- Array Construction
- Methods
- Operators
- Types
- Constructors
- Variables
- Functions
- Bean References
- Ternary Operator (If-Then-Else)
- The Elvis Operator

- Safe Navigation Operator

Literal Expressions

The types of literal expressions supported are strings, numeric values (int, real, hex), boolean, and null. Strings are delimited by single quotation marks. To put a single quotation mark itself in a string, use two single quotation mark characters.

The following listing shows simple usage of literals. Typically, they are not used in isolation like this but, rather, as part of a more complex expression — for example, using a literal on one side of a logical comparison operator.

Java

```
ExpressionParser parser = new SpelExpressionParser();

// evals to "Hello World"
String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

double avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

// evals to 2147483647
int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();
```

Kotlin

```
val parser = SpelExpressionParser()

// evals to "Hello World"
val helloWorld = parser.parseExpression("'Hello World'").value as String

val avogadrosNumber = parser.parseExpression("6.0221415E+23").value as Double

// evals to 2147483647
val maxValue = parser.parseExpression("0x7FFFFFFF").value as Int

val trueValue = parser.parseExpression("true").value as Boolean

val nullValue = parser.parseExpression("null").value
```

Numbers support the use of the negative sign, exponential notation, and decimal points. By default, real numbers are parsed by using `Double.parseDouble()`.

Properties, Arrays, Lists, Maps, and Indexers

Navigating with property references is easy. To do so, use a period to indicate a nested property

value. The instances of the `Inventor` class, `pupin` and `tesla`, were populated with data listed in the Classes used in the examples section. To navigate "down" the object graph and get Tesla's year of birth and Pupin's city of birth, we use the following expressions:

Java

```
// evals to 1856
int year = (Integer) parser.parseExpression("birthdate.year +
1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.city").getValue(context);
```

Kotlin

```
// evals to 1856
val year = parser.parseExpression("birthdate.year + 1900").getValue(context) as Int

val city = parser.parseExpression("placeOfBirth.city").getValue(context) as String
```

Case insensitivity is allowed for the first letter of property names. Thus, the expressions in the above example may be written as `Birthdate.Year + 1900` and `PlaceOfBirth.City`, respectively. In addition, properties may optionally be accessed via method invocations — for example, `getPlaceOfBirth().getCity()` instead of `placeOfBirth.city`.

The contents of arrays and lists are obtained by using square bracket notation, as the following example shows:

Java

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

// Inventions Array

// evaluates to "Induction motor"
String invention = parser.parseExpression("inventions[3]").getValue(
        context, tesla, String.class);

// Members List

// evaluates to "Nikola Tesla"
String name = parser.parseExpression("members[0].name").getValue(
        context, ieee, String.class);

// List and Array navigation
// evaluates to "Wireless communication"
String invention = parser.parseExpression("members[0].inventions[6]").getValue(
        context, ieee, String.class);
```

Kotlin

```
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

// Inventions Array

// evaluates to "Induction motor"
val invention = parser.parseExpression("inventions[3]").getValue(
        context, tesla, String::class.java)

// Members List

// evaluates to "Nikola Tesla"
val name = parser.parseExpression("members[0].name").getValue(
        context, ieee, String::class.java)

// List and Array navigation
// evaluates to "Wireless communication"
val invention = parser.parseExpression("members[0].inventions[6]").getValue(
        context, ieee, String::class.java)
```

The contents of maps are obtained by specifying the literal key value within the brackets. In the following example, because keys for the `officers` map are strings, we can specify string literals:

Java

```
// Officer's Dictionary

Inventor pupin = parser.parseExpression("officers['president']").getValue(
        societyContext, Inventor.class);

// evaluates to "Idvor"
String city =
parser.parseExpression("officers['president'].placeOfBirth.city").getValue(
        societyContext, String.class);

// setting values
parser.parseExpression("officers['advisors'][0].placeOfBirth.country").setValue(
        societyContext, "Croatia");
```

Kotlin

```
// Officer's Dictionary

val pupin = parser.parseExpression("officers['president']").getValue(
        societyContext, Inventor::class.java)

// evaluates to "Idvor"
val city = parser.parseExpression("officers['president'].placeOfBirth.city").getValue(
        societyContext, String::class.java)

// setting values
parser.parseExpression("officers['advisors'][0].placeOfBirth.country").setValue(
        societyContext, "Croatia")
```

Inline Lists

You can directly express lists in an expression by using {} notation.

Java

```
// evaluates to a Java list containing the four numbers
List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List)
parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);
```

Kotlin

```
// evaluates to a Java list containing the four numbers
val numbers = parser.parseExpression("{1,2,3,4}").getValue(context) as List<*>

val listOfLists = parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context) as
List<*>
```

`{}` by itself means an empty list. For performance reasons, if the list is itself entirely composed of fixed literals, a constant list is created to represent the expression (rather than building a new list on each evaluation).

Inline Maps

You can also directly express maps in an expression by using `{key:value}` notation. The following example shows how to do so:

Java

```
// evaluates to a Java map containing the two entries
Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-
1856'}").getValue(context);

Map mapOfMaps = (Map)
parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',y
ear:1856}}").getValue(context);
```

Kotlin

```
// evaluates to a Java map containing the two entries
val inventorInfo = parser.parseExpression("{name:'Nikola',dob:'10-July-
1856'}").getValue(context) as Map<*, *>

val mapOfMaps =
parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',y
ear:1856}}").getValue(context) as Map<*, *>
```

`{:}` by itself means an empty map. For performance reasons, if the map is itself composed of fixed literals or other nested constant structures (lists or maps), a constant map is created to represent the expression (rather than building a new map on each evaluation). Quoting of the map keys is optional (unless the key contains a period (`.`)). The examples above do not use quoted keys.

Array Construction

You can build arrays by using the familiar Java syntax, optionally supplying an initializer to have the array populated at construction time. The following example shows how to do so:

Java

```
int[] numbers1 = (int[]) parser.parseExpression("new int[4]").getValue(context);

// Array with initializer
int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue(context);

// Multi dimensional array
int[][] numbers3 = (int[][]) parser.parseExpression("new
int[4][5]").getValue(context);
```

Kotlin

```
val numbers1 = parser.parseExpression("new int[4]").getValue(context) as IntArray

// Array with initializer
val numbers2 = parser.parseExpression("new int[]{1,2,3}").getValue(context) as
IntArray

// Multi dimensional array
val numbers3 = parser.parseExpression("new int[4][5]").getValue(context) as
Array<IntArray>
```

You cannot currently supply an initializer when you construct a multi-dimensional array.

Methods

You can invoke methods by using typical Java programming syntax. You can also invoke methods on literals. Variable arguments are also supported. The following examples show how to invoke methods:

Java

```
// string literal, evaluates to "bc"
String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

// evaluates to true
boolean isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(
        societyContext, Boolean.class);
```

Kotlin

```
// string literal, evaluates to "bc"
val bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String::class.java)

// evaluates to true
val isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(
        societyContext, Boolean::class.java)
```

Operators

The Spring Expression Language supports the following kinds of operators:

- Relational Operators
- Logical Operators
- Mathematical Operators
- The Assignment Operator

Relational Operators

The relational operators (equal, not equal, less than, less than or equal, greater than, and greater than or equal) are supported by using standard operator notation. These operators work on `Number` types as well as types implementing `Comparable`. The following listing shows a few examples of operators:

Java

```
// evaluates to true
boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

// evaluates to false
boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

// evaluates to true
boolean trueValue = parser.parseExpression("'black' <
'block'").getValue(Boolean.class);

// uses CustomValue:::compareTo
boolean trueValue = parser.parseExpression("new CustomValue(1) < new
CustomValue(2)").getValue(Boolean.class);
```

Kotlin

```
// evaluates to true
val trueValue = parser.parseExpression("2 == 2").getValue(Boolean::class.java)

// evaluates to false
val falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean::class.java)

// evaluates to true
val trueValue = parser.parseExpression("'black' <
'block'").getValue(Boolean::class.java)

// uses CustomValue:::compareTo
val trueValue = parser.parseExpression("new CustomValue(1) < new
CustomValue(2)").getValue(Boolean::class.java);
```

Greater-than and less-than comparisons against `null` follow a simple rule: `null` is treated as nothing (that is NOT as zero). As a consequence, any other value is always greater than `null` (`X > null` is always `true`) and no other value is ever less than nothing (`X < null` is always `false`).

If you prefer numeric comparisons instead, avoid number-based `null` comparisons in favor of comparisons against zero (for example, `X > 0` or `X < 0`).

In addition to the standard relational operators, SpEL supports the `instanceof` and regular expression-based `matches` operator. The following listing shows examples of both:

Java

```
// evaluates to false
boolean falseValue = parser.parseExpression(
        "'xyz' instanceof T(Integer)").getValue(Boolean.class);

// evaluates to true
boolean trueValue = parser.parseExpression(
        "'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

// evaluates to false
boolean falseValue = parser.parseExpression(
        "'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);
```

Kotlin

```
// evaluates to false
val falseValue = parser.parseExpression(
        "'xyz' instanceof T(Integer)").getValue(Boolean::class.java)

// evaluates to true
val trueValue = parser.parseExpression(
        "'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean::class.java)

// evaluates to false
val falseValue = parser.parseExpression(
        "'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean::class.java)
```

Be careful with primitive types, as they are immediately boxed up to their wrapper types. For example, `1 instanceof T(int)` evaluates to `false`, while `1 instanceof T(Integer)` evaluates to `true`, as expected.

Each symbolic operator can also be specified as a purely alphabetic equivalent. This avoids problems where the symbols used have special meaning for the document type in which the expression is embedded (such as in an XML document). The textual equivalents are:

- `lt` (`<`)

- `gt` (`>`)
- `le` (`<=`)
- `ge` (`>=`)
- `eq` (`==`)
- `ne` (`!=`)
- `div` (`/`)
- `mod` (`%`)
- `not` (`!`).

All of the textual operators are case-insensitive.

Logical Operators

SpEL supports the following logical operators:

- `and` (`&&`)
- `or` (`||`)
- `not` (`!`)

The following example shows how to use the logical operators:

Java

```
// -- AND --

// evaluates to false
boolean falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";
boolean trueValue = parser.parseExpression(expression).getValue(societyContext,
Boolean.class);

// -- OR --

// evaluates to true
boolean trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

// evaluates to true
String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";
boolean trueValue = parser.parseExpression(expression).getValue(societyContext,
Boolean.class);

// -- NOT --

// evaluates to false
boolean falseValue = parser.parseExpression("!true").getValue(Boolean.class);

// -- AND and NOT --
String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";
boolean falseValue = parser.parseExpression(expression).getValue(societyContext,
Boolean.class);
```

Kotlin

```
// -- AND --

// evaluates to false
val falseValue = parser.parseExpression("true and
false").getValue(Boolean::class.java)

// evaluates to true
val expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')"
val trueValue = parser.parseExpression(expression).getValue(societyContext,
Boolean::class.java)

// -- OR --

// evaluates to true
val trueValue = parser.parseExpression("true or false").getValue(Boolean::class.java)

// evaluates to true
val expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')"
val trueValue = parser.parseExpression(expression).getValue(societyContext,
Boolean::class.java)

// -- NOT --

// evaluates to false
val falseValue = parser.parseExpression("!true").getValue(Boolean::class.java)

// -- AND and NOT --
val expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')"
val falseValue = parser.parseExpression(expression).getValue(societyContext,
Boolean::class.java)
```

Mathematical Operators

You can use the addition operator (`+`) on both numbers and strings. You can use the subtraction (`-`), multiplication (`*`), and division (`/`) operators only on numbers. You can also use the modulus (`%`) and exponential power (`^`) operators on numbers. Standard operator precedence is enforced. The following example shows the mathematical operators in use:

Java

```
// Addition
int two = parser.parseExpression("1 + 1").getValue(Integer.class);  // 2

String testString = parser.parseExpression(
        "'test' + ' ' + 'string'").getValue(String.class);  // 'test string'

// Subtraction
int four = parser.parseExpression("1 - -3").getValue(Integer.class);  // 4

double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class);  // -9000

// Multiplication
int six = parser.parseExpression("-2 * -3").getValue(Integer.class);  // 6

double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double.class);
// 24.0

// Division
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);  // -2

double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class);  // 1.0

// Modulus
int three = parser.parseExpression("7 % 4").getValue(Integer.class);  // 3

int one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class);  // 1

// Operator precedence
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.class);  //
-21
```

Kotlin

```
// Addition
val two = parser.parseExpression("1 + 1").getValue(Int::class.java)  // 2

val testString = parser.parseExpression(
        "'test' + ' ' + 'string'").getValue(String::class.java)  // 'test string'

// Subtraction
val four = parser.parseExpression("1 - -3").getValue(Int::class.java)  // 4

val d = parser.parseExpression("1000.00 - 1e4").getValue(Double::class.java)  // -9000

// Multiplication
val six = parser.parseExpression("-2 * -3").getValue(Int::class.java)  // 6

val twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double::class.java)
// 24.0

// Division
val minusTwo = parser.parseExpression("6 / -3").getValue(Int::class.java)  // -2

val one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double::class.java)  // 1.0

// Modulus
val three = parser.parseExpression("7 % 4").getValue(Int::class.java)  // 3

val one = parser.parseExpression("8 / 5 % 2").getValue(Int::class.java)  // 1

// Operator precedence
val minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Int::class.java)  //
-21
```

The Assignment Operator

To set a property, use the assignment operator (`=`). This is typically done within a call to `setValue` but can also be done inside a call to `getValue`. The following listing shows both ways to use the assignment operator:

Java

```
Inventor inventor = new Inventor();
EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

parser.parseExpression("name").setValue(context, inventor, "Aleksandar Seovic");

// alternatively
String aleks = parser.parseExpression(
        "name = 'Aleksandar Seovic'").getValue(context, inventor, String.class);
```

Kotlin

```
val inventor = Inventor()
val context = SimpleEvaluationContext.forReadWriteDataBinding().build()

parser.parseExpression("name").setValue(context, inventor, "Aleksandar Seovic")

// alternatively
val aleks = parser.parseExpression(
        "name = 'Aleksandar Seovic'").getValue(context, inventor, String::class.java)
```

Types

You can use the special `T` operator to specify an instance of `java.lang.Class` (the type). Static methods are invoked by using this operator as well. The `StandardEvaluationContext` uses a `TypeLocator` to find types, and the `StandardTypeLocator` (which can be replaced) is built with an understanding of the `java.lang` package. This means that `T()` references to types within the `java.lang` package do not need to be fully qualified, but all other type references must be. The following example shows how to use the `T` operator:

Java

```
Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

boolean trueValue = parser.parseExpression(
        "T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")
        .getValue(Boolean.class);
```

Kotlin

```
val dateClass =
parser.parseExpression("T(java.util.Date)").getValue(Class::class.java)

val stringClass = parser.parseExpression("T(String)").getValue(Class::class.java)

val trueValue = parser.parseExpression(
        "T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")
        .getValue(Boolean::class.java)
```

Constructors

You can invoke constructors by using the `new` operator. You should use the fully qualified class name for all types except those located in the `java.lang` package (`Integer`, `Float`, `String`, and so on). The following example shows how to use the `new` operator to invoke constructors:

Java

```
Inventor einstein = p.parseExpression(
        "new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")
        .getValue(Inventor.class);

// create new Inventor instance within the add() method of List
p.parseExpression(
        "Members.add(new org.spring.samples.spel.inventor.Inventor(
            'Albert Einstein', 'German'))").getValue(societyContext);
```

Kotlin

```
val einstein = p.parseExpression(
        "new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")
        .getValue(Inventor::class.java)

// create new Inventor instance within the add() method of List
p.parseExpression(
        "Members.add(new org.spring.samples.spel.inventor.Inventor('Albert Einstein',
'German'))")
        .getValue(societyContext)
```

Variables

You can reference variables in the expression by using the `#variableName` syntax. Variables are set by using the `setVariable` method on `EvaluationContext` implementations.

Valid variable names must be composed of one or more of the following supported characters.

- letters: `A` to `Z` and `a` to `z`
- digits: `0` to `9`
- underscore: `_`
- dollar sign: `$`

The following example shows how to use variables.

Java

```
Inventor tesla = new Inventor("Nikola Tesla", "Serbian");

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();
context.setVariable("newName", "Mike Tesla");

parser.parseExpression("name = #newName").getValue(context, tesla);
System.out.println(tesla.getName())  // "Mike Tesla"
```

Kotlin

```
val tesla = Inventor("Nikola Tesla", "Serbian")

val context = SimpleEvaluationContext.forReadWriteDataBinding().build()
context.setVariable("newName", "Mike Tesla")

parser.parseExpression("name = #newName").getValue(context, tesla)
println(tesla.name)  // "Mike Tesla"
```

The `#this` and `#root` Variables

The `#this` variable is always defined and refers to the current evaluation object (against which unqualified references are resolved). The `#root` variable is always defined and refers to the root context object. Although `#this` may vary as components of an expression are evaluated, `#root` always refers to the root. The following examples show how to use the `#this` and `#root` variables:

Java

```
// create an array of integers
List<Integer> primes = new ArrayList<Integer>();
primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

// create parser and set variable 'primes' as the array of integers
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataAccess();
context.setVariable("primes", primes);

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpression(
        "#primes.?[#this>10]").getValue(context);
```

Kotlin

```
// create an array of integers
val primes = ArrayList<Int>()
primes.addAll(listOf(2, 3, 5, 7, 11, 13, 17))

// create parser and set variable 'primes' as the array of integers
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataAccess()
context.setVariable("primes", primes)

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
val primesGreaterThanTen = parser.parseExpression(
        "#primes.?[#this>10]").getValue(context) as List<Int>
```

Functions

You can extend SpEL by registering user-defined functions that can be called within the expression string. The function is registered through the `EvaluationContext`. The following example shows how to register a user-defined function:

Java

```
Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();
context.setVariable("myFunction", method);
```

Kotlin

```
val method: Method = ...

val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()
context.setVariable("myFunction", method)
```

For example, consider the following utility method that reverses a string:

Java

```
public abstract class StringUtils {

    public static String reverseString(String input) {
        StringBuilder backwards = new StringBuilder(input.length());
        for (int i = 0; i < input.length(); i++) {
            backwards.append(input.charAt(input.length() - 1 - i));
        }
        return backwards.toString();
    }
}
```

Kotlin

```
fun reverseString(input: String): String {
    val backwards = StringBuilder(input.length)
    for (i in 0 until input.length) {
        backwards.append(input[input.length - 1 - i])
    }
    return backwards.toString()
}
```

You can then register and use the preceding method, as the following example shows:

Java

```
ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();
context.setVariable("reverseString",
        StringUtils.class.getDeclaredMethod("reverseString", String.class));

String helloWorldReversed = parser.parseExpression(
        "#reverseString('hello')").getValue(context, String.class);
```

Kotlin

```
val parser = SpelExpressionParser()

val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()
context.setVariable("reverseString", ::reverseString::javaMethod)

val helloWorldReversed = parser.parseExpression(
        "#reverseString('hello')").getValue(context, String::class.java)
```

Bean References

If the evaluation context has been configured with a bean resolver, you can look up beans from an expression by using the `@` symbol. The following example shows how to do so:

Java

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"something") on MyBeanResolver during
evaluation
Object bean = parser.parseExpression("@something").getValue(context);
```

Kotlin

```
val parser = SpelExpressionParser()
val context = StandardEvaluationContext()
context.setBeanResolver(MyBeanResolver())

// This will end up calling resolve(context,"something") on MyBeanResolver during
evaluation
val bean = parser.parseExpression("@something").getValue(context)
```

To access a factory bean itself, you should instead prefix the bean name with an `&` symbol. The following example shows how to do so:

Java

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"&foo") on MyBeanResolver during
evaluation
Object bean = parser.parseExpression("&foo").getValue(context);
```

Kotlin

```
val parser = SpelExpressionParser()
val context = StandardEvaluationContext()
context.setBeanResolver(MyBeanResolver())

// This will end up calling resolve(context,"&foo") on MyBeanResolver during
evaluation
val bean = parser.parseExpression("&foo").getValue(context)
```

Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. The following listing shows a minimal example:

Java

```
String falseString = parser.parseExpression(
        "false ? 'trueExp' : 'falseExp'").getValue(String.class);
```

Kotlin

```
val falseString = parser.parseExpression(
        "false ? 'trueExp' : 'falseExp'").getValue(String::class.java)
```

In this case, the boolean `false` results in returning the string value `'falseExp'`. A more realistic example follows:

Java

```
parser.parseExpression("name").setValue(societyContext, "IEEE");
societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' " +
        "+ Name + ' Society' : #queryName + ' is not a member of the ' + Name + '
Society'";

String queryResultString = parser.parseExpression(expression)
        .getValue(societyContext, String.class);
// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

Kotlin

```
parser.parseExpression("name").setValue(societyContext, "IEEE")
societyContext.setVariable("queryName", "Nikola Tesla")

expression = "isMember(#queryName)? #queryName + ' is a member of the ' " + "+ Name +
' Society' : #queryName + ' is not a member of the ' + Name + ' Society'"

val queryResultString = parser.parseExpression(expression)
        .getValue(societyContext, String::class.java)
// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

See the next section on the Elvis operator for an even shorter syntax for the ternary operator.

The Elvis Operator

The Elvis operator is a shortening of the ternary operator syntax and is used in the Groovy language. With the ternary operator syntax, you usually have to repeat a variable twice, as the following example shows:

```
String name = "Elvis Presley";
String displayName = (name != null ? name : "Unknown");
```

Instead, you can use the Elvis operator (named for the resemblance to Elvis' hair style). The following example shows how to use the Elvis operator:

Java

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(new Inventor(),
String.class);
System.out.println(name);  // 'Unknown'
```

Kotlin

```
val parser = SpelExpressionParser()

val name = parser.parseExpression("name?:'Unknown'").getValue(Inventor(),
String::class.java)
println(name)  // 'Unknown'
```

The following listing shows a more complex example:

Java

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
String name = parser.parseExpression("name?:'Elvis Presley'").getValue(context, tesla,
String.class);
System.out.println(name);  // Nikola Tesla

tesla.setName(null);
name = parser.parseExpression("name?:'Elvis Presley'").getValue(context, tesla,
String.class);
System.out.println(name);  // Elvis Presley
```

Kotlin

```
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

val tesla = Inventor("Nikola Tesla", "Serbian")
var name = parser.parseExpression("name?:'Elvis Presley'").getValue(context, tesla,
String::class.java)
println(name)  // Nikola Tesla

tesla.setName(null)
name = parser.parseExpression("name?:'Elvis Presley'").getValue(context, tesla,
String::class.java)
println(name)  // Elvis Presley
```

You can use the Elvis operator to apply default values in expressions. The following example shows how to use the Elvis operator in a `@Value` expression:

```
@Value("#{systemProperties['pop3.port'] ?: 25}")
```

This will inject a system property `pop3.port` if it is defined or 25 if not.

Safe Navigation Operator

The safe navigation operator is used to avoid a `NullPointerException` and comes from the Groovy language. Typically, when you have a reference to an object, you might need to verify that it is not null before accessing methods or properties of the object. To avoid this, the safe navigation operator returns null instead of throwing an exception. The following example shows how to use the safe navigation operator:

Java

```
ExpressionParser parser = new SpelExpressionParser();
EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
tesla.setPlaceOfBirth(new PlaceOfBirth("Smiljan"));

String city = parser.parseExpression("placeOfBirth?.city").getValue(context, tesla,
String.class);
System.out.println(city);  // Smiljan

tesla.setPlaceOfBirth(null);
city = parser.parseExpression("placeOfBirth?.city").getValue(context, tesla,
String.class);
System.out.println(city);  // null - does not throw NullPointerException!!!
```

Kotlin

```
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

val tesla = Inventor("Nikola Tesla", "Serbian")
tesla.setPlaceOfBirth(PlaceOfBirth("Smiljan"))

var city = parser.parseExpression("placeOfBirth?.city").getValue(context, tesla,
String::class.java)
println(city)  // Smiljan

tesla.setPlaceOfBirth(null)
city = parser.parseExpression("placeOfBirth?.city").getValue(context, tesla,
String::class.java)
println(city)  // null - does not throw NullPointerException!!!
```

Collection Selection

Selection is a powerful expression language feature that lets you transform a source collection into another collection by selecting from its entries.

Selection uses a syntax of `.?[selectionExpression]`. It filters the collection and returns a new collection that contains a subset of the original elements. For example, selection lets us easily get a list of Serbian inventors, as the following example shows:

Java

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(
        "members.?[nationality == 'Serbian']").getValue(societyContext);
```

Kotlin

```
val list = parser.parseExpression(
        "members.?[nationality == 'Serbian']").getValue(societyContext) as
List<Inventor>
```

Selection is supported for arrays and anything that implements `java.lang.Iterable` or `java.util.Map`. For a list or array, the selection criteria is evaluated against each individual element. Against a map, the selection criteria is evaluated against each map entry (objects of the Java type `Map.Entry`). Each map entry has its `key` and `value` accessible as properties for use in the selection.

The following expression returns a new map that consists of those elements of the original map where the entry's value is less than 27:

Java

```
Map newMap = parser.parseExpression("map.?[value<27]").getValue();
```

Kotlin

```
val newMap = parser.parseExpression("map.?[value<27]").getValue()
```

In addition to returning all the selected elements, you can retrieve only the first or the last element. To obtain the first element matching the selection, the syntax is `.^[selectionExpression]`. To obtain the last matching selection, the syntax is `.$[selectionExpression]`.

Collection Projection

Projection lets a collection drive the evaluation of a sub-expression, and the result is a new collection. The syntax for projection is `.![projectionExpression]`. For example, suppose we have a list of inventors but want the list of cities where they were born. Effectively, we want to evaluate 'placeOfBirth.city' for every entry in the inventor list. The following example uses projection to do so:

Java

```
// returns ['Smiljan', 'Idvor' ]
List placesOfBirth = (List)parser.parseExpression("members.![placeOfBirth.city]");
```

Kotlin

```
// returns ['Smiljan', 'Idvor' ]
val placesOfBirth = parser.parseExpression("members.![placeOfBirth.city]") as List<*>
```

Projection is supported for arrays and anything that implements `java.lang.Iterable` or `java.util.Map`. When using a map to drive projection, the projection expression is evaluated against each entry in the map (represented as a Java `Map.Entry`). The result of a projection across a map is a list that consists of the evaluation of the projection expression against each map entry.

Expression templating

Expression templates allow mixing literal text with one or more evaluation blocks. Each evaluation block is delimited with prefix and suffix characters that you can define. A common choice is to use `#{ }` as the delimiters, as the following example shows:

Java

```
String randomPhrase = parser.parseExpression(
        "random number is #{T(java.lang.Math).random()}",
        new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

Kotlin

```
val randomPhrase = parser.parseExpression(
        "random number is #{T(java.lang.Math).random()}",
        TemplateParserContext()).getValue(String::class.java)

// evaluates to "random number is 0.7038186818312008"
```

The string is evaluated by concatenating the literal text `'random number is '` with the result of evaluating the expression inside the `#{ }` delimiter (in this case, the result of calling that `random()` method). The second argument to the `parseExpression()` method is of the type `ParserContext`. The `ParserContext` interface is used to influence how the expression is parsed in order to support the expression templating functionality. The definition of `TemplateParserContext` follows:

Java

```
public class TemplateParserContext implements ParserContext {

    public String getExpressionPrefix() {
        return "#{";
    }

    public String getExpressionSuffix() {
        return "}";
    }

    public boolean isTemplate() {
        return true;
    }
}
```

Kotlin

```
class TemplateParserContext : ParserContext {

    override fun getExpressionPrefix(): String {
        return "#{"
    }

    override fun getExpressionSuffix(): String {
        return "}"
    }

    override fun isTemplate(): Boolean {
        return true
    }
}
```

2.4.4. Classes Used in the Examples

This section lists the classes used in the examples throughout this chapter.

Inventor.Java

```
package org.spring.samples.spel.inventor;

import java.util.Date;
import java.util.GregorianCalendar;

public class Inventor {

    private String name;
    private String nationality;
    private String[] inventions;
```

```
    private Date birthdate;
    private PlaceOfBirth placeOfBirth;

    public Inventor(String name, String nationality) {
        GregorianCalendar c= new GregorianCalendar();
        this.name = name;
        this.nationality = nationality;
        this.birthdate = c.getTime();
    }

    public Inventor(String name, Date birthdate, String nationality) {
        this.name = name;
        this.nationality = nationality;
        this.birthdate = birthdate;
    }

    public Inventor() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

    public Date getBirthdate() {
        return birthdate;
    }

    public void setBirthdate(Date birthdate) {
        this.birthdate = birthdate;
    }

    public PlaceOfBirth getPlaceOfBirth() {
        return placeOfBirth;
    }

    public void setPlaceOfBirth(PlaceOfBirth placeOfBirth) {
        this.placeOfBirth = placeOfBirth;
    }
```

```
    public void setInventions(String[] inventions) {
        this.inventions = inventions;
    }

    public String[] getInventions() {
        return inventions;
    }
}
```

Inventor.kt

```
class Inventor(
    var name: String,
    var nationality: String,
    var inventions: Array<String>? = null,
    var birthdate: Date =  GregorianCalendar().time,
    var placeOfBirth: PlaceOfBirth? = null)
```

PlaceOfBirth.java

```
package org.spring.samples.spel.inventor;

public class PlaceOfBirth {

    private String city;
    private String country;

    public PlaceOfBirth(String city) {
        this.city=city;
    }

    public PlaceOfBirth(String city, String country) {
        this(city);
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String s) {
        this.city = s;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

PlaceOfBirth.kt

```
class PlaceOfBirth(var city: String, var country: String? = null) {
```

Society.java

```
package org.spring.samples.spel.inventor;

import java.util.*;

public class Society {

    private String name;

    public static String Advisors = "advisors";
    public static String President = "president";

    private List<Inventor> members = new ArrayList<Inventor>();
    private Map officers = new HashMap();

    public List getMembers() {
        return members;
    }

    public Map getOfficers() {
        return officers;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isMember(String name) {
        for (Inventor inventor : members) {
            if (inventor.getName().equals(name)) {
                return true;
            }
        }
        return false;
    }
}
```

Society.kt

```
package org.spring.samples.spel.inventor

import java.util.*

class Society {

    val Advisors = "advisors"
    val President = "president"

    var name: String? = null

    val members = ArrayList<Inventor>()
    val officers = mapOf<Any, Any>()

    fun isMember(name: String): Boolean {
        for (inventor in members) {
            if (inventor.name == name) {
                return true
            }
        }
        return false
    }
}
```

2.5. Aspect Oriented Programming with Spring

Aspect-oriented Programming (AOP) complements Object-oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns (such as transaction management) that cut across multiple types and objects. (Such concerns are often termed “crosscutting” concerns in AOP literature.)

One of the key components of Spring is the AOP framework. While the Spring IoC container does not depend on AOP (meaning you do not need to use AOP if you don’t want to), AOP complements Spring IoC to provide a very capable middleware solution.

Spring AOP with AspectJ pointcuts

Spring provides simple and powerful ways of writing custom aspects by using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language while still using Spring AOP for weaving.

This chapter discusses the schema- and @AspectJ-based AOP support. The lower-level AOP support is discussed in the following chapter.

AOP is used in the Spring Framework to:

- Provide declarative enterprise services. The most important such service is declarative transaction management.
- Let users implement custom aspects, complementing their use of OOP with AOP.

If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter.

2.5.1. AOP Concepts

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring used its own terminology.

- Aspect: A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented by using regular classes (the schema-based approach) or regular classes annotated with the `@Aspect` annotation (the @AspectJ style).
- Join point: A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.
- Advice: Action taken by an aspect at a particular join point. Different types of advice include “around”, “before” and “after” advice. (Advice types are discussed later.) Many AOP frameworks, including Spring, model an advice as an interceptor and maintain a chain of interceptors around the join point.
- Pointcut: A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.
- Introduction: Declaring additional methods or fields on behalf of a type. Spring AOP lets you introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)
- Target object: An object being advised by one or more aspects. Also referred to as the “advised object”. Since Spring AOP is implemented by using runtime proxies, this object is always a proxied object.
- AOP proxy: An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy is a JDK dynamic proxy or a CGLIB proxy.
- Weaving: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Spring AOP includes the following types of advice:

- Before advice: Advice that runs before a join point but that does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- After returning advice: Advice to be run after a join point completes normally (for example, if a method returns without throwing an exception).
- After throwing advice: Advice to be run if a method exits by throwing an exception.
- After (finally) advice: Advice to be run regardless of the means by which a join point exits (normal or exceptional return).
- Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the `proceed()` method on the `JoinPoint` used for around advice, and, hence, you cannot fail to invoke it.

All advice parameters are statically typed so that you work with advice parameters of the appropriate type (e.g. the type of the return value from a method execution) rather than `Object` arrays.

The concept of join points matched by pointcuts is the key to AOP, which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the object-oriented hierarchy. For example, you can apply an around advice providing declarative transaction management to a set of methods that span multiple objects (such as all business operations in the service layer).

2.5.2. Spring AOP Capabilities and Goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy and is thus suitable for use in a servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable). Rather, the aim is to provide a close integration between AOP implementation and Spring IoC, to help solve

common problems in enterprise applications.

Thus, for example, the Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured by using normal bean definition syntax (although this allows powerful "auto-proxying" capabilities). This is a crucial difference from other AOP implementations. You cannot do some things easily or efficiently with Spring AOP, such as advise very fine-grained objects (typically, domain objects). AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP never strives to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks such as Spring AOP and full-blown frameworks such as AspectJ are valuable and that they are complementary, rather than in competition. Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API. Spring AOP remains backward-compatible. See the following chapter for a discussion of the Spring AOP APIs.

One of the central tenets of the Spring Framework is that of non-invasiveness. This is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business or domain model. However, in some places, the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase. The rationale in giving you such options is because, in certain scenarios, it might be just plain easier to read or code some specific piece of functionality in such a way. However, the Spring Framework (almost) always offers you the choice: You have the freedom to make an informed decision as to which option best suits your particular use case or scenario.

One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ, Spring AOP, or both. You also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.

See Choosing which AOP Declaration Style to Use for a more complete discussion of the "whys and wherefores" of each style.

2.5.3. AOP Proxies

Spring AOP defaults to using standard JDK dynamic proxies for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. By default, CGLIB is used if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes, business classes normally implement one or more business interfaces. It is possible to force the use of CGLIB, in those (hopefully rare) cases where you need to advise a method that is not declared on an interface or where you need to pass a

proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is proxy-based. See Understanding AOP Proxies for a thorough examination of exactly what this implementation detail actually means.

2.5.4. @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations. The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release. Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP, though, and there is no dependency on the AspectJ compiler or weaver.

Using the AspectJ compiler and weaver enables use of the full AspectJ language and is discussed in Using AspectJ with Spring Applications.

Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration, you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects and auto-proxying beans based on whether or not they are advised by those aspects. By auto-proxying, we mean that, if Spring determines that a bean is advised by one or more aspects, it automatically generates a proxy for that bean to intercept method invocations and ensures that advice is run as needed.

The @AspectJ support can be enabled with XML- or Java-style configuration. In either case, you also need to ensure that AspectJ's `aspectjweaver.jar` library is on the classpath of your application (version 1.8 or later). This library is available in the `lib` directory of an AspectJ distribution or from the Maven Central repository.

Enabling @AspectJ Support with Java Configuration

To enable @AspectJ support with Java `@Configuration`, add the `@EnableAspectJAutoProxy` annotation, as the following example shows:

Java

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

}
```

Kotlin

```
@Configuration
@EnableAspectJAutoProxy
class AppConfig
```

Enabling @AspectJ Support with XML Configuration

To enable @AspectJ support with XML-based configuration, use the `aop:aspectj-autoproxy` element, as the following example shows:

```
<aop:aspectj-autoproxy/>
```

This assumes that you use schema support as described in XML Schema-based configuration. See the AOP schema for how to import the tags in the `aop` namespace.

Declaring an Aspect

With @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the `@Aspect` annotation) is automatically detected by Spring and used to configure Spring AOP. The next two examples show the minimal definition required for a not-very-useful aspect.

The first of the two example shows a regular bean definition in the application context that points to a bean class that has the `@Aspect` annotation:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of the aspect here -->
</bean>
```

The second of the two examples shows the `NotVeryUsefulAspect` class definition, which is annotated with the `org.aspectj.lang.annotation.Aspect` annotation;

Java

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Kotlin

```
package org.xyz

import org.aspectj.lang.annotation.Aspect;

@Aspect
class NotVeryUsefulAspect
```

Aspects (classes annotated with `@Aspect`) can have methods and fields, the same as any other class.

They can also contain pointcut, advice, and introduction (inter-type) declarations.

Autodetecting aspects through component scanning

You can register aspect classes as regular beans in your Spring XML configuration, via `@Bean` methods in `@Configuration` classes, or have Spring autodetect them through classpath scanning — the same as any other Spring-managed bean. However, note that the `@Aspect` annotation is not sufficient for autodetection in the classpath. For that purpose, you need to add a separate `@Component` annotation (or, alternatively, a custom stereotype annotation that qualifies, as per the rules of Spring's component scanner).

Advising aspects with other aspects?

In Spring AOP, aspects themselves cannot be the targets of advice from other aspects. The `@Aspect` annotation on a class marks it as an aspect and, hence, excludes it from auto-proxying.

Declaring a Pointcut

Pointcuts determine join points of interest and thus enable us to control when advice runs. Spring AOP only supports method execution join points for Spring beans, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters and a pointcut expression that determines exactly which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated by using the `@Pointcut` annotation (the method serving as the pointcut signature must have a `void` return type).

An example may help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named `anyOldTransfer` that matches the execution of any method named `transfer`:

Java

```
@Pointcut("execution(* transfer(..))") // the pointcut expression
private void anyOldTransfer() {} // the pointcut signature
```

Kotlin

```
@Pointcut("execution(* transfer(..))") // the pointcut expression
private fun anyOldTransfer() {} // the pointcut signature
```

The pointcut expression that forms the value of the `@Pointcut` annotation is a regular AspectJ pointcut expression. For a full discussion of AspectJ's pointcut language, see the AspectJ Programming Guide (and, for extensions, the AspectJ 5 Developer's Notebook) or one of the books on AspectJ (such as *Eclipse AspectJ*, by Colyer et al., or *AspectJ in Action*, by Ramnivas Laddad).

Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

- `execution`: For matching method execution join points. This is the primary pointcut designator to use when working with Spring AOP.
- `within`: Limits matching to join points within certain types (the execution of a method declared within a matching type when using Spring AOP).
- `this`: Limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type.
- `target`: Limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type.
- `args`: Limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types.
- `@target`: Limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type.
- `@args`: Limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given types.
- `@within`: Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP).
- `@annotation`: Limits matching to join points where the subject of the join point (the method being run in Spring AOP) has the given annotation.

> **Other pointcut types**
>
> The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring: `call`, `get`, `set`, `preinitialization`, `staticinitialization`, `initialization`, `handler`, `adviceexecution`, `withincode`, `cflow`, `cflowbelow`, `if`, `@this`, and `@withincode`. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP results in an `IllegalArgumentException` being thrown.
>
> The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

Because Spring AOP limits matching to only method execution join points, the preceding discussion of the pointcut designators gives a narrower definition than you can find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and, at an execution join point, both `this` and `target` refer to the same object: the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (which is bound to `this`) and the target object behind the proxy (which is bound to `target`).

Due to the proxy-based nature of Spring's AOP framework, calls within the target object are, by definition, not intercepted. For JDK proxies, only public interface method calls on the proxy can be intercepted. With CGLIB, public and protected method calls on the proxy are intercepted (and even package-visible methods, if necessary). However, common interactions through proxies should always be designed through public signatures.

Note that pointcut definitions are generally matched against any intercepted method. If a pointcut is strictly meant to be public-only, even in a CGLIB proxy scenario with potential non-public interactions through proxies, it needs to be defined accordingly.

If your interception needs include method calls or even constructors within the target class, consider the use of Spring-driven native AspectJ weaving instead of Spring's proxy-based AOP framework. This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving before making a decision.

Spring AOP also supports an additional PCD named `bean`. This PCD lets you limit the matching of join points to a particular named Spring bean or to a set of named Spring beans (when using wildcards). The `bean` PCD has the following form:

Java

```
bean(idOrNameOfBean)
```

Kotlin

```
bean(idOrNameOfBean)
```

The `idOrNameOfBean` token can be the name of any Spring bean. Limited wildcard support that uses the `*` character is provided, so, if you establish some naming conventions for your Spring beans, you can write a `bean` PCD expression to select them. As is the case with other pointcut designators, the `bean` PCD can be used with the `&&` (and), `||` (or), and `!` (negation) operators, too.

The `bean` PCD is supported only in Spring AOP and not in native AspectJ weaving. It is a Spring-specific extension to the standard PCDs that AspectJ defines and is, therefore, not available for aspects declared in the `@Aspect` model.

The `bean` PCD operates at the instance level (building on the Spring bean name concept) rather than at the type level only (to which weaving-based AOP is limited). Instance-based pointcut designators are a special capability of Spring's proxy-based AOP framework and its close integration with the Spring bean factory, where it is natural and straightforward to identify specific beans by name.

Combining Pointcut Expressions

You can combine pointcut expressions by using `&&`, `||` and `!`. You can also refer to pointcut

expressions by name. The following example shows three pointcut expressions:

Java

```
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {} ①

@Pointcut("within(com.xyz.myapp.trading..*)")
private void inTrading() {} ②

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {} ③
```

① `anyPublicOperation` matches if a method execution join point represents the execution of any public method.

② `inTrading` matches if a method execution is in the trading module.

③ `tradingOperation` matches if a method execution represents any public method in the trading module.

Kotlin

```
@Pointcut("execution(public * *(..))")
private fun anyPublicOperation() {} ①

@Pointcut("within(com.xyz.myapp.trading..*)")
private fun inTrading() {} ②

@Pointcut("anyPublicOperation() && inTrading()")
private fun tradingOperation() {} ③
```

① `anyPublicOperation` matches if a method execution join point represents the execution of any public method.

② `inTrading` matches if a method execution is in the trading module.

③ `tradingOperation` matches if a method execution represents any public method in the trading module.

It is a best practice to build more complex pointcut expressions out of smaller named components, as shown earlier. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere, and so on). Visibility does not affect pointcut matching.

Sharing Common Pointcut Definitions

When working with enterprise applications, developers often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a `CommonPointcuts` aspect that captures common pointcut expressions for this purpose. Such an aspect typically resembles the following example:

Java

```
package com.xyz.myapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CommonPointcuts {

    /**
     * A join point is in the web layer if the method is defined
     * in a type in the com.xyz.myapp.web package or any sub-package
     * under that.
     */
    @Pointcut("within(com.xyz.myapp.web..*)")
    public void inWebLayer() {}

    /**
     * A join point is in the service layer if the method is defined
     * in a type in the com.xyz.myapp.service package or any sub-package
     * under that.
     */
    @Pointcut("within(com.xyz.myapp.service..*)")
    public void inServiceLayer() {}

    /**
     * A join point is in the data access layer if the method is defined
     * in a type in the com.xyz.myapp.dao package or any sub-package
     * under that.
     */
    @Pointcut("within(com.xyz.myapp.dao..*)")
    public void inDataAccessLayer() {}

    /**
     * A business service is the execution of any method defined on a service
     * interface. This definition assumes that interfaces are placed in the
     * "service" package, and that implementation types are in sub-packages.
     *
     * If you group service interfaces by functional area (for example,
     * in packages com.xyz.myapp.abc.service and com.xyz.myapp.def.service) then
     * the pointcut expression "execution(* com.xyz.myapp..service.*.*(..))"
     * could be used instead.
     *
     * Alternatively, you can write the expression using the 'bean'
     * PCD, like so "bean(*Service)". (This assumes that you have
     * named your Spring service beans in a consistent fashion.)
     */
    @Pointcut("execution(* com.xyz.myapp..service.*.*(..))")
    public void businessService() {}

    /**
```

```
     * A data access operation is the execution of any method defined on a
     * dao interface. This definition assumes that interfaces are placed in the
     * "dao" package, and that implementation types are in sub-packages.
     */
    @Pointcut("execution(* com.xyz.myapp.dao.*.*(..))")
    public void dataAccessOperation() {}

}
```

Kotlin

```
package com.xyz.myapp

import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Pointcut

@Aspect
class CommonPointcuts {

    /**
    * A join point is in the web layer if the method is defined
    * in a type in the com.xyz.myapp.web package or any sub-package
    * under that.
    */
    @Pointcut("within(com.xyz.myapp.web..*)")
    fun inWebLayer() {
    }

    /**
    * A join point is in the service layer if the method is defined
    * in a type in the com.xyz.myapp.service package or any sub-package
    * under that.
    */
    @Pointcut("within(com.xyz.myapp.service..*)")
    fun inServiceLayer() {
    }

    /**
    * A join point is in the data access layer if the method is defined
    * in a type in the com.xyz.myapp.dao package or any sub-package
    * under that.
    */
    @Pointcut("within(com.xyz.myapp.dao..*)")
    fun inDataAccessLayer() {
    }

    /**
    * A business service is the execution of any method defined on a service
    * interface. This definition assumes that interfaces are placed in the
    * "service" package, and that implementation types are in sub-packages.
    *
```

```
    * If you group service interfaces by functional area (for example,
    * in packages com.xyz.myapp.abc.service and com.xyz.myapp.def.service) then
    * the pointcut expression "execution(* com.xyz.myapp..service.*.*(..))"
    * could be used instead.
    *
    * Alternatively, you can write the expression using the 'bean'
    * PCD, like so "bean(*Service)". (This assumes that you have
    * named your Spring service beans in a consistent fashion.)
    */
    @Pointcut("execution(* com.xyz.myapp..service.*.*(..))")
    fun businessService() {
    }

    /**
    * A data access operation is the execution of any method defined on a
    * dao interface. This definition assumes that interfaces are placed in the
    * "dao" package, and that implementation types are in sub-packages.
    */
    @Pointcut("execution(* com.xyz.myapp.dao.*.*(..))")
    fun dataAccessOperation() {
    }

}
```

You can refer to the pointcuts defined in such an aspect anywhere you need a pointcut expression. For example, to make the service layer transactional, you could write the following:

```
<aop:config>
    <aop:advisor
        pointcut="com.xyz.myapp.CommonPointcuts.businessService()"
        advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

The `<aop:config>` and `<aop:advisor>` elements are discussed in Schema-based AOP Support. The transaction elements are discussed in Transaction Management.

Examples

Spring AOP users are likely to use the `execution` pointcut designator the most often. The format of an execution expression follows:

```
    execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-
pattern(param-pattern)
                throws-pattern?)
```

All parts except the returning type pattern (`ret-type-pattern` in the preceding snippet), the name pattern, and the parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. `*` is most frequently used as the returning type pattern. It matches any return type. A fully-qualified type name matches only when the method returns the given type. The name pattern matches the method name. You can use the `*` wildcard as all or part of a name pattern. If you specify a declaring type pattern, include a trailing `.` to join it to the name pattern component. The parameters pattern is slightly more complex: `()` matches a method that takes no parameters, whereas `(..)` matches any number (zero or more) of parameters. The `(*)` pattern matches a method that takes one parameter of any type. `(*,String)` matches a method that takes two parameters. The first can be of any type, while the second must be a `String`. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

The following examples show some common pointcut expressions:

- The execution of any public method:

```
    execution(public * *(..))
```

- The execution of any method with a name that begins with `set`:

```
    execution(* set*(..))
```

- The execution of any method defined by the `AccountService` interface:

```
    execution(* com.xyz.service.AccountService.*(..))
```

- The execution of any method defined in the `service` package:

```
    execution(* com.xyz.service.*.*(..))
```

- The execution of any method defined in the service package or one of its sub-packages:

```
    execution(* com.xyz.service..*.*(..))
```

- Any join point (method execution only in Spring AOP) within the service package:

```
    within(com.xyz.service.*)
```

- Any join point (method execution only in Spring AOP) within the service package or one of its sub-packages:

```
within(com.xyz.service..*)
```

- Any join point (method execution only in Spring AOP) where the proxy implements the `AccountService` interface:

```
this(com.xyz.service.AccountService)
```

`this` is more commonly used in a binding form. See the section on Declaring Advice for how to make the proxy object available in the advice body.

- Any join point (method execution only in Spring AOP) where the target object implements the `AccountService` interface:

```
target(com.xyz.service.AccountService)
```

`target` is more commonly used in a binding form. See the Declaring Advice section for how to make the target object available in the advice body.

- Any join point (method execution only in Spring AOP) that takes a single parameter and where the argument passed at runtime is `Serializable`:

```
args(java.io.Serializable)
```

`args` is more commonly used in a binding form. See the Declaring Advice section for how to make the method arguments available in the advice body.

Note that the pointcut given in this example is different from `execution(* *(java.io.Serializable))`. The args version matches if the argument passed at runtime is `Serializable`, and the execution version matches if the method signature declares a single parameter of type `Serializable`.

- Any join point (method execution only in Spring AOP) where the target object has a `@Transactional` annotation:

```
@target(org.springframework.transaction.annotation.Transactional)
```

You can also use `@target` in a binding form. See the Declaring Advice section for how to make the annotation object available in the advice body.

- Any join point (method execution only in Spring AOP) where the declared type of the target object has an `@Transactional` annotation:

```
@within(org.springframework.transaction.annotation.Transactional)
```

You can also use `@within` in a binding form. See the Declaring Advice section for how to make the annotation object available in the advice body.

- Any join point (method execution only in Spring AOP) where the executing method has an `@Transactional` annotation:

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

You can also use `@annotation` in a binding form. See the Declaring Advice section for how to make the annotation object available in the advice body.

- Any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the `@Classified` annotation:

```
@args(com.xyz.security.Classified)
```

You can also use `@args` in a binding form. See the Declaring Advice section how to make the annotation object(s) available in the advice body.

- Any join point (method execution only in Spring AOP) on a Spring bean named `tradeService`:

```
bean(tradeService)
```

- Any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression `*Service`:

```
bean(*Service)
```

Writing Good Pointcuts

During compilation, AspectJ processes pointcuts in order to optimize matching performance. Examining code and determining if each join point matches (statically or dynamically) a given pointcut is a costly process. (A dynamic match means the match cannot be fully determined from static analysis and that a test is placed in the code to determine if there is an actual match when the code is running). On first encountering a pointcut declaration, AspectJ rewrites it into an optimal form for the matching process. What does this mean? Basically, pointcuts are rewritten in DNF (Disjunctive Normal Form) and the components of the pointcut are sorted such that those

components that are cheaper to evaluate are checked first. This means you do not have to worry about understanding the performance of various pointcut designators and may supply them in any order in a pointcut declaration.

However, AspectJ can work only with what it is told. For optimal performance of matching, you should think about what they are trying to achieve and narrow the search space for matches as much as possible in the definition. The existing designators naturally fall into one of three groups: kinded, scoping, and contextual:

- Kinded designators select a particular kind of join point: `execution`, `get`, `set`, `call`, and `handler`.
- Scoping designators select a group of join points of interest (probably of many kinds): `within` and `withincode`
- Contextual designators match (and optionally bind) based on context: `this`, `target`, and `@annotation`

A well written pointcut should include at least the first two types (kinded and scoping). You can include the contextual designators to match based on join point context or bind that context for use in the advice. Supplying only a kinded designator or only a contextual designator works but could affect weaving performance (time and memory used), due to extra processing and analysis. Scoping designators are very fast to match, and using them means AspectJ can very quickly dismiss groups of join points that should not be further processed. A good pointcut should always include one if possible.

Declaring Advice

Advice is associated with a pointcut expression and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut or a pointcut expression declared in place.

Before Advice

You can declare before advice in an aspect by using the `@Before` annotation:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Before

@Aspect
class BeforeExample {

    @Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doAccessCheck() {
        // ...
    }
}
```

If we use an in-place pointcut expression, we could rewrite the preceding example as the following example:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    public void doAccessCheck() {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Before

@Aspect
class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    fun doAccessCheck() {
        // ...
    }
}
```

After Returning Advice

After returning advice runs when a matched method execution returns normally. You can declare it by using the `@AfterReturning` annotation:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterReturning

@Aspect
class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doAccessCheck() {
        // ...
    }
}
```

You can have multiple advice declarations (and other members as well), all inside the same aspect. We show only a single advice declaration in these examples to focus the effect of each one.

Sometimes, you need access in the advice body to the actual value that was returned. You can use the form of `@AfterReturning` that binds the return value to get that access, as the following example shows:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning(
        pointcut="com.xyz.myapp.CommonPointcuts.dataAccessOperation()",
        returning="retVal")
    public void doAccessCheck(Object retVal) {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterReturning

@Aspect
class AfterReturningExample {

    @AfterReturning(
        pointcut = "com.xyz.myapp.CommonPointcuts.dataAccessOperation()",
        returning = "retVal")
    fun doAccessCheck(retVal: Any) {
        // ...
    }
}
```

The name used in the `returning` attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value is passed to the advice method as the corresponding argument value. A `returning` clause also restricts matching to only those method executions that return a value of the specified type (in this case, `Object`, which matches any return value).

Please note that it is not possible to return a totally different reference when using after returning advice.

After Throwing Advice

After throwing advice runs when a matched method execution exits by throwing an exception. You can declare it by using the `@AfterThrowing` annotation, as the following example shows:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doRecoveryActions() {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterThrowing

@Aspect
class AfterThrowingExample {

    @AfterThrowing("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doRecoveryActions() {
        // ...
    }
}
```

Often, you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. You can use the `throwing` attribute to both restrict matching (if desired — use `Throwable` as the exception type otherwise) and bind the thrown exception to an advice parameter. The following example shows how to do so:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.xyz.myapp.CommonPointcuts.dataAccessOperation()",
        throwing="ex")
    public void doRecoveryActions(DataAccessException ex) {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterThrowing

@Aspect
class AfterThrowingExample {

    @AfterThrowing(
        pointcut = "com.xyz.myapp.CommonPointcuts.dataAccessOperation()",
        throwing = "ex")
    fun doRecoveryActions(ex: DataAccessException) {
        // ...
    }
}
```

The name used in the `throwing` attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception is passed to the advice method as the corresponding argument value. A `throwing` clause also restricts matching to only those method executions that throw an exception of the specified type (`DataAccessException`, in this case).

Note that `@AfterThrowing` does not indicate a general exception handling callback. Specifically, an `@AfterThrowing` advice method is only supposed to receive exceptions from the join point (user-declared target method) itself but not from an accompanying `@After`/`@AfterReturning` method.

After (Finally) Advice

After (finally) advice runs when a matched method execution exits. It is declared by using the `@After` annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources and similar purposes. The following example shows how to use after finally advice:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

    @After("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doReleaseLock() {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.After

@Aspect
class AfterFinallyExample {

    @After("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doReleaseLock() {
        // ...
    }
}
```

Note that `@After` advice in AspectJ is defined as "after finally advice", analogous to a finally block in a try-catch statement. It will be invoked for any outcome, normal return or exception thrown from the join point (user-declared target method), in contrast to `@AfterReturning` which only applies to successful normal returns.

Around Advice

The last kind of advice is *around* advice. Around advice runs "around" a matched method's execution. It has the opportunity to do work both before and after the method runs and to determine when, how, and even if the method actually gets to run at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner – for example, starting and stopping a timer.

Always use the least powerful form of advice that meets your requirements.

For example, do not use *around* advice if *before* advice is sufficient for your needs.

Around advice is declared by annotating a method with the `@Around` annotation. The method should declare `Object` as its return type, and the first parameter of the method must be of type `ProceedingJoinPoint`. Within the body of the advice method, you must invoke `proceed()` on the `ProceedingJoinPoint` in order for the underlying method to run. Invoking `proceed()` without arguments will result in the caller's original arguments being supplied to the underlying method when it is invoked. For advanced use cases, there is an overloaded variant of the `proceed()` method which accepts an array of arguments (`Object[]`). The values in the array will be used as the arguments to the underlying method when it is invoked.

The behavior of `proceed` when called with an `Object[]` is a little different than the behavior of `proceed` for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to `proceed` must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to (do not worry if this does not make sense right now).

The approach taken by Spring is simpler and a better match to its proxy-based, execution-only semantics. You only need to be aware of this difference if you compile `@AspectJ` aspects written for Spring and use `proceed` with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

The value returned by the around advice is the return value seen by the caller of the method. For example, a simple caching aspect could return a value from a cache if it has one or invoke `proceed()` (and return that value) if it does not. Note that `proceed` may be invoked once, many times, or not at all within the body of the around advice. All of these are legal.

If you declare the return type of your around advice method as `void`, `null` will always be returned to the caller, effectively ignoring the result of any invocation of `proceed()`. It is therefore recommended that an around advice method declare a return type of `Object`. The advice method should typically return the value returned from an invocation of `proceed()`, even if the underlying method has a `void` return type. However, the advice may optionally return a cached value, a wrapped value, or some other value depending on the use case.

The following example shows how to use around advice:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Around
import org.aspectj.lang.ProceedingJoinPoint

@Aspect
class AroundExample {

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    fun doBasicProfiling(pjp: ProceedingJoinPoint): Any {
        // start stopwatch
        val retVal = pjp.proceed()
        // stop stopwatch
        return retVal
    }
}
```

Advice Parameters

Spring offers fully typed advice, meaning that you declare the parameters you need in the advice signature (as we saw earlier for the returning and throwing examples) rather than work with `Object[]` arrays all the time. We see how to make argument and other contextual values available to the advice body later in this section. First, we take a look at how to write generic advice that can find out about the method the advice is currently advising.

Access to the Current `JoinPoint`

Any advice method may declare, as its first parameter, a parameter of type `org.aspectj.lang.JoinPoint`. Note that around advice is required to declare a first parameter of type `ProceedingJoinPoint`, which is a subclass of `JoinPoint`.

The `JoinPoint` interface provides a number of useful methods:

- `getArgs()`: Returns the method arguments.
- `getThis()`: Returns the proxy object.
- `getTarget()`: Returns the target object.
- `getSignature()`: Returns a description of the method that is being advised.
- `toString()`: Prints a useful description of the method being advised.

See the javadoc for more detail.

Passing Parameters to Advice

We have already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of `args`. If you use a parameter name in place of a type name in an `args` expression, the value of the corresponding argument is passed as the parameter value when the advice is

invoked. An example should make this clearer. Suppose you want to advise the execution of DAO operations that take an `Account` object as the first parameter, and you need access to the account in the advice body. You could write the following:

Java

```
@Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,..)")
public void validateAccount(Account account) {
    // ...
}
```

Kotlin

```
@Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,..)")
fun validateAccount(account: Account) {
    // ...
}
```

The `args(account,..)` part of the pointcut expression serves two purposes. First, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of `Account`. Second, it makes the actual `Account` object available to the advice through the `account` parameter.

Another way of writing this is to declare a pointcut that "provides" the `Account` object value when it matches a join point, and then refer to the named pointcut from the advice. This would look as follows:

Java

```
@Pointcut("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
    // ...
}
```

Kotlin

```
@Pointcut("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,..)")
private fun accountDataAccessOperation(account: Account) {
}

@Before("accountDataAccessOperation(account)")
fun validateAccount(account: Account) {
    // ...
}
```

See the AspectJ programming guide for more details.

The proxy object (`this`), target object (`target`), and annotations (`@within`, `@target`, `@annotation`, and `@args`) can all be bound in a similar fashion. The next two examples show how to match the execution of methods annotated with an `@Auditable` annotation and extract the audit code:

The first of the two examples shows the definition of the `@Auditable` annotation:

Java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
    AuditCode value();
}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
@Target(AnnotationTarget.FUNCTION)
annotation class Auditable(val value: AuditCode)
```

The second of the two examples shows the advice that matches the execution of `@Auditable` methods:

Java

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")
public void audit(Auditable auditable) {
    AuditCode code = auditable.value();
    // ...
}
```

Kotlin

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")
fun audit(auditable: Auditable) {
    val code = auditable.value()
    // ...
}
```

Advice Parameters and Generics

Spring AOP can handle generics used in class declarations and method parameters. Suppose you have a generic type like the following:

Java

```
public interface Sample<T> {
    void sampleGenericMethod(T param);
    void sampleGenericCollectionMethod(Collection<T> param);
}
```

Kotlin

```
interface Sample<T> {
    fun sampleGenericMethod(param: T)
    fun sampleGenericCollectionMethod(param: Collection<T>)
}
```

You can restrict interception of method types to certain parameter types by tying the advice parameter to the parameter type for which you want to intercept the method:

Java

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
public void beforeSampleMethod(MyType param) {
    // Advice implementation
}
```

Kotlin

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")
fun beforeSampleMethod(param: MyType) {
    // Advice implementation
}
```

This approach does not work for generic collections. So you cannot define a pointcut as follows:

Java

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
public void beforeSampleMethod(Collection<MyType> param) {
    // Advice implementation
}
```

Kotlin

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")
fun beforeSampleMethod(param: Collection<MyType>) {
    // Advice implementation
}
```

To make this work, we would have to inspect every element of the collection, which is not reasonable, as we also cannot decide how to treat `null` values in general. To achieve something similar to this, you have to type the parameter to `Collection<?>` and manually check the type of the elements.

Determining Argument Names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in advice and pointcut method signatures. Parameter names are not available through Java reflection, so Spring AOP uses the following strategy to determine parameter names:

- If the parameter names have been explicitly specified by the user, the specified parameter names are used. Both the advice and the pointcut annotations have an optional `argNames` attribute that you can use to specify the argument names of the annotated method. These argument names are available at runtime. The following example shows how to use the `argNames` attribute:

Java

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) &&
@annotation(auditable)",
        argNames="bean,auditable")
public void audit(Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code and bean
}
```

Kotlin

```
@Before(value = "com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) &&
@annotation(auditable)", argNames = "bean,auditable")
fun audit(bean: Any, auditable: Auditable) {
    val code = auditable.value()
    // ... use code and bean
}
```

If the first parameter is of the `JoinPoint`, `ProceedingJoinPoint`, or `JoinPoint.StaticPart` type, you can leave out the name of the parameter from the value of the `argNames` attribute. For example, if you modify the preceding advice to receive the join point object, the `argNames` attribute need not include it:

Java

```
@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) &&
@annotation(auditable)",
        argNames="bean,auditable")
public void audit(JoinPoint jp, Object bean, Auditable auditable) {
    AuditCode code = auditable.value();
    // ... use code, bean, and jp
}
```

Kotlin

```
@Before(value = "com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) &&
@annotation(auditable)", argNames = "bean,auditable")
fun audit(jp: JoinPoint, bean: Any, auditable: Auditable) {
    val code = auditable.value()
    // ... use code, bean, and jp
}
```

The special treatment given to the first parameter of the `JoinPoint`, `ProceedingJoinPoint`, and `JoinPoint.StaticPart` types is particularly convenient for advice instances that do not collect any other join point context. In such situations, you may omit the `argNames` attribute. For example, the following advice need not declare the `argNames` attribute:

Java

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")
public void audit(JoinPoint jp) {
    // ... use jp
}
```

Kotlin

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")
fun audit(jp: JoinPoint) {
    // ... use jp
}
```

- Using the `argNames` attribute is a little clumsy, so if the `argNames` attribute has not been specified, Spring AOP looks at the debug information for the class and tries to determine the parameter names from the local variable table. This information is present as long as the classes have been compiled with debug information (`-g:vars` at a minimum). The consequences of compiling with this flag on are: (1) your code is slightly easier to understand (reverse engineer), (2) the class file sizes are very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables is not applied by your compiler. In other words, you should encounter no difficulties by building with this flag on.

If an @AspectJ aspect has been compiled by the AspectJ compiler (`ajc`) even without the debug information, you need not add the `argNames` attribute, as the compiler retain the needed information.

- If the code has been compiled without the necessary debug information, Spring AOP tries to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method takes only one parameter, the pairing is obvious). If the binding of variables is ambiguous given the available information, an `AmbiguousBindingException` is thrown.
- If all of the above strategies fail, an `IllegalArgumentException` is thrown.

Proceeding with Arguments

We remarked earlier that we would describe how to write a `proceed` call with arguments that works consistently across Spring AOP and AspectJ. The solution is to ensure that the advice signature binds each of the method parameters in order. The following example shows how to do so:

Java

```
@Around("execution(List<Account> find*(..)) && " +
        "com.xyz.myapp.CommonPointcuts.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp,
        String accountHolderNamePattern) throws Throwable {
    String newPattern = preProcess(accountHolderNamePattern);
    return pjp.proceed(new Object[] {newPattern});
}
```

Kotlin

```
@Around("execution(List<Account> find*(..)) && " +
        "com.xyz.myapp.CommonPointcuts.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
fun preProcessQueryPattern(pjp: ProceedingJoinPoint,
                           accountHolderNamePattern: String): Any {
    val newPattern = preProcess(accountHolderNamePattern)
    return pjp.proceed(arrayOf<Any>(newPattern))
}
```

In many cases, you do this binding anyway (as in the preceding example).

Advice Ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so, given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so, given two pieces of after advice, the one with the highest precedence will run

second).

When two pieces of advice defined in different aspects both need to run at the same join point, unless you specify otherwise, the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the `org.springframework.core.Ordered` interface in the aspect class or annotating it with the `@Order` annotation. Given two aspects, the aspect returning the lower value from `Ordered.getOrder()` (or the annotation value) has the higher precedence.

Each of the distinct advice types of a particular aspect is conceptually meant to apply to the join point directly. As a consequence, an `@AfterThrowing` advice method is not supposed to receive an exception from an accompanying `@After` /`@AfterReturning` method.

As of Spring Framework 5.2.7, advice methods defined in the same `@Aspect` class that need to run at the same join point are assigned precedence based on their advice type in the following order, from highest to lowest precedence: `@Around`, `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`. Note, however, that an `@After` advice method will effectively be invoked after any `@AfterReturning` or `@AfterThrowing` advice methods in the same aspect, following AspectJ's "after finally advice" semantics for `@After`.

When two pieces of the same type of advice (for example, two `@After` advice methods) defined in the same `@Aspect` class both need to run at the same join point, the ordering is undefined (since there is no way to retrieve the source code declaration order through reflection for javac-compiled classes). Consider collapsing such advice methods into one advice method per join point in each `@Aspect` class or refactor the pieces of advice into separate `@Aspect` classes that you can order at the aspect level via `Ordered` or `@Order`.

Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

You can make an introduction by using the `@DeclareParents` annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface named `UsageTracked` and an implementation of that interface named `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface (e.g. for statistics via JMX):

Java

```
@Aspect
public class UsageTracking {

    @DeclareParents(value="com.xzy.myapp.service.*+",
defaultImpl=DefaultUsageTracked.class)
    public static UsageTracked mixin;

    @Before("com.xyz.myapp.CommonPointcuts.businessService() && this(usageTracked)")
    public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
    }

}
```

Kotlin

```
@Aspect
class UsageTracking {

    companion object {
        @DeclareParents(value = "com.xzy.myapp.service.*+", defaultImpl =
DefaultUsageTracked::class)
        lateinit var mixin: UsageTracked
    }

    @Before("com.xyz.myapp.CommonPointcuts.businessService() && this(usageTracked)")
    fun recordUsage(usageTracked: UsageTracked) {
        usageTracked.incrementUseCount()
    }
}
```

The interface to be implemented is determined by the type of the annotated field. The `value` attribute of the `@DeclareParents` annotation is an AspectJ type pattern. Any bean of a matching type implements the `UsageTracked` interface. Note that, in the before advice of the preceding example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically, you would write the following:

Java

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

Kotlin

```
val usageTracked = context.getBean("myService") as UsageTracked
```

Aspect Instantiation Models

This is an advanced topic. If you are just starting out with AOP, you can safely skip it until later.

By default, there is a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles. Spring supports AspectJ's `perthis` and `pertarget` instantiation models; `percflow`, `percflowbelow`, and `pertypewithin` are not currently supported.

You can declare a `perthis` aspect by specifying a `perthis` clause in the `@Aspect` annotation. Consider the following example:

Java

```
@Aspect("perthis(com.xyz.myapp.CommonPointcuts.businessService())")
public class MyAspect {

    private int someState;

    @Before("com.xyz.myapp.CommonPointcuts.businessService()")
    public void recordServiceUsage() {
        // ...
    }
}
```

Kotlin

```
@Aspect("perthis(com.xyz.myapp.CommonPointcuts.businessService())")
class MyAspect {

    private val someState: Int = 0

    @Before("com.xyz.myapp.CommonPointcuts.businessService()")
    fun recordServiceUsage() {
        // ...
    }
}
```

In the preceding example, the effect of the `perthis` clause is that one aspect instance is created for each unique service object that performs a business service (each unique object bound to `this` at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it runs. As soon as the aspect instance has been created, the advice declared within it runs at matched join points, but only when the service object is the one with which this aspect is associated. See the AspectJ Programming Guide for more information on `per` clauses.

The `pertarget` instantiation model works in exactly the same way as `perthis`, but it creates one

aspect instance for each unique target object at matched join points.

An AOP Example

Now that you have seen how all the constituent parts work, we can put them together to do something useful.

The execution of business services can sometimes fail due to concurrency issues (for example, a deadlock loser). If the operation is retried, it is likely to succeed on the next try. For business services where it is appropriate to retry in such conditions (idempotent operations that do not need to go back to the user for conflict resolution), we want to transparently retry the operation to avoid the client seeing a `PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer and, hence, is ideal for implementing through an aspect.

Because we want to retry the operation, we need to use around advice so that we can call `proceed` multiple times. The following listing shows the basic aspect implementation:

Java

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

    private static final int DEFAULT_MAX_RETRIES = 2;

    private int maxRetries = DEFAULT_MAX_RETRIES;
    private int order = 1;

    public void setMaxRetries(int maxRetries) {
        this.maxRetries = maxRetries;
    }

    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        PessimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(PessimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        } while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }
}
```

Kotlin

```
@Aspect
class ConcurrentOperationExecutor : Ordered {

    private val DEFAULT_MAX_RETRIES = 2
    private var maxRetries = DEFAULT_MAX_RETRIES
    private var order = 1

    fun setMaxRetries(maxRetries: Int) {
        this.maxRetries = maxRetries
    }

    override fun getOrder(): Int {
        return this.order
    }

    fun setOrder(order: Int) {
        this.order = order
    }

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    fun doConcurrentOperation(pjp: ProceedingJoinPoint): Any {
        var numAttempts = 0
        var lockFailureException: PessimisticLockingFailureException
        do {
            numAttempts++
            try {
                return pjp.proceed()
            } catch (ex: PessimisticLockingFailureException) {
                lockFailureException = ex
            }

        } while (numAttempts <= this.maxRetries)
        throw lockFailureException
    }
}
```

Note that the aspect implements the `Ordered` interface so that we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties are both configured by Spring. The main action happens in the `doConcurrentOperation` around advice. Notice that, for the moment, we apply the retry logic to each `businessService()`. We try to proceed, and if we fail with a `PessimisticLockingFailureException`, we try again, unless we have exhausted all of our retry attempts.

The corresponding Spring configuration follows:

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

To refine the aspect so that it retries only idempotent operations, we might define the following `Idempotent` annotation:

Java

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
annotation class Idempotent// marker annotation
```

We can then use the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations involves refining the pointcut expression so that only `@Idempotent` operations match, as follows:

Java

```
@Around("com.xyz.myapp.CommonPointcuts.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    // ...
}
```

Kotlin

```
@Around("com.xyz.myapp.CommonPointcuts.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
fun doConcurrentOperation(pjp: ProceedingJoinPoint): Any {
    // ...
}
```

2.5.5. Schema-based AOP Support

If you prefer an XML-based format, Spring also offers support for defining aspects using the `aop` namespace tags. The exact same pointcut expressions and advice kinds as when using the @AspectJ

style are supported. Hence, in this section we focus on that syntax and refer the reader to the discussion in the previous section (@AspectJ support) for an understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the `spring-aop` schema, as described in XML Schema-based configuration. See the AOP schema for how to import the tags in the `aop` namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an `<aop:config>` element (you can have more than one `<aop:config>` element in an application context configuration). An `<aop:config>` element can contain pointcut, advisor, and aspect elements (note that these must be declared in that order).

The `<aop:config>` style of configuration makes heavy use of Spring's auto-proxying mechanism. This can cause issues (such as advice not being woven) if you already use explicit auto-proxying through the use of `BeanNameAutoProxyCreator` or something similar. The recommended usage pattern is to use either only the `<aop:config>` style or only the `AutoProxyCreator` style and never mix them.

Declaring an Aspect

When you use the schema support, an aspect is a regular Java object defined as a bean in your Spring application context. The state and behavior are captured in the fields and methods of the object, and the pointcut and advice information are captured in the XML.

You can declare an aspect by using the `<aop:aspect>` element, and reference the backing bean by using the `ref` attribute, as the following example shows:

```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
    ...
</bean>
```

The bean that backs the aspect (`aBean` in this case) can of course be configured and dependency injected just like any other Spring bean.

Declaring a Pointcut

You can declare a named pointcut inside an `<aop:config>` element, letting the pointcut definition be shared across several aspects and advisors.

A pointcut that represents the execution of any business service in the service layer can be defined as follows:

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in @AspectJ support. If you use the schema based declaration style, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression. Another way of defining the above pointcut would be as follows:

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="com.xyz.myapp.CommonPointcuts.businessService()"/>

</aop:config>
```

Assume that you have a `CommonPointcuts` aspect as described in Sharing Common Pointcut Definitions.

Then declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut, as the following example shows:

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        ...
    </aop:aspect>

</aop:config>
```

In much the same way as an @AspectJ aspect, pointcuts declared by using the schema based definition style can collect join point context. For example, the following pointcut collects the `this` object as the join point context and passes it to the advice:

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) &amp;&amp;
this(service)"/>

        <aop:before pointcut-ref="businessService" method="monitor"/>

        ...
    </aop:aspect>

</aop:config>
```

The advice must be declared to receive the collected join point context by including parameters of the matching names, as follows:

Java

```
public void monitor(Object service) {
    // ...
}
```

Kotlin

```
fun monitor(service: Any) {
    // ...
}
```

When combining pointcut sub-expressions, `&&` is awkward within an XML document, so you can use the `and`, `or`, and `not` keywords in place of `&&`, `||`, and `!`, respectively. For example, the previous pointcut can be better written as follows:

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) and
this(service)"/>

        <aop:before pointcut-ref="businessService" method="monitor"/>

        ...
    </aop:aspect>
</aop:config>
```

Note that pointcuts defined in this way are referred to by their XML `id` and cannot be used as named pointcuts to form composite pointcuts. The named pointcut support in the schema-based definition style is thus more limited than that offered by the @AspectJ style.

Declaring Advice

The schema-based AOP support uses the same five kinds of advice as the @AspectJ style, and they have exactly the same semantics.

Before Advice

Before advice runs before a matched method execution. It is declared inside an `<aop:aspect>` by using the `<aop:before>` element, as the following example shows:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

Here, `dataAccessOperation` is the `id` of a pointcut defined at the top (`<aop:config>`) level. To define the pointcut inline instead, replace the `pointcut-ref` attribute with a `pointcut` attribute, as follows:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
        method="doAccessCheck"/>

    ...
</aop:aspect>
```

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The `method` attribute identifies a method (`doAccessCheck`) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element that contains the advice. Before a data access operation is performed (a method execution join point matched by the pointcut expression), the `doAccessCheck` method on the aspect bean is invoked.

After Returning Advice

After returning advice runs when a matched method execution completes normally. It is declared inside an `<aop:aspect>` in the same way as before advice. The following example shows how to

declare it:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...
</aop:aspect>
```

As in the @AspectJ style, you can get the return value within the advice body. To do so, use the `returning` attribute to specify the name of the parameter to which the return value should be passed, as the following example shows:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        returning="retVal"
        method="doAccessCheck"/>

    ...
</aop:aspect>
```

The `doAccessCheck` method must declare a parameter named `retVal`. The type of this parameter constrains matching in the same way as described for `@AfterReturning`. For example, you can declare the method signature as follows:

Java

```
public void doAccessCheck(Object retVal) {...
```

Kotlin

```
fun doAccessCheck(retVal: Any) {...
```

After Throwing Advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared inside an `<aop:aspect>` by using the `after-throwing` element, as the following example shows:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        method="doRecoveryActions"/>

    ...
</aop:aspect>
```

As in the @AspectJ style, you can get the thrown exception within the advice body. To do so, use the `throwing` attribute to specify the name of the parameter to which the exception should be passed as the following example shows:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        throwing="dataAccessEx"
        method="doRecoveryActions"/>

    ...
</aop:aspect>
```

The `doRecoveryActions` method must declare a parameter named `dataAccessEx`. The type of this parameter constrains matching in the same way as described for `@AfterThrowing`. For example, the method signature may be declared as follows:

Java

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

Kotlin

```
fun doRecoveryActions(dataAccessEx: DataAccessException) {...
```

After (Finally) Advice

After (finally) advice runs no matter how a matched method execution exits. You can declare it by using the `after` element, as the following example shows:

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
        pointcut-ref="dataAccessOperation"
        method="doReleaseLock"/>

    ...
</aop:aspect>
```

Around Advice

The last kind of advice is *around* advice. Around advice runs "around" a matched method's execution. It has the opportunity to do work both before and after the method runs and to determine when, how, and even if the method actually gets to run at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner – for example, starting and stopping a timer.

Always use the least powerful form of advice that meets your requirements.

For example, do not use *around* advice if *before* advice is sufficient for your needs.

You can declare around advice by using the `aop:around` element. The advice method should declare `Object` as its return type, and the first parameter of the method must be of type `ProceedingJoinPoint`. Within the body of the advice method, you must invoke `proceed()` on the `ProceedingJoinPoint` in order for the underlying method to run. Invoking `proceed()` without arguments will result in the caller's original arguments being supplied to the underlying method when it is invoked. For advanced use cases, there is an overloaded variant of the `proceed()` method which accepts an array of arguments (`Object[]`). The values in the array will be used as the arguments to the underlying method when it is invoked. See Around Advice for notes on calling `proceed` with an `Object[]`.

The following example shows how to declare around advice in XML:

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
        pointcut-ref="businessService"
        method="doBasicProfiling"/>

    ...
</aop:aspect>
```

The implementation of the `doBasicProfiling` advice can be exactly the same as in the @AspectJ example (minus the annotation, of course), as the following example shows:

Java

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
}
```

Kotlin

```
fun doBasicProfiling(pjp: ProceedingJoinPoint): Any {
    // start stopwatch
    val retVal = pjp.proceed()
    // stop stopwatch
    return pjp.proceed()
}
```

Advice Parameters

The schema-based declaration style supports fully typed advice in the same way as described for the @AspectJ support — by matching pointcut parameters by name against advice method parameters. See Advice Parameters for details. If you wish to explicitly specify argument names for the advice methods (not relying on the detection strategies previously described), you can do so by using the `arg-names` attribute of the advice element, which is treated in the same manner as the `argNames` attribute in an advice annotation (as described in Determining Argument Names). The following example shows how to specify an argument name in XML:

```
<aop:before
    pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
    method="audit"
    arg-names="auditable"/>
```

The `arg-names` attribute accepts a comma-delimited list of parameter names.

The following slightly more involved example of the XSD-based approach shows some around advice used in conjunction with a number of strongly typed parameters:

Java

```
package x.y.service;

public interface PersonService {

    Person getPerson(String personName, int age);
}

public class DefaultPersonService implements PersonService {

    public Person getPerson(String name, int age) {
        return new Person(name, age);
    }
}
```

Kotlin

```
package x.y.service

interface PersonService {

    fun getPerson(personName: String, age: Int): Person
}

class DefaultPersonService : PersonService {

    fun getPerson(name: String, age: Int): Person {
        return Person(name, age)
    }
}
```

Next up is the aspect. Notice the fact that the `profile(..)` method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call. The presence of this parameter is an indication that the `profile(..)` is to be used as `around` advice, as the following example shows:

Java

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

    public Object profile(ProceedingJoinPoint call, String name, int age) throws
Throwable {
        StopWatch clock = new StopWatch("Profiling for '" + name + "' and '" + age +
"'");
        try {
            clock.start(call.toShortString());
            return call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
    }
}
```

Kotlin

```
import org.aspectj.lang.ProceedingJoinPoint
import org.springframework.util.StopWatch

class SimpleProfiler {

    fun profile(call: ProceedingJoinPoint, name: String, age: Int): Any {
        val clock = StopWatch("Profiling for '$name' and '$age'")
        try {
            clock.start(call.toShortString())
            return call.proceed()
        } finally {
            clock.stop()
            println(clock.prettyPrint())
        }
    }
}
```

Finally, the following example XML configuration effects the execution of the preceding advice for a particular join point:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
    <bean id="personService" class="x.y.service.DefaultPersonService"/>

    <!-- this is the actual advice itself -->
    <bean id="profiler" class="x.y.SimpleProfiler"/>

    <aop:config>
        <aop:aspect ref="profiler">

            <aop:pointcut id="theExecutionOfSomePersonServiceMethod"
                expression="execution(*
x.y.service.PersonService.getPerson(String,int))
                and args(name, age)"/>

            <aop:around pointcut-ref="theExecutionOfSomePersonServiceMethod"
                method="profile"/>

        </aop:aspect>
    </aop:config>

</beans>
```

Consider the following driver script:

Java

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.PersonService;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
        PersonService person = (PersonService) ctx.getBean("personService");
        person.getPerson("Pengo", 12);
    }
}
```

Kotlin

```
fun main() {
    val ctx = ClassPathXmlApplicationContext("x/y/plain.xml")
    val person = ctx.getBean("personService") as PersonService
    person.getPerson("Pengo", 12)
}
```

With such a Boot class, we would get output similar to the following on standard output:

```
StopWatch 'Profiling for 'Pengo' and '12': running time (millis) = 0
-----------------------------------------
ms     %     Task name
-----------------------------------------
00000  ?  execution(getFoo)
```

Advice Ordering

When multiple pieces of advice need to run at the same join point (executing method) the ordering rules are as described in Advice Ordering. The precedence between aspects is determined via the `order` attribute in the `<aop:aspect>` element or by either adding the `@Order` annotation to the bean that backs the aspect or by having the bean implement the `Ordered` interface.

In contrast to the precedence rules for advice methods defined in the same `@Aspect` class, when two pieces of advice defined in the same `<aop:aspect>` element both need to run at the same join point, the precedence is determined by the order in which the advice elements are declared within the enclosing `<aop:aspect>` element, from highest to lowest precedence.

For example, given an `around` advice and a `before` advice defined in the same `<aop:aspect>` element that apply to the same join point, to ensure that the `around` advice has higher precedence than the `before` advice, the `<aop:around>` element must be declared before the `<aop:before>` element.

As a general rule of thumb, if you find that you have multiple pieces of advice defined in the same `<aop:aspect>` element that apply to the same join point, consider collapsing such advice methods into one advice method per join point in each `<aop:aspect>` element or refactor the pieces of advice into separate `<aop:aspect>` elements that you can order at the aspect level.

Introductions

Introductions (known as inter-type declarations in AspectJ) let an aspect declare that advised objects implement a given interface and provide an implementation of that interface on behalf of those objects.

You can make an introduction by using the `aop:declare-parents` element inside an `aop:aspect`. You can use the `aop:declare-parents` element to declare that matching types have a new parent (hence

the name). For example, given an interface named `UsageTracked` and an implementation of that interface named `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics through JMX for example.)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

    <aop:declare-parents
        types-matching="com.xzy.myapp.service.*+"
        implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
        default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

    <aop:before
        pointcut="com.xyz.myapp.CommonPointcuts.businessService()
            and this(usageTracked)"
            method="recordUsage"/>

</aop:aspect>
```

The class that backs the `usageTracking` bean would then contain the following method:

Java

```
public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
}
```

Kotlin

```
fun recordUsage(usageTracked: UsageTracked) {
    usageTracked.incrementUseCount()
}
```

The interface to be implemented is determined by the `implement-interface` attribute. The value of the `types-matching` attribute is an AspectJ type pattern. Any bean of a matching type implements the `UsageTracked` interface. Note that, in the before advice of the preceding example, service beans can be directly used as implementations of the `UsageTracked` interface. To access a bean programmatically, you could write the following:

Java

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

Kotlin

```
val usageTracked = context.getBean("myService") as UsageTracked
```

Aspect Instantiation Models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

Advisors

The concept of "advisors" comes from the AOP support defined in Spring and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean and must implement one of the advice interfaces described in Advice Types in Spring. Advisors can take advantage of AspectJ pointcut expressions.

Spring supports the advisor concept with the `<aop:advisor>` element. You most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring. The following example shows an advisor:

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:advisor
        pointcut-ref="businessService"
        advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

As well as the `pointcut-ref` attribute used in the preceding example, you can also use the `pointcut` attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the `order` attribute to define the `Ordered` value of the advisor.

An AOP Schema Example

This section shows how the concurrent locking failure retry example from An AOP Example looks when rewritten with the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, a deadlock loser). If the operation is retried, it is likely to succeed on the next try. For business services where it is appropriate to retry in such conditions (idempotent operations that do not need to go back to the user for conflict resolution), we want to transparently retry the operation to avoid the client seeing a `PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer and, hence, is ideal for implementing through an

aspect.

Because we want to retry the operation, we need to use around advice so that we can call `proceed` multiple times. The following listing shows the basic aspect implementation (which is a regular Java class that uses the schema support):

Java

```
public class ConcurrentOperationExecutor implements Ordered {

    private static final int DEFAULT_MAX_RETRIES = 2;

    private int maxRetries = DEFAULT_MAX_RETRIES;
    private int order = 1;

    public void setMaxRetries(int maxRetries) {
        this.maxRetries = maxRetries;
    }

    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        PessimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(PessimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        } while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }
}
```

Kotlin

```
class ConcurrentOperationExecutor : Ordered {

    private val DEFAULT_MAX_RETRIES = 2

    private var maxRetries = DEFAULT_MAX_RETRIES
    private var order = 1

    fun setMaxRetries(maxRetries: Int) {
        this.maxRetries = maxRetries
    }

    override fun getOrder(): Int {
        return this.order
    }

    fun setOrder(order: Int) {
        this.order = order
    }

    fun doConcurrentOperation(pjp: ProceedingJoinPoint): Any {
        var numAttempts = 0
        var lockFailureException: PessimisticLockingFailureException
        do {
            numAttempts++
            try {
                return pjp.proceed()
            } catch (ex: PessimisticLockingFailureException) {
                lockFailureException = ex
            }

        } while (numAttempts <= this.maxRetries)
        throw lockFailureException
    }
}
```

Note that the aspect implements the `Ordered` interface so that we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties are both configured by Spring. The main action happens in the `doConcurrentOperation` around advice method. We try to proceed. If we fail with a `PessimisticLockingFailureException`, we try again, unless we have exhausted all of our retry attempts.

This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is as follows:

```
<aop:config>

    <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

        <aop:pointcut id="idempotentOperation"
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        <aop:around
            pointcut-ref="idempotentOperation"
            method="doConcurrentOperation"/>

    </aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"
    class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
        <property name="maxRetries" value="3"/>
        <property name="order" value="100"/>
</bean>
```

Notice that, for the time being, we assume that all business services are idempotent. If this is not the case, we can refine the aspect so that it retries only genuinely idempotent operations, by introducing an `Idempotent` annotation and using the annotation to annotate the implementation of service operations, as the following example shows:

Java

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
annotation class Idempotent {
    // marker annotation
}
```

The change to the aspect to retry only idempotent operations involves refining the pointcut expression so that only `@Idempotent` operations match, as follows:

```
<aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..)) and
        @annotation(com.xyz.myapp.service.Idempotent)"/>
```

2.5.6. Choosing which AOP Declaration Style to Use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ and between the Aspect language (code) style, the @AspectJ annotation style, or the Spring XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

Spring AOP or Full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ, as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects, typically), you need to use AspectJ. You also need to use AspectJ if you wish to advise join points other than simple method executions (for example, field get or set join points and so on).

When you use AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. Clearly, if you do not use Java 5+, the choice has been made for you: Use the code style. If aspects play a large role in your design, and you are able to use the AspectJ Development Tools (AJDT) plugin for Eclipse, the AspectJ language syntax is the preferred option. It is cleaner and simpler because the language was purposefully designed for writing aspects. If you do not use Eclipse or have only a few aspects that do not play a major role in your application, you may want to consider using the @AspectJ style, sticking with regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

@AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, you have a choice of @AspectJ or XML style. There are various tradeoffs to consider.

The XML style may be most familiar to existing Spring users, and it is backed by genuine POJOs. When using AOP as a tool to configure enterprise services, XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration that you might want to change independently). With the XML style, it is arguably clearer from your configuration which aspects are present in the system.

The XML style has two disadvantages. First, it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of how a requirement is implemented is split across the declaration of the backing bean class and the XML in the configuration file. When you use the @AspectJ style, this information is encapsulated in a single module: the aspect. Secondly, the XML style is slightly more limited in what it can express than the @AspectJ style: Only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style you can write something like the following:

Java

```
@Pointcut("execution(* get*())")
public void propertyAccess() {}

@Pointcut("execution(org.xyz.Account+ *(..))")
public void operationReturningAnAccount() {}

@Pointcut("propertyAccess() && operationReturningAnAccount()")
public void accountPropertyAccess() {}
```

Kotlin

```
@Pointcut("execution(* get*())")
fun propertyAccess() {}

@Pointcut("execution(org.xyz.Account+ *(..))")
fun operationReturningAnAccount() {}

@Pointcut("propertyAccess() && operationReturningAnAccount()")
fun accountPropertyAccess() {}
```

In the XML style you can declare the first two pointcuts:

```
<aop:pointcut id="propertyAccess"
        expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
        expression="execution(org.xyz.Account+ *(..))"/>
```

The downside of the XML approach is that you cannot define the `accountPropertyAccess` pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage that the @AspectJ aspects can be understood (and thus consumed) both by Spring AOP and by AspectJ. So, if you later decide you need the capabilities of AspectJ to implement additional requirements, you can easily migrate to a classic AspectJ setup. On balance, the Spring team prefers the @AspectJ style for custom aspects beyond simple configuration of enterprise services.

2.5.7. Mixing Aspect Types

It is perfectly possible to mix @AspectJ style aspects by using the auto-proxying support, schema-defined `<aop:aspect>` aspects, `<aop:advisor>` declared advisors, and even proxies and interceptors in other styles in the same configuration. All of these are implemented by using the same underlying support mechanism and can co-exist without any difficulty.

2.5.8. Proxying Mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. JDK dynamic proxies are built into the JDK, whereas CGLIB is a common open-source class definition library (repackaged into `spring-core`).

If the target object to be proxied implements at least one interface, a JDK dynamic proxy is used. All of the interfaces implemented by the target type are proxied. If the target object does not implement any interfaces, a CGLIB proxy is created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not only those implemented by its interfaces), you can do so. However, you should consider the following issues:

- With CGLIB, `final` methods cannot be advised, as they cannot be overridden in runtime-generated subclasses.
- As of Spring 4.0, the constructor of your proxied object is NOT called twice anymore, since the CGLIB proxy instance is created through Objenesis. Only if your JVM does not allow for constructor bypassing, you might see double invocations and corresponding debug log entries from Spring's AOP support.

To force the use of CGLIB proxies, set the value of the `proxy-target-class` attribute of the `<aop:config>` element to true, as follows:

```
<aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
</aop:config>
```

To force CGLIB proxying when you use the @AspectJ auto-proxy support, set the `proxy-target-class` attribute of the `<aop:aspectj-autoproxy>` element to `true`, as follows:

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

Multiple `<aop:config/>` sections are collapsed into a single unified auto-proxy creator at runtime, which applies the *strongest* proxy settings that any of the `<aop:config/>` sections (typically from different XML bean definition files) specified. This also applies to the `<tx:annotation-driven/>` and `<aop:aspectj-autoproxy/>` elements.

To be clear, using `proxy-target-class="true"` on `<tx:annotation-driven/>`, `<aop:aspectj-autoproxy/>`, or `<aop:config/>` elements forces the use of CGLIB proxies *for all three of them*.

Understanding AOP Proxies

Spring AOP is proxy-based. It is vitally important that you grasp the semantics of what that last statement actually means before you write your own aspects or use any of the Spring AOP-based

aspects supplied with the Spring Framework.

Consider first the scenario where you have a plain-vanilla, un-proxied, nothing-special-about-it, straight object reference, as the following code snippet shows:

Java

```
public class SimplePojo implements Pojo {

    public void foo() {
        // this next method invocation is a direct call on the 'this' reference
        this.bar();
    }

    public void bar() {
        // some logic...
    }
}
```

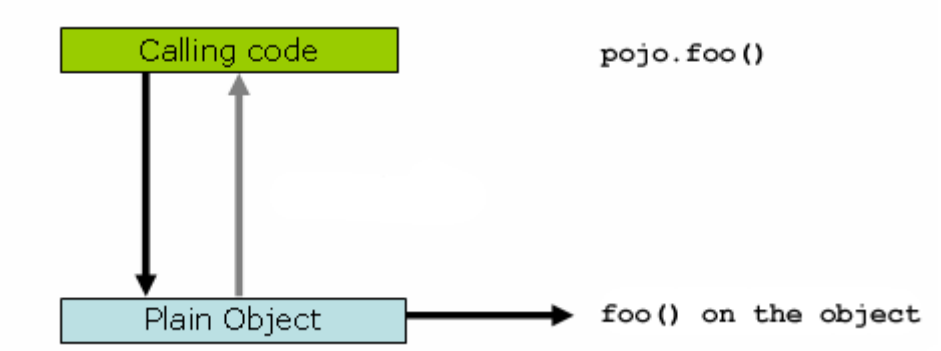
Kotlin

```
class SimplePojo : Pojo {

    fun foo() {
        // this next method invocation is a direct call on the 'this' reference
        this.bar()
    }

    fun bar() {
        // some logic...
    }
}
```

If you invoke a method on an object reference, the method is invoked directly on that object reference, as the following image and listing show:



Java

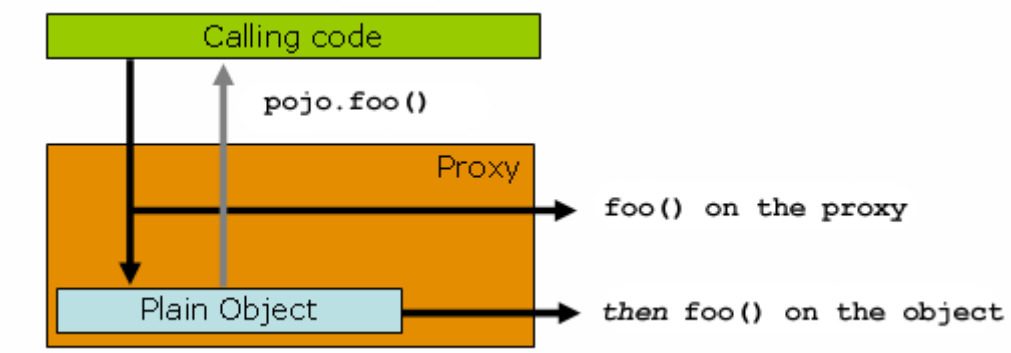
```
public class Main {

    public static void main(String[] args) {
        Pojo pojo = new SimplePojo();
        // this is a direct method call on the 'pojo' reference
        pojo.foo();
    }
}
```

Kotlin

```
fun main() {
    val pojo = SimplePojo()
    // this is a direct method call on the 'pojo' reference
    pojo.foo()
}
```

Things change slightly when the reference that client code has is a proxy. Consider the following diagram and code snippet:



Java

```
public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());

        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

Kotlin

```
fun main() {
    val factory = ProxyFactory(SimplePojo())
    factory.addInterface(Pojo::class.java)
    factory.addAdvice(RetryAdvice())

    val pojo = factory.proxy as Pojo
    // this is a method call on the proxy!
    pojo.foo()
}
```

The key thing to understand here is that the client code inside the `main(..)` method of the `Main` class has a reference to the proxy. This means that method calls on that object reference are calls on the proxy. As a result, the proxy can delegate to all of the interceptors (advice) that are relevant to that particular method call. However, once the call has finally reached the target object (the `SimplePojo` reference in this case), any method calls that it may make on itself, such as `this.bar()` or `this.foo()`, are going to be invoked against the `this` reference, and not the proxy. This has important implications. It means that self-invocation is not going to result in the advice associated with a method invocation getting a chance to run.

Okay, so what is to be done about this? The best approach (the term "best" is used loosely here) is to refactor your code such that the self-invocation does not happen. This does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and we hesitate to point it out, precisely because it is so horrendous. You can (painful as it is to us) totally tie the logic within your class to Spring AOP, as the following example shows:

Java

```
public class SimplePojo implements Pojo {

    public void foo() {
        // this works, but... gah!
        ((Pojo) AopContext.currentProxy()).bar();
    }

    public void bar() {
        // some logic...
    }
}
```

Kotlin

```
class SimplePojo : Pojo {

    fun foo() {
        // this works, but... gah!
        (AopContext.currentProxy() as Pojo).bar()
    }

    fun bar() {
        // some logic...
    }
}
```

This totally couples your code to Spring AOP, and it makes the class itself aware of the fact that it is being used in an AOP context, which flies in the face of AOP. It also requires some additional configuration when the proxy is being created, as the following example shows:

Java

```
public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());
        factory.setExposeProxy(true);

        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

Kotlin

```
fun main() {
    val factory = ProxyFactory(SimplePojo())
    factory.addInterface(Pojo::class.java)
    factory.addAdvice(RetryAdvice())
    factory.isExposeProxy = true

    val pojo = factory.proxy as Pojo
    // this is a method call on the proxy!
    pojo.foo()
}
```

Finally, it must be noted that AspectJ does not have this self-invocation issue because it is not a proxy-based AOP framework.

2.5.9. Programmatic Creation of @AspectJ Proxies

In addition to declaring aspects in your configuration by using either `<aop:config>` or `<aop:aspectj-autoproxy>`, it is also possible to programmatically create proxies that advise target objects. For the full details of Spring's AOP API, see the next chapter. Here, we want to focus on the ability to automatically create proxies by using @AspectJ aspects.

You can use the `org.springframework.aop.aspectj.annotation.AspectJProxyFactory` class to create a proxy for a target object that is advised by one or more @AspectJ aspects. The basic usage for this class is very simple, as the following example shows:

Java

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be
an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

Kotlin

```
// create a factory that can generate a proxy for the given target object
val factory = AspectJProxyFactory(targetObject)

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager::class.java)

// you can also add existing aspect instances, the type of the object supplied must be
an @AspectJ aspect
factory.addAspect(usageTracker)

// now get the proxy object...
val proxy = factory.getProxy<Any>()
```

See the javadoc for more information.

2.5.10. Using AspectJ with Spring Applications

Everything we have covered so far in this chapter is pure Spring AOP. In this section, we look at how you can use the AspectJ compiler or weaver instead of or in addition to Spring AOP if your

needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library, which is available stand-alone in your distribution as `spring-aspects.jar`. You need to add this to your classpath in order to use the aspects in it. Using AspectJ to Dependency Inject Domain Objects with Spring and Other Spring aspects for AspectJ discuss the content of this library and how you can use it. Configuring AspectJ Aspects by Using Spring IoC discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, Load-time Weaving with AspectJ in the Spring Framework provides an introduction to load-time weaving for Spring applications that use AspectJ.

Using AspectJ to Dependency Inject Domain Objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a pre-existing object, given the name of a bean definition that contains the configuration to be applied. `spring-aspects.jar` contains an annotation-driven aspect that exploits this capability to allow dependency injection of any object. The support is intended to be used for objects created outside of the control of any container. Domain objects often fall into this category because they are often created programmatically with the `new` operator or by an ORM tool as a result of a database query.

The `@Configurable` annotation marks a class as being eligible for Spring-driven configuration. In the simplest case, you can use purely it as a marker annotation, as the following example shows:

Java

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Account {
    // ...
}
```

Kotlin

```
package com.xyz.myapp.domain

import org.springframework.beans.factory.annotation.Configurable

@Configurable
class Account {
    // ...
}
```

When used as a marker interface in this way, Spring configures new instances of the annotated type (`Account`, in this case) by using a bean definition (typically prototype-scoped) with the same name as the fully-qualified type name (`com.xyz.myapp.domain.Account`). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is

to omit the `id` attribute, as the following example shows:

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
    <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation, as the following example shows:

Java

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
    // ...
}
```

Kotlin

```
package com.xyz.myapp.domain

import org.springframework.beans.factory.annotation.Configurable

@Configurable("account")
class Account {
    // ...
}
```

Spring now looks for a bean definition named `account` and uses that as the definition to configure new `Account` instances.

You can also use autowiring to avoid having to specify a dedicated bean definition at all. To have Spring apply autowiring, use the `autowire` property of the `@Configurable` annotation. You can specify either `@Configurable(autowire=Autowire.BY_TYPE)` or `@Configurable(autowire=Autowire.BY_NAME)` for autowiring by type or by name, respectively. As an alternative, it is preferable to specify explicit, annotation-driven dependency injection for your `@Configurable` beans through `@Autowired` or `@Inject` at the field or method level (see Annotation-based Container Configuration for further details).

Finally, you can enable Spring dependency checking for the object references in the newly created and configured object by using the `dependencyCheck` attribute (for example, `@Configurable(autowire=Autowire.BY_NAME,dependencyCheck=true)`). If this attribute is set to `true`, Spring validates after configuration that all properties (which are not primitives or collections) have been set.

Note that using the annotation on its own does nothing. It is the `AnnotationBeanConfigurerAspect` in `spring-aspects.jar` that acts on the presence of the annotation. In essence, the aspect says, "after returning from the initialization of a new object of a type annotated with `@Configurable`, configure the newly created object using Spring in accordance with the properties of the annotation". In this context, "initialization" refers to newly instantiated objects (for example, objects instantiated with the `new` operator) as well as to `Serializable` objects that are undergoing deserialization (for example, through readResolve()).

One of the key phrases in the above paragraph is "in essence". For most cases, the exact semantics of "after returning from the initialization of a new object" are fine. In this context, "after initialization" means that the dependencies are injected after the object has been constructed. This means that the dependencies are not available for use in the constructor bodies of the class. If you want the dependencies to be injected before the constructor bodies run and thus be available for use in the body of the constructors, you need to define this on the `@Configurable` declaration, as follows:

Java

```
@Configurable(preConstruction = true)
```

Kotlin

```
@Configurable(preConstruction = true)
```

You can find more information about the language semantics of the various pointcut types in AspectJ in this appendix of the AspectJ Programming Guide.

For this to work, the annotated types must be woven with the AspectJ weaver. You can either use a build-time Ant or Maven task to do this (see, for example, the AspectJ Development Environment Guide) or load-time weaving (see Load-time Weaving with AspectJ in the Spring Framework). The `AnnotationBeanConfigurerAspect` itself needs to be configured by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). If you use Java-based configuration, you can add `@EnableSpringConfigured` to any `@Configuration` class, as follows:

Java

```
@Configuration
@EnableSpringConfigured
public class AppConfig {
}
```

Kotlin

```
@Configuration
@EnableSpringConfigured
class AppConfig {
}
```

If you prefer XML based configuration, the Spring `context` namespace defines a convenient `context:spring-configured` element, which you can use as follows:

```
<context:spring-configured/>
```

Instances of `@Configurable` objects created before the aspect has been configured result in a message being issued to the debug log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case, you can use the `depends-on` bean attribute to manually specify that the bean depends on the configuration aspect. The following example shows how to use the `depends-on` attribute:

```
<bean id="myService"
        class="com.xzy.myapp.service.MyService"
        depends-
on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">

    <!-- ... -->

</bean>
```

Do not activate `@Configurable` processing through the bean configurer aspect unless you really mean to rely on its semantics at runtime. In particular, make sure that you do not use `@Configurable` on bean classes that are registered as regular Spring beans with the container. Doing so results in double initialization, once through the container and once through the aspect.

Unit Testing `@Configurable` Objects

One of the goals of the `@Configurable` support is to enable independent unit testing of domain objects without the difficulties associated with hard-coded lookups. If `@Configurable` types have not been woven by AspectJ, the annotation has no affect during unit testing. You can set mock or stub property references in the object under test and proceed as normal. If `@Configurable` types have been woven by AspectJ, you can still unit test outside of the container as normal, but you see a warning message each time that you construct a `@Configurable` object indicating that it has not been configured by Spring.

Working with Multiple Application Contexts

The `AnnotationBeanConfigurerAspect` that is used to implement the `@Configurable` support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of `static` members: There is one aspect instance per classloader that defines the type. This means that, if you define multiple application contexts within the same classloader hierarchy, you need to consider where to define the `@EnableSpringConfigured` bean and where to place `spring-aspects.jar` on the classpath.

Consider a typical Spring web application configuration that has a shared parent application context that defines common business services, everything needed to support those services, and one child application context for each servlet (which contains definitions particular to that servlet). All of these contexts co-exist within the same classloader hierarchy, and so the `AnnotationBeanConfigurerAspect` can hold a reference to only one of them. In this case, we recommend defining the `@EnableSpringConfigured` bean in the shared (parent) application context. This defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts by using the @Configurable mechanism (which is probably not something you want to do anyway).

When deploying multiple web applications within the same container, ensure that each web application loads the types in `spring-aspects.jar` by using its own classloader (for example, by placing `spring-aspects.jar` in `WEB-INF/lib`). If `spring-aspects.jar` is added only to the container-wide classpath (and hence loaded by the shared parent classloader), all web applications share the same aspect instance (which is probably not what you want).

Other Spring aspects for AspectJ

In addition to the `@Configurable` aspect, `spring-aspects.jar` contains an AspectJ aspect that you can use to drive Spring's transaction management for types and methods annotated with the `@Transactional` annotation. This is primarily intended for users who want to use the Spring Framework's transaction support outside of the Spring container.

The aspect that interprets `@Transactional` annotations is the `AnnotationTransactionAspect`. When you use this aspect, you must annotate the implementation class (or methods within that class or both), not the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are not inherited.

A `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any public operation in the class.

A `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods of any visibility may be annotated, including private methods. Annotating non-public methods directly is the only way to get transaction demarcation for the execution of such methods.

Since Spring Framework 4.2, `spring-aspects` provides a similar aspect that offers the exact same features for the standard `jakarta.transaction.Transactional` annotation. Check `JtaAnnotationTransactionAspect` for more details.

For AspectJ programmers who want to use the Spring configuration and transaction management support but do not want to (or cannot) use annotations, `spring-aspects.jar` also contains `abstract` aspects you can extend to provide your own pointcut definitions. See the sources for the `AbstractBeanConfigurerAspect` and `AbstractTransactionAspect` aspects for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model by using prototype bean definitions that match the fully qualified class names:

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

    public DomainObjectConfiguration() {
        setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());
    }

    // the creation of a new bean (any object in the domain model)
    protected pointcut beanCreation(Object beanInstance) :
        initialization(new(..)) &&
        CommonPointcuts.inDomainModel() &&
        this(beanInstance);
}
```

Configuring AspectJ Aspects by Using Spring IoC

When you use AspectJ aspects with Spring applications, it is natural to both want and expect to be able to configure such aspects with Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ-created aspects through Spring depends on the AspectJ instantiation model (the `per-xxx` clause) used by the aspect.

The majority of AspectJ aspects are singleton aspects. Configuration of these aspects is easy. You can create a bean definition that references the aspect type as normal and include the `factory-method="aspectOf"` bean attribute. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. The following example shows how to use the `factory-method="aspectOf"` attribute:

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
        factory-method="aspectOf"> ①

    <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
</bean>
```

① Note the `factory-method="aspectOf"` attribute

Non-singleton aspects are harder to configure. However, it is possible to do so by creating prototype bean definitions and using the `@Configurable` support from `spring-aspects.jar` to configure the aspect instances once they have bean created by the AspectJ runtime.

If you have some @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring

AOP, and these aspects are all configured in Spring, you need to tell the Spring AOP @AspectJ auto-proxying support which exact subset of the @AspectJ aspects defined in the configuration should be used for auto-proxying. You can do this by using one or more `<include/>` elements inside the `<aop:aspectj-autoproxy/>` declaration. Each `<include/>` element specifies a name pattern, and only beans with names matched by at least one of the patterns are used for Spring AOP auto-proxy configuration. The following example shows how to use `<include/>` elements:

```
<aop:aspectj-autoproxy>
    <aop:include name="thisBean"/>
    <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

Do not be misled by the name of the `<aop:aspectj-autoproxy/>` element. Using it results in the creation of Spring AOP proxies. The @AspectJ style of aspect declaration is being used here, but the AspectJ runtime is not involved.

Load-time Weaving with AspectJ in the Spring Framework

Load-time weaving (LTW) refers to the process of weaving AspectJ aspects into an application's class files as they are being loaded into the Java virtual machine (JVM). The focus of this section is on configuring and using LTW in the specific context of the Spring Framework. This section is not a general introduction to LTW. For full details on the specifics of LTW and configuring LTW with only AspectJ (with Spring not being involved at all), see the LTW section of the AspectJ Development Environment Guide.

The value that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected by using a Java (5+) agent, which is switched on by specifying a VM argument when starting up a JVM. It is, thus, a JVM-wide setting, which may be fine in some situations but is often a little too coarse. Spring-enabled LTW lets you switch on LTW on a per-`ClassLoader` basis, which is more fine-grained and which can make more sense in a 'single-JVM-multiple-application' environment (such as is found in a typical application server environment).

Further, in certain environments, this support enables load-time weaving without making any modifications to the application server's launch script that is needed to add `-javaagent:path/to/aspectjweaver.jar` or (as we describe later in this section) `-javaagent:path/to/spring-instrument.jar`. Developers configure the application context to enable load-time weaving instead of relying on administrators who typically are in charge of the deployment configuration, such as the launch script.

Now that the sales pitch is over, let us first walk through a quick example of AspectJ LTW that uses Spring, followed by detailed specifics about elements introduced in the example. For a complete example, see the Petclinic sample application.

A First Example

Assume that you are an application developer who has been tasked with diagnosing the cause of some performance problems in a system. Rather than break out a profiling tool, we are going to

switch on a simple profiling aspect that lets us quickly get some performance metrics. We can then apply a finer-grained profiling tool to that specific area immediately afterwards.

The example presented here uses XML configuration. You can also configure and use @AspectJ with Java configuration. Specifically, you can use the `@EnableLoadTimeWeaving` annotation as an alternative to `<context:load-time-weaver/>` (see below for details).

The following example shows the profiling aspect, which is not fancy. It is a time-based profiler that uses the @AspectJ-style of aspect declaration:

Java

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled(){}
}
```

Kotlin

```
package foo

import org.aspectj.lang.ProceedingJoinPoint
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Around
import org.aspectj.lang.annotation.Pointcut
import org.springframework.util.StopWatch
import org.springframework.core.annotation.Order

@Aspect
class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    fun profile(pjp: ProceedingJoinPoint): Any {
        val sw = StopWatch(javaClass.simpleName)
        try {
            sw.start(pjp.getSignature().getName())
            return pjp.proceed()
        } finally {
            sw.stop()
            println(sw.prettyPrint())
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    fun methodsToBeProfiled() {
    }
}
```

We also need to create an `META-INF/aop.xml` file, to inform the AspectJ weaver that we want to weave our `ProfilingAspect` into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called `META-INF/aop.xml` is standard AspectJ. The following example shows the `aop.xml` file:

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"https://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

    <weaver>
        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>
    </weaver>

    <aspects>
        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>
    </aspects>

</aspectj>
```

Now we can move on to the Spring-specific portion of the configuration. We need to configure a `LoadTimeWeaver` (explained later). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more `META-INF/aop.xml` files into the classes in your application. The good thing is that it does not require a lot of configuration (there are some more options that you can specify, but these are detailed later), as can be seen in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <!-- a service object; we will be profiling its methods -->
    <bean id="entitlementCalculationService"
            class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>
</beans>
```

Now that all the required artifacts (the aspect, the `META-INF/aop.xml` file, and the Spring configuration) are in place, we can create the following driver class with a `main(..)` method to demonstrate the LTW in action:

Java

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml",
Main.class);

        EntitlementCalculationService entitlementCalculationService =
                (EntitlementCalculationService)
ctx.getBean("entitlementCalculationService");

        // the profiling aspect is 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

Kotlin

```
package foo

import org.springframework.context.support.ClassPathXmlApplicationContext

fun main() {
    val ctx = ClassPathXmlApplicationContext("beans.xml")

    val entitlementCalculationService = ctx.getBean("entitlementCalculationService")
as EntitlementCalculationService

    // the profiling aspect is 'woven' around this method execution
    entitlementCalculationService.calculateEntitlement()
}
```

We have one last thing to do. The introduction to this section did say that one could switch on LTW selectively on a per-`ClassLoader` basis with Spring, and this is true. However, for this example, we use a Java agent (supplied with Spring) to switch on LTW. We use the following command to run the `Main` class shown earlier:

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

The `-javaagent` is a flag for specifying and enabling agents to instrument programs that run on the JVM. The Spring Framework ships with such an agent, the `InstrumentationSavingAgent`, which is packaged in the `spring-instrument.jar` that was supplied as the value of the `-javaagent` argument in the preceding example.

The output from the execution of the `Main` program looks something like the next example. (I have introduced a `Thread.sleep(..)` statement into the `calculateEntitlement()` implementation so that the profiler actually captures something other than 0 milliseconds (the `01234` milliseconds is not an overhead introduced by the AOP). The following listing shows the output we got when we ran our profiler:

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
------ ----- ----------------------------
ms     %     Task name
------ ----- ----------------------------
01234  100%  calculateEntitlement
```

Since this LTW is effected by using full-blown AspectJ, we are not limited only to advising Spring beans. The following slight variation on the `Main` program yields the same result:

Java

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService =
                new StubEntitlementCalculationService();

        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

Kotlin

```
package foo

import org.springframework.context.support.ClassPathXmlApplicationContext

fun main(args: Array<String>) {
    ClassPathXmlApplicationContext("beans.xml")

    val entitlementCalculationService = StubEntitlementCalculationService()

    // the profiling aspect will be 'woven' around this method execution
    entitlementCalculationService.calculateEntitlement()
}
```

Notice how, in the preceding program, we bootstrap the Spring container and then create a new instance of the `StubEntitlementCalculationService` totally outside the context of Spring. The profiling advice still gets woven in.

Admittedly, the example is simplistic. However, the basics of the LTW support in Spring have all been introduced in the earlier example, and the rest of this section explains the "why" behind each bit of configuration and usage in detail.

The `ProfilingAspect` used in this example may be basic, but it is quite useful. It is a nice example of a development-time aspect that developers can use during development and then easily exclude from builds of the application being deployed into UAT or production.

Aspects

The aspects that you use in LTW have to be AspectJ aspects. You can write them in either the AspectJ language itself, or you can write your aspects in the @AspectJ-style. Your aspects are then both valid AspectJ and Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the classpath.

'META-INF/aop.xml'

The AspectJ LTW infrastructure is configured by using one or more `META-INF/aop.xml` files that are on the Java classpath (either directly or, more typically, in jar files).

The structure and contents of this file is detailed in the LTW part of the AspectJ reference documentation. Because the `aop.xml` file is 100% AspectJ, we do not describe it further here.

Required libraries (JARS)

At minimum, you need the following libraries to use the Spring Framework's support for AspectJ LTW:

- `spring-aop.jar`

- `aspectjweaver.jar`

If you use the Spring-provided agent to enable instrumentation, you also need:

- `spring-instrument.jar`

Spring Configuration

The key component in Spring's LTW support is the `LoadTimeWeaver` interface (in the `org.springframework.instrument.classloading` package), and the numerous implementations of it that ship with the Spring distribution. A `LoadTimeWeaver` is responsible for adding one or more `java.lang.instrument.ClassFileTransformers` to a `ClassLoader` at runtime, which opens the door to all manner of interesting applications, one of which happens to be the LTW of aspects.

If you are unfamiliar with the idea of runtime class file transformation, see the javadoc API documentation for the `java.lang.instrument` package before continuing. While that documentation is not comprehensive, at least you can see the key interfaces and classes (for reference as you read through this section).

Configuring a `LoadTimeWeaver` for a particular `ApplicationContext` can be as easy as adding one line. (Note that you almost certainly need to use an `ApplicationContext` as your Spring container — typically, a `BeanFactory` is not enough because the LTW support uses `BeanFactoryPostProcessors`.)

To enable the Spring Framework's LTW support, you need to configure a `LoadTimeWeaver`, which typically is done by using the `@EnableLoadTimeWeaving` annotation, as follows:

Java

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {
}
```

Kotlin

```
@Configuration
@EnableLoadTimeWeaving
class AppConfig {
}
```

Alternatively, if you prefer XML-based configuration, use the `<context:load-time-weaver/>` element. Note that the element is defined in the `context` namespace. The following example shows how to use `<context:load-time-weaver/>`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver/>

</beans>
```

The preceding configuration automatically defines and registers a number of LTW-specific infrastructure beans, such as a `LoadTimeWeaver` and an `AspectJWeavingEnabler`, for you. The default `LoadTimeWeaver` is the `DefaultContextLoadTimeWeaver` class, which attempts to decorate an automatically detected `LoadTimeWeaver`. The exact type of `LoadTimeWeaver` that is "automatically detected" is dependent upon your runtime environment. The following table summarizes various `LoadTimeWeaver` implementations:

Table 13. DefaultContextLoadTimeWeaver LoadTimeWeavers

Runtime Environment	`LoadTimeWeaver` **implementation**
Running in Apache Tomcat	`TomcatLoadTimeWeaver`
Running in GlassFish (limited to EAR deployments)	`GlassFishLoadTimeWeaver`
Running in Red Hat's JBoss AS or WildFly	`JBossLoadTimeWeaver`
Running in IBM's WebSphere	`WebSphereLoadTimeWeaver`
Running in Oracle's WebLogic	`WebLogicLoadTimeWeaver`
JVM started with Spring `InstrumentationSavingAgent` (`java -javaagent:path/to/spring-instrument.jar`)	`InstrumentationLoadTimeWeaver`
Fallback, expecting the underlying ClassLoader to follow common conventions (namely `addTransformer` and optionally a `getThrowawayClassLoader` method)	`ReflectiveLoadTimeWeaver`

Note that the table lists only the `LoadTimeWeavers` that are autodetected when you use the `DefaultContextLoadTimeWeaver`. You can specify exactly which `LoadTimeWeaver` implementation to use.

To specify a specific `LoadTimeWeaver` with Java configuration, implement the `LoadTimeWeavingConfigurer` interface and override the `getLoadTimeWeaver()` method. The following example specifies a `ReflectiveLoadTimeWeaver`:

Java

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig implements LoadTimeWeavingConfigurer {

    @Override
    public LoadTimeWeaver getLoadTimeWeaver() {
        return new ReflectiveLoadTimeWeaver();
    }
}
```

Kotlin

```
@Configuration
@EnableLoadTimeWeaving
class AppConfig : LoadTimeWeavingConfigurer {

    override fun getLoadTimeWeaver(): LoadTimeWeaver {
        return ReflectiveLoadTimeWeaver()
    }
}
```

If you use XML-based configuration, you can specify the fully qualified classname as the value of the `weaver-class` attribute on the `<context:load-time-weaver/>` element. Again, the following example specifies a `ReflectiveLoadTimeWeaver`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:load-time-weaver
            weaver-
class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>
```

The `LoadTimeWeaver` that is defined and registered by the configuration can be later retrieved from the Spring container by using the well known name, `loadTimeWeaver`. Remember that the `LoadTimeWeaver` exists only as a mechanism for Spring's LTW infrastructure to add one or more `ClassFileTransformers`. The actual `ClassFileTransformer` that does the LTW is the `ClassPreProcessorAgentAdapter` (from the `org.aspectj.weaver.loadtime` package) class. See the class-

level javadoc of the `ClassPreProcessorAgentAdapter` class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this document.

There is one final attribute of the configuration left to discuss: the `aspectjWeaving` attribute (or `aspectj-weaving` if you use XML). This attribute controls whether LTW is enabled or not. It accepts one of three possible values, with the default value being `autodetect` if the attribute is not present. The following table summarizes the three possible values:

Table 14. AspectJ weaving attribute values

Annotation Value	XML Value	Explanation
`ENABLED`	`on`	AspectJ weaving is on, and aspects are woven at load-time as appropriate.
`DISABLED`	`off`	LTW is off. No aspect is woven at load-time.
`AUTODETECT`	`autodetect`	If the Spring LTW infrastructure can find at least one `META-INF/aop.xml` file, then AspectJ weaving is on. Otherwise, it is off. This is the default value.

Environment-specific Configuration

This last section contains any additional settings and configuration that you need when you use Spring's LTW support in environments such as application servers and web containers.

Tomcat, JBoss, WebSphere, WebLogic

Tomcat, JBoss/WildFly, IBM WebSphere Application Server and Oracle WebLogic Server all provide a general app `ClassLoader` that is capable of local instrumentation. Spring's native LTW may leverage those ClassLoader implementations to provide AspectJ weaving. You can simply enable load-time weaving, as described earlier. Specifically, you do not need to modify the JVM launch script to add `-javaagent:path/to/spring-instrument.jar`.

Note that on JBoss, you may need to disable the app server scanning to prevent it from loading the classes before the application actually starts. A quick workaround is to add to your artifact a file named `WEB-INF/jboss-scanning.xml` with the following content:

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

Generic Java Applications

When class instrumentation is required in environments that are not supported by specific `LoadTimeWeaver` implementations, a JVM agent is the general solution. For such cases, Spring provides `InstrumentationLoadTimeWeaver` which requires a Spring-specific (but very general) JVM agent, `spring-instrument.jar`, autodetected by common `@EnableLoadTimeWeaving` and `<context:load-`

`time-weaver/>` setups.

To use it, you must start the virtual machine with the Spring agent by supplying the following JVM options:

```
-javaagent:/path/to/spring-instrument.jar
```

Note that this requires modification of the JVM launch script, which may prevent you from using this in application server environments (depending on your server and your operation policies). That said, for one-app-per-JVM deployments such as standalone Spring Boot applications, you typically control the entire JVM setup in any case.

2.5.11. Further Resources

More information on AspectJ can be found on the AspectJ website.

Eclipse AspectJ by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

AspectJ in Action, Second Edition by Ramnivas Laddad (Manning, 2009) comes highly recommended. The focus of the book is on AspectJ, but a lot of general AOP themes are explored (in some depth).

2.6. Spring AOP APIs

The previous chapter described the Spring's support for AOP with @AspectJ and schema-based aspect definitions. In this chapter, we discuss the lower-level Spring AOP APIs. For common applications, we recommend the use of Spring AOP with AspectJ pointcuts as described in the previous chapter.

2.6.1. Pointcut API in Spring

This section describes how Spring handles the crucial pointcut concept.

Concepts

Spring's pointcut model enables pointcut reuse independent of advice types. You can target different advice with the same pointcut.

The `org.springframework.aop.Pointcut` interface is the central interface, used to target advices to particular classes and methods. The complete interface follows:

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();
}
```

Splitting the `Pointcut` interface into two parts allows reuse of class and method matching parts and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ClassFilter` interface is used to restrict the pointcut to a given set of target classes. If the `matches()` method always returns true, all target classes are matched. The following listing shows the `ClassFilter` interface definition:

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

The `MethodMatcher` interface is normally more important. The complete interface follows:

```
public interface MethodMatcher {

    boolean matches(Method m, Class<?> targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class<?> targetClass, Object... args);
}
```

The `matches(Method, Class)` method is used to test whether this pointcut ever matches a given method on a target class. This evaluation can be performed when an AOP proxy is created to avoid the need for a test on every method invocation. If the two-argument `matches` method returns `true` for a given method, and the `isRuntime()` method for the MethodMatcher returns `true`, the three-argument matches method is invoked on every method invocation. This lets a pointcut look at the arguments passed to the method invocation immediately before the target advice starts.

Most `MethodMatcher` implementations are static, meaning that their `isRuntime()` method returns `false`. In this case, the three-argument `matches` method is never invoked.

If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

Operations on Pointcuts

Spring supports operations (notably, union and intersection) on pointcuts.

Union means the methods that either pointcut matches. Intersection means the methods that both pointcuts match. Union is usually more useful. You can compose pointcuts by using the static methods in the `org.springframework.aop.support.Pointcuts` class or by using the `ComposablePointcut` class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

AspectJ Expression Pointcuts

Since 2.0, the most important type of pointcut used by Spring is `org.springframework.aop.aspectj.AspectJExpressionPointcut`. This is a pointcut that uses an AspectJ-supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

Convenience Pointcut Implementations

Spring provides several convenient pointcut implementations. You can use some of them directly; others are intended to be subclassed in application-specific pointcuts.

Static Pointcuts

Static pointcuts are based on the method and the target class and cannot take into account the method's arguments. Static pointcuts suffice — and are best — for most usages. Spring can evaluate a static pointcut only once, when a method is first invoked. After that, there is no need to evaluate the pointcut again with each method invocation.

The rest of this section describes some of the static pointcut implementations that are included with Spring.

Regular Expression Pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. `org.springframework.aop.support.JdkRegexpMethodPointcut` is a generic regular expression pointcut that uses the regular expression support in the JDK.

With the `JdkRegexpMethodPointcut` class, you can provide a list of pattern strings. If any of these is a match, the pointcut evaluates to `true`. (As a consequence, the resulting pointcut is effectively the union of the specified patterns.)

The following example shows how to use `JdkRegexpMethodPointcut`:

```
<bean id="settersAndAbsquatulatePointcut"
        class="org.springframework.aop.support.JdkRegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring provides a convenience class named `RegexpMethodPointcutAdvisor`, which lets us also reference an `Advice` (remember that an `Advice` can be an interceptor, before advice, throws advice, and others). Behind the scenes, Spring uses a `JdkRegexpMethodPointcut`. Using `RegexpMethodPointcutAdvisor` simplifies wiring, as the one bean encapsulates both pointcut and advice, as the following example shows:

```
<bean id="settersAndAbsquatulateAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref bean="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

You can use `RegexpMethodPointcutAdvisor` with any `Advice` type.

Attribute-driven Pointcuts

An important type of static pointcut is a metadata-driven pointcut. This uses the values of metadata attributes (typically, source-level metadata).

Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method arguments as well as static information. This means that they must be evaluated with every method invocation and that the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

Control Flow Pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ `cflow` pointcuts, although less powerful. (There is currently no way to specify that a pointcut runs below a join point matched by

another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the join point was invoked by a method in the `com.mycompany.web` package or by the `SomeCaller` class. Control flow pointcuts are specified by using the `org.springframework.aop.support.ControlFlowPointcut` class.

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about five times that of other dynamic pointcuts.

Pointcut Superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you should probably subclass `StaticMethodMatcherPointcut`. This requires implementing only one abstract method (although you can override other methods to customize behavior). The following example shows how to subclass `StaticMethodMatcherPointcut`:

Java

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

Kotlin

```
class TestStaticPointcut : StaticMethodMatcherPointcut() {

    override fun matches(method: Method, targetClass: Class<*>): Boolean {
        // return true if custom criteria match
    }
}
```

There are also superclasses for dynamic pointcuts. You can use custom pointcuts with any advice type.

Custom Pointcuts

Because pointcuts in Spring AOP are Java classes rather than language features (as in AspectJ), you can declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, we recommend using the AspectJ pointcut expression language, if you can.

Later versions of Spring may offer support for “semantic pointcuts” as offered by JAC — for example, “all methods that change instance variables in the target object.”

2.6.2. Advice API in Spring

Now we can examine how Spring AOP handles advice.

Advice Lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects or be unique to each advised object. This corresponds to per-class or per-instance advice.

Per-class advice is used most often. It is appropriate for generic advice, such as transaction advisors. These do not depend on the state of the proxied object or add new state. They merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

You can use a mix of shared and per-instance advice in the same AOP proxy.

Advice Types in Spring

Spring provides several advice types and is extensible to support arbitrary advice types. This section describes the basic concepts and standard advice types.

Interception Around Advice

The most fundamental advice type in Spring is interception around advice.

Spring is compliant with the AOP `Alliance` interface for around advice that uses method interception. Classes that implement `MethodInterceptor` and that implement around advice should also implement the following interface:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

The `MethodInvocation` argument to the `invoke()` method exposes the method being invoked, the target join point, the AOP proxy, and the arguments to the method. The `invoke()` method should return the invocation’s result: the return value of the join point.

The following example shows a simple `MethodInterceptor` implementation:

Java

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

Kotlin

```
class DebugInterceptor : MethodInterceptor {

    override fun invoke(invocation: MethodInvocation): Any {
        println("Before: invocation=[$invocation]")
        val rval = invocation.proceed()
        println("Invocation returned")
        return rval
    }
}
```

Note the call to the `proceed()` method of `MethodInvocation`. This proceeds down the interceptor chain towards the join point. Most interceptors invoke this method and return its return value. However, a `MethodInterceptor`, like any around advice, can return a different value or throw an exception rather than invoke the proceed method. However, you do not want to do this without good reason.

`MethodInterceptor` implementations offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice types discussed in the remainder of this section implement common AOP concepts but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with `MethodInterceptor` around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces.

Before Advice

A simpler advice type is a before advice. This does not need a `MethodInvocation` object, since it is called only before entering the method.

The main advantage of a before advice is that there is no need to invoke the `proceed()` method and, therefore, no possibility of inadvertently failing to proceed down the interceptor chain.

The following listing shows the `MethodBeforeAdvice` interface:

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

(Spring's API design would allow for field before advice, although the usual objects apply to field interception and it is unlikely for Spring to ever implement it.)

Note that the return type is `void`. Before advice can insert custom behavior before the join point runs but cannot change the return value. If a before advice throws an exception, it stops further execution of the interceptor chain. The exception propagates back up the interceptor chain. If it is unchecked or on the signature of the invoked method, it is passed directly to the client. Otherwise, it is wrapped in an unchecked exception by the AOP proxy.

The following example shows a before advice in Spring, which counts all method invocations:

Java

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {

    private int count;

    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

Kotlin

```
class CountingBeforeAdvice : MethodBeforeAdvice {

    var count: Int = 0

    override fun before(m: Method, args: Array<Any>, target: Any?) {
        ++count
    }
}
```

Before advice can be used with any pointcut.

Throws Advice

Throws advice is invoked after the return of the join point if the join point threw an exception. Spring offers typed throws advice. Note that this means that the

`org.springframework.aop.ThrowsAdvice` interface does not contain any methods. It is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be in the following form:

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The next two listing show classes that are examples of throws advice.

The following advice is invoked if a `RemoteException` is thrown (including from subclasses):

Java

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

Kotlin

```
class RemoteThrowsAdvice : ThrowsAdvice {

    fun afterThrowing(ex: RemoteException) {
        // Do something with remote exception
    }
}
```

Unlike the preceding advice, the next example declares four arguments, so that it has access to the invoked method, method arguments, and target object. The following advice is invoked if a `ServletException` is thrown:

Java

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException
ex) {
        // Do something with all arguments
    }
}
```

Kotlin

```
class ServletThrowsAdviceWithArguments : ThrowsAdvice {

    fun afterThrowing(m: Method, args: Array<Any>, target: Any, ex: ServletException)
{
        // Do something with all arguments
    }
}
```

The final example illustrates how these two methods could be used in a single class that handles both `RemoteException` and `ServletException`. Any number of throws advice methods can be combined in a single class. The following listing shows the final example:

Java

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException
ex) {
        // Do something with all arguments
    }
}
```

Kotlin

```
class CombinedThrowsAdvice : ThrowsAdvice {

    fun afterThrowing(ex: RemoteException) {
        // Do something with remote exception
    }

    fun afterThrowing(m: Method, args: Array<Any>, target: Any, ex: ServletException)
{
        // Do something with all arguments
    }
}
```

If a throws-advice method throws an exception itself, it overrides the original exception (that is, it changes the exception thrown to the user). The overriding exception is typically a RuntimeException, which is compatible with any method signature. However, if a throws-advice method throws a checked exception, it must match the declared exceptions of the target method and is, hence, to some degree coupled to specific target method signatures. *Do not throw an undeclared checked exception that is incompatible with the target method's signature!*

Throws advice can be used with any pointcut.

After Returning Advice

An after returning advice in Spring must implement the `org.springframework.aop.AfterReturningAdvice` interface, which the following listing shows:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

An after returning advice has access to the return value (which it cannot modify), the invoked method, the method's arguments, and the target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

Java

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object
target)
            throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

Kotlin

```
class CountingAfterReturningAdvice : AfterReturningAdvice {

    var count: Int = 0
        private set

    override fun afterReturning(returnValue: Any?, m: Method, args: Array<Any>,
target: Any?) {
        ++count
    }
}
```

This advice does not change the execution path. If it throws an exception, it is thrown up the interceptor chain instead of the return value.

After returning advice can be used with any pointcut.

Introduction Advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an `IntroductionAdvisor` and an `IntroductionInterceptor` that implement the following interface:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

The `invoke()` method inherited from the AOP Alliance `MethodInterceptor` interface must implement the introduction. That is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call — it cannot invoke `proceed()`.

Introduction advice cannot be used with any pointcut, as it applies only at the class, rather than the method, level. You can only use introduction advice with the `IntroductionAdvisor`, which has the following methods:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

    ClassFilter getClassFilter();

    void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

    Class<?>[] getInterfaces();
}
```

There is no `MethodMatcher` and, hence, no `Pointcut` associated with introduction advice. Only class filtering is logical.

The `getInterfaces()` method returns the interfaces introduced by this advisor.

The `validateInterfaces()` method is used internally to see whether or not the introduced interfaces can be implemented by the configured `IntroductionInterceptor`.

Consider an example from the Spring test suite and suppose we want to introduce the following interface to one or more objects:

Java

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

Kotlin

```
interface Lockable {
    fun lock()
    fun unlock()
    fun locked(): Boolean
}
```

This illustrates a mixin. We want to be able to cast advised objects to `Lockable`, whatever their type and call lock and unlock methods. If we call the `lock()` method, we want all setter methods to throw a `LockedException`. Thus, we can add an aspect that provides the ability to make objects immutable without them having any knowledge of it: a good example of AOP.

First, we need an `IntroductionInterceptor` that does the heavy lifting. In this case, we extend the `org.springframework.aop.support.DelegatingIntroductionInterceptor` convenience class. We could implement `IntroductionInterceptor` directly, but using `DelegatingIntroductionInterceptor` is best for most cases.

The `DelegatingIntroductionInterceptor` is designed to delegate an introduction to an actual implementation of the introduced interfaces, concealing the use of interception to do so. You can set the delegate to any object using a constructor argument. The default delegate (when the no-argument constructor is used) is `this`. Thus, in the next example, the delegate is the `LockMixin` subclass of `DelegatingIntroductionInterceptor`. Given a delegate (by default, itself), a `DelegatingIntroductionInterceptor` instance looks for all interfaces implemented by the delegate (other than `IntroductionInterceptor`) and supports introductions against any of them. Subclasses such as `LockMixin` can call the `suppressInterface(Class intf)` method to suppress interfaces that should not be exposed. However, no matter how many interfaces an `IntroductionInterceptor` is prepared to support, the `IntroductionAdvisor` used controls which interfaces are actually exposed. An introduced interface conceals any implementation of the same interface by the target.

Thus, `LockMixin` extends `DelegatingIntroductionInterceptor` and implements `Lockable` itself. The superclass automatically picks up that `Lockable` can be supported for introduction, so we do not need to specify that. We could introduce any number of interfaces in this way.

Note the use of the `locked` instance variable. This effectively adds additional state to that held in the target object.

The following example shows the example `LockMixin` class:

Java

```
public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {
            throw new LockedException();
        }
        return super.invoke(invocation);
    }

}
```

Kotlin

```
class LockMixin : DelegatingIntroductionInterceptor(), Lockable {

    private var locked: Boolean = false

    fun lock() {
        this.locked = true
    }

    fun unlock() {
        this.locked = false
    }

    fun locked(): Boolean {
        return this.locked
    }

    override fun invoke(invocation: MethodInvocation): Any? {
        if (locked() && invocation.method.name.indexOf("set") == 0) {
            throw LockedException()
        }
        return super.invoke(invocation)
    }

}
```

Often, you need not override the `invoke()` method. The `DelegatingIntroductionInterceptor` implementation (which calls the `delegate` method if the method is introduced, otherwise proceeds towards the join point) usually suffices. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The required introduction only needs to hold a distinct `LockMixin` instance and specify the introduced interfaces (in this case, only `Lockable`). A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype). In this case, there is no configuration relevant for a `LockMixin`, so we create it by using `new`. The following example shows our `LockMixinAdvisor` class:

Java

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

Kotlin

```
class LockMixinAdvisor : DefaultIntroductionAdvisor(LockMixin(), Lockable::class.java)
```

We can apply this advisor very simply, because it requires no configuration. (However, it is impossible to use an `IntroductionInterceptor` without an `IntroductionAdvisor`.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different instance of `LockMixinAdvisor`, and hence `LockMixin`, for each advised object. The advisor comprises part of the advised object's state.

We can apply this advisor programmatically by using the `Advised.addAdvisor()` method or (the recommended way) in XML configuration, as any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

2.6.3. The Advisor API in Spring

In Spring, an Advisor is an aspect that contains only a single advice object associated with a pointcut expression.

Apart from the special case of introductions, any advisor can be used with any advice. `org.springframework.aop.support.DefaultPointcutAdvisor` is the most commonly used advisor class. It can be used with a `MethodInterceptor`, `BeforeAdvice`, or `ThrowsAdvice`.

It is possible to mix advisor and advice types in Spring in the same AOP proxy. For example, you could use an interception around advice, throws advice, and before advice in one proxy configuration. Spring automatically creates the necessary interceptor chain.

2.6.4. Using the `ProxyFactoryBean` to Create AOP Proxies

If you use the Spring IoC container (an `ApplicationContext` or `BeanFactory`) for your business objects (and you should be!), you want to use one of Spring's AOP `FactoryBean` implementations. (Remember that a factory bean introduces a layer of indirection, letting it create objects of a different type.)

The Spring AOP support also uses factory beans under the covers.

The basic way to create an AOP proxy in Spring is to use the `org.springframework.aop.framework.ProxyFactoryBean`. This gives complete control over the pointcuts, any advice that applies, and their ordering. However, there are simpler options that are preferable if you do not need such control.

Basics

The `ProxyFactoryBean`, like other Spring `FactoryBean` implementations, introduces a level of indirection. If you define a `ProxyFactoryBean` named `foo`, objects that reference `foo` do not see the `ProxyFactoryBean` instance itself but an object created by the implementation of the `getObject()` method in the `ProxyFactoryBean` . This method creates an AOP proxy that wraps a target object.

One of the most important benefits of using a `ProxyFactoryBean` or another IoC-aware class to create AOP proxies is that advices and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

JavaBean Properties

In common with most `FactoryBean` implementations provided with Spring, the `ProxyFactoryBean` class is itself a JavaBean. Its properties are used to:

- Specify the target you want to proxy.
- Specify whether to use CGLIB (described later and see also JDK- and CGLIB-based proxies).

Some key properties are inherited from `org.springframework.aop.framework.ProxyConfig` (the superclass for all AOP proxy factories in Spring). These key properties include the following:

- `proxyTargetClass`: `true` if the target class is to be proxied, rather than the target class's interfaces. If this property value is set to `true`, then CGLIB proxies are created (but see also JDK- and CGLIB-based proxies).
- `optimize`: Controls whether or not aggressive optimizations are applied to proxies created through CGLIB. You should not blithely use this setting unless you fully understand how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies. It has no effect with JDK dynamic proxies.
- `frozen`: If a proxy configuration is `frozen`, changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you do not want callers to be able to manipulate the proxy (through the `Advised` interface) after the proxy has been created. The default value of this property is `false`, so changes (such as adding additional advice) are allowed.
- `exposeProxy`: Determines whether or not the current proxy should be exposed in a `ThreadLocal` so that it can be accessed by the target. If a target needs to obtain the proxy and the `exposeProxy` property is set to `true`, the target can use the `AopContext.currentProxy()` method.

Other properties specific to `ProxyFactoryBean` include the following:

- `proxyInterfaces`: An array of `String` interface names. If this is not supplied, a CGLIB proxy for the target class is used (but see also JDK- and CGLIB-based proxies).
- `interceptorNames`: A `String` array of `Advisor`, interceptor, or other advice names to apply. Ordering is significant, on a first come-first served basis. That is to say that the first interceptor in the list is the first to be able to intercept the invocation.

  The names are bean names in the current factory, including bean names from ancestor factories. You cannot mention bean references here, since doing so results in the `ProxyFactoryBean` ignoring the singleton setting of the advice.

  You can append an interceptor name with an asterisk (`*`). Doing so results in the application of all advisor beans with names that start with the part before the asterisk to be applied. You can find an example of using this feature in Using "Global" Advisors.

- singleton: Whether or not the factory should return a single object, no matter how often the `getObject()` method is called. Several `FactoryBean` implementations offer such a method. The default value is `true`. If you want to use stateful advice - for example, for stateful mixins - use prototype advices along with a singleton value of `false`.

JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the `ProxyFactoryBean` chooses to create either a JDK-based proxy or a CGLIB-based proxy for a particular target object (which is to be proxied).

The behavior of the `ProxyFactoryBean` with regard to creating JDK- or CGLIB-based proxies changed between versions 1.2.x and 2.0 of Spring. The `ProxyFactoryBean` now exhibits similar semantics with regard to auto-detecting interfaces as those of the `TransactionProxyFactoryBean` class.

If the class of a target object that is to be proxied (hereafter simply referred to as the target class) does not implement any interfaces, a CGLIB-based proxy is created. This is the easiest scenario, because JDK proxies are interface-based, and no interfaces means JDK proxying is not even possible. You can plug in the target bean and specify the list of interceptors by setting the `interceptorNames` property. Note that a CGLIB-based proxy is created even if the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `false`. (Doing so makes no sense and is best removed from the bean definition, because it is, at best, redundant, and, at worst confusing.)

If the target class implements one (or more) interfaces, the type of proxy that is created depends on the configuration of the `ProxyFactoryBean`.

If the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `true`, a CGLIB-based proxy is created. This makes sense and is in keeping with the principle of least surprise. Even if the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, the fact that the `proxyTargetClass` property is set to `true` causes CGLIB-based proxying to be in effect.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, a JDK-based proxy is created. The created proxy implements all of the interfaces that were specified in the `proxyInterfaces` property. If the target class happens to implement a whole lot more interfaces than those specified in the `proxyInterfaces` property, that is all well and good, but those additional interfaces are not implemented by the returned proxy.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has not been set, but the target class does implement one (or more) interfaces, the `ProxyFactoryBean` auto-detects the fact that the target class does actually implement at least one interface, and a JDK-based proxy is created. The interfaces that are actually proxied are all of the interfaces that the target class implements. In effect, this is the same as supplying a list of each and every interface that the target class implements to the `proxyInterfaces` property. However, it is significantly less work and less prone to typographical errors.

Proxying Interfaces

Consider a simple example of `ProxyFactoryBean` in action. This example involves:

- A target bean that is proxied. This is the `personTarget` bean definition in the example.
- An `Advisor` and an `Interceptor` used to provide advice.
- An AOP proxy bean definition to specify the target object (the `personTarget` bean), the interfaces to proxy, and the advices to apply.

The following listing shows the example:

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name" value="Tony"/>
    <property name="age" value="51"/>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor"
class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>

    <property name="target" ref="personTarget"/>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

Note that the `interceptorNames` property takes a list of `String`, which holds the bean names of the interceptors or advisors in the current factory. You can use advisors, interceptors, before, after returning, and throws advice objects. The ordering of advisors is significant.

You might be wondering why the list does not hold bean references. The reason for this is that, if the singleton property of the `ProxyFactoryBean` is set to `false`, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it is necessary to be able to obtain an instance of the prototype from the factory. Holding a reference is not sufficient.

The `person` bean definition shown earlier can be used in place of a `Person` implementation, as follows:

Java

```
Person person = (Person) factory.getBean("person");
```

Kotlin

```
val person = factory.getBean("person") as Person;
```

Other beans in the same IoC context can express a strongly typed dependency on it, as with an ordinary Java object. The following example shows how to do so:

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref bean="person"/></property>
</bean>
```

The `PersonUser` class in this example exposes a property of type `Person`. As far as it is concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its class would be a dynamic proxy class. It would be possible to cast it to the `Advised` interface (discussed later).

You can conceal the distinction between target and proxy by using an anonymous inner bean. Only the `ProxyFactoryBean` definition is different. The advice is included only for completeness. The following example shows how to use an anonymous inner bean:

```
<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor"
class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>
    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name" value="Tony"/>
            <property name="age" value="51"/>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

Using an anonymous inner bean has the advantage that there is only one object of type `Person`. This is useful if we want to prevent users of the application context from obtaining a reference to the un-advised object or need to avoid any ambiguity with Spring IoC autowiring. There is also, arguably, an advantage in that the `ProxyFactoryBean` definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an advantage (for example, in certain test scenarios).

Proxying Classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our earlier example, there was no `Person` interface. We needed to advise a class called `Person` that did not implement any business interface. In this case, you can configure Spring to use CGLIB proxying rather than dynamic proxies. To do so, set the `proxyTargetClass` property on the `ProxyFactoryBean` shown earlier to `true`. While it is best to program to interfaces rather than classes, the ability to advise classes that do not implement interfaces can be useful when working with legacy code. (In general, Spring is not prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach.)

If you want to, you can force the use of CGLIB in any case, even if you do have interfaces.

CGLIB proxying works by generating a subclass of the target class at runtime. Spring configures this generated subclass to delegate method calls to the original target. The subclass is used to implement the Decorator pattern, weaving in the advice.

CGLIB proxying should generally be transparent to users. However, there are some issues to consider:

- `Final` methods cannot be advised, as they cannot be overridden.
- There is no need to add CGLIB to your classpath. As of Spring 3.2, CGLIB is repackaged and included in the spring-core JAR. In other words, CGLIB-based AOP works “out of the box”, as do JDK dynamic proxies.

There is little performance difference between CGLIB proxying and dynamic proxies. Performance should not be a decisive consideration in this case.

Using “Global” Advisors

By appending an asterisk to an interceptor name, all advisors with bean names that match the part before the asterisk are added to the advisor chain. This can come in handy if you need to add a standard set of “global” advisors. The following example defines two global advisors:

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="service"/>
    <property name="interceptorNames">
        <list>
            <value>global*</value>
        </list>
    </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance"
class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

2.6.5. Concise Proxy Definitions

Especially when defining transactional proxies, you may end up with many similar proxy definitions. The use of parent and child bean definitions, along with inner bean definitions, can result in much cleaner and more concise proxy definitions.

First, we create a parent, template, bean definition for the proxy, as follows:

```
<bean id="txProxyTemplate" abstract="true"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

This is never instantiated itself, so it can actually be incomplete. Then, each proxy that needs to be created is a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target is never used on its own anyway. The following example shows such a child bean:

```
<bean id="myService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MyServiceImpl">
        </bean>
    </property>
</bean>
```

You can override properties from the parent template. In the following example, we override the transaction propagation settings:

```
<bean id="mySpecialService" parent="txProxyTemplate">
    <property name="target">
        <bean class="org.springframework.samples.MySpecialServiceImpl">
        </bean>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="store*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

Note that in the parent bean example, we explicitly marked the parent bean definition as being abstract by setting the `abstract` attribute to `true`, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple bean factories), by default, pre-instantiate all singletons. Therefore, it is important (at least for singleton beans) that, if you have a (parent) bean definition that you intend to use only as a template, and this definition specifies a class, you must make sure to set the `abstract` attribute to `true`. Otherwise, the application context actually tries to pre-instantiate it.

2.6.6. Creating AOP Proxies Programmatically with the `ProxyFactory`

It is easy to create AOP proxies programmatically with Spring. This lets you use Spring AOP without dependency on Spring IoC.

The interfaces implemented by the target object are automatically proxied. The following listing shows creation of a proxy for a target object, with one interceptor and one advisor:

Java

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addAdvice(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

Kotlin

```
val factory = ProxyFactory(myBusinessInterfaceImpl)
factory.addAdvice(myMethodInterceptor)
factory.addAdvisor(myAdvisor)
val tb = factory.proxy as MyBusinessInterface
```

The first step is to construct an object of type `org.springframework.aop.framework.ProxyFactory`. You can create this with a target object, as in the preceding example, or specify the interfaces to be proxied in an alternate constructor.

You can add advices (with interceptors as a specialized kind of advice), advisors, or both and manipulate them for the life of the `ProxyFactory`. If you add an `IntroductionInterceptionAroundAdvisor`, you can cause the proxy to implement additional interfaces.

There are also convenience methods on `ProxyFactory` (inherited from `AdvisedSupport`) that let you add other advice types, such as before and throws advice. `AdvisedSupport` is the superclass of both `ProxyFactory` and `ProxyFactoryBean`.

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from Java code with AOP, as you should in general.

2.6.7. Manipulating Advised Objects

However you create AOP proxies, you can manipulate them BY using the `org.springframework.aop.framework.Advised` interface. Any AOP proxy can be cast to this interface, no matter which other interfaces it implements. This interface includes the following methods:

Java

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

Kotlin

```
fun getAdvisors(): Array<Advisor>

@Throws(AopConfigException::class)
fun addAdvice(advice: Advice)

@Throws(AopConfigException::class)
fun addAdvice(pos: Int, advice: Advice)

@Throws(AopConfigException::class)
fun addAdvisor(advisor: Advisor)

@Throws(AopConfigException::class)
fun addAdvisor(pos: Int, advisor: Advisor)

fun indexOf(advisor: Advisor): Int

@Throws(AopConfigException::class)
fun removeAdvisor(advisor: Advisor): Boolean

@Throws(AopConfigException::class)
fun removeAdvisor(index: Int)

@Throws(AopConfigException::class)
fun replaceAdvisor(a: Advisor, b: Advisor): Boolean

fun isFrozen(): Boolean
```

The `getAdvisors()` method returns an `Advisor` for every advisor, interceptor, or other advice type that has been added to the factory. If you added an `Advisor`, the returned advisor at this index is the object that you added. If you added an interceptor or other advice type, Spring wrapped this in an advisor with a pointcut that always returns `true`. Thus, if you added a `MethodInterceptor`, the advisor returned for this index is a `DefaultPointcutAdvisor` that returns your `MethodInterceptor` and a pointcut that matches all classes and methods.

The `addAdvisor()` methods can be used to add any `Advisor`. Usually, the advisor holding pointcut and advice is the generic `DefaultPointcutAdvisor`, which you can use with any advice or pointcut (but not for introductions).

By default, it is possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it is impossible to add or remove an introduction advisor, as existing proxies from the factory do not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

The following example shows casting an AOP proxy to the `Advised` interface and examining and manipulating its advice:

Java

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);
```

Kotlin

```
val advised = myObject as Advised
val advisors = advised.advisors
val oldAdvisorCount = advisors.size
println("$oldAdvisorCount advisors")

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(DebugInterceptor())

// Add selective advice using a pointcut
advised.addAdvisor(DefaultPointcutAdvisor(mySpecialPointcut, myAdvice))

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.advisors.size)
```

It is questionable whether it is advisable (no pun intended) to modify advice on a business object in production, although there are, no doubt, legitimate usage cases. However, it can be very useful in development (for example, in tests). We have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation that we want to test. (For example, the advice can get inside a transaction created for that method, perhaps to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a `frozen` flag. In that case, the `Advised` `isFrozen()` method returns `true`, and any attempts to modify advice through addition or removal results in an `AopConfigException`. The ability to freeze the state of an advised object is useful in some cases (for example, to prevent calling code removing a security interceptor).

2.6.8. Using the "auto-proxy" facility

So far, we have considered explicit creation of AOP proxies by using a `ProxyFactoryBean` or similar factory bean.

Spring also lets us use “auto-proxy” bean definitions, which can automatically proxy selected bean definitions. This is built on Spring’s “bean post processor” infrastructure, which enables modification of any bean definition as the container loads.

In this model, you set up some special bean definitions in your XML bean definition file to configure the auto-proxy infrastructure. This lets you declare the targets eligible for auto-proxying. You need not use `ProxyFactoryBean`.

There are two ways to do this:

- By using an auto-proxy creator that refers to specific beans in the current context.
- A special case of auto-proxy creation that deserves to be considered separately: auto-proxy creation driven by source-level metadata attributes.

Auto-proxy Bean Definitions

This section covers the auto-proxy creators provided by the `org.springframework.aop.framework.autoproxy` package.

`BeanNameAutoProxyCreator`

The `BeanNameAutoProxyCreator` class is a `BeanPostProcessor` that automatically creates AOP proxies for beans with names that match literal values or wildcards. The following example shows how to create a `BeanNameAutoProxyCreator` bean:

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames" value="jdk*,onlyJdk"/>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

As with `ProxyFactoryBean`, there is an `interceptorNames` property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

As with auto-proxying in general, the main point of using `BeanNameAutoProxyCreator` is to apply the same configuration consistently to multiple objects, with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as `jdkMyBean` and `onlyJdk` in the preceding example, are plain old bean definitions with the target class. An AOP proxy is automatically created by the `BeanNameAutoProxyCreator`. The same advice is applied to all matching beans. Note that, if advisors are used (rather than the interceptor in the preceding example), the pointcuts may apply differently to different beans.

`DefaultAdvisorAutoProxyCreator`

A more general and extremely powerful auto-proxy creator is `DefaultAdvisorAutoProxyCreator`. This automagically applies eligible advisors in the current context, without the need to include specific bean names in the auto-proxy advisor's bean definition. It offers the same merit of consistent configuration and avoidance of duplication as `BeanNameAutoProxyCreator`.

Using this mechanism involves:

- Specifying a `DefaultAdvisorAutoProxyCreator` bean definition.
- Specifying any number of advisors in the same or related contexts. Note that these must be advisors, not interceptors or other advices. This is necessary, because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate bean definitions.

The `DefaultAdvisorAutoProxyCreator` automatically evaluates the pointcut contained in each advisor, to see what (if any) advice it should apply to each business object (such as `businessObject1` and

`businessObject2` in the example).

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object is not proxied. As bean definitions are added for new business objects, they are automatically proxied if necessary.

Auto-proxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling `getBean("businessObject1")` on this `ApplicationContext` returns an AOP proxy, not the target business object. (The “inner bean” idiom shown earlier also offers this benefit.)

The following example creates a `DefaultAdvisorAutoProxyCreator` bean and the other elements discussed in this section:

```
<bean
class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean
class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
    <!-- Properties omitted -->
</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

The `DefaultAdvisorAutoProxyCreator` is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can add new business objects without including specific proxy configuration. You can also easily drop in additional aspects (for example, tracing or performance monitoring aspects) with minimal change to configuration.

The `DefaultAdvisorAutoProxyCreator` offers support for filtering (by using a naming convention so that only certain advisors are evaluated, which allows the use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the `org.springframework.core.Ordered` interface to ensure correct ordering if this is an issue. The `TransactionAttributeSourceAdvisor` used in the preceding example has a configurable order value. The default setting is unordered.

2.6.9. Using `TargetSource` Implementations

Spring offers the concept of a `TargetSource`, expressed in the `org.springframework.aop.TargetSource` interface. This interface is responsible for returning the “target object” that implements the join point. The `TargetSource` implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers who use Spring AOP do not normally need to work directly with `TargetSource` implementations, but this provides a powerful means of supporting pooling, hot swappable, and other sophisticated targets. For example, a pooling `TargetSource` can return a different target instance for each invocation, by using a pool to manage instances.

If you do not specify a `TargetSource`, a default implementation is used to wrap a local object. The same target is returned for each invocation (as you would expect).

The rest of this section describes the standard target sources provided with Spring and how you can use them.

When using a custom target source, your target will usually need to be a prototype rather than a singleton bean definition. This allows Spring to create a new target instance when required.

Hot-swappable Target Sources

The `org.springframework.aop.target.HotSwappableTargetSource` exists to let the target of an AOP proxy be switched while letting callers keep their references to it.

Changing the target source's target takes effect immediately. The `HotSwappableTargetSource` is thread-safe.

You can change the target by using the `swap()` method on HotSwappableTargetSource, as the follow example shows:

Java

```
HotSwappableTargetSource swapper = (HotSwappableTargetSource)
beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

Kotlin

```
val swapper = beanFactory.getBean("swapper") as HotSwappableTargetSource
val oldTarget = swapper.swap(newTarget)
```

The following example shows the required XML definitions:

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
    <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="swapper"/>
</bean>
```

The preceding `swap()` call changes the target of the swappable bean. Clients that hold a reference to that bean are unaware of the change but immediately start hitting the new target.

Although this example does not add any advice (it is not necessary to add advice to use a `TargetSource`), any `TargetSource` can be used in conjunction with arbitrary advice.

Pooling Target Sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring pooling and SLSB pooling is that Spring pooling can be applied to any POJO. As with Spring in general, this service can be applied in a non-invasive way.

Spring provides support for Commons Pool 2.2, which provides a fairly efficient pooling implementation. You need the `commons-pool` Jar on your application's classpath to use this feature. You can also subclass `org.springframework.aop.target.AbstractPoolingTargetSource` to support any other pooling API.

Commons Pool 1.5+ is also supported but is deprecated as of Spring Framework 4.2.

The following listing shows an example configuration:

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
        scope="prototype">
    ... properties omitted
</bean>

<bean id="poolTargetSource"
class="org.springframework.aop.target.CommonsPool2TargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
    <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="poolTargetSource"/>
    <property name="interceptorNames" value="myInterceptor"/>
</bean>
```

Note that the target object (`businessObjectTarget` in the preceding example) must be a prototype. This lets the `PoolingTargetSource` implementation create new instances of the target to grow the pool as necessary. See the javadoc of `AbstractPoolingTargetSource` and the concrete subclass you wish to use for information about its properties. `maxSize` is the most basic and is always guaranteed to be present.

In this case, `myInterceptor` is the name of an interceptor that would need to be defined in the same IoC context. However, you need not specify interceptors to use pooling. If you want only pooling

and no other advice, do not set the `interceptorNames` property at all.

You can configure Spring to be able to cast any pooled object to the `org.springframework.aop.target.PoolingConfig` interface, which exposes information about the configuration and current size of the pool through an introduction. You need to define an advisor similar to the following:

```
<bean id="poolConfigAdvisor"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="poolTargetSource"/>
    <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>
```

This advisor is obtained by calling a convenience method on the `AbstractPoolingTargetSource` class, hence the use of `MethodInvokingFactoryBean`. This advisor's name (`poolConfigAdvisor`, here) must be in the list of interceptors names in the `ProxyFactoryBean` that exposes the pooled object.

The cast is defined as follows:

Java

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

Kotlin

```
val conf = beanFactory.getBean("businessObject") as PoolingConfig
println("Max pool size is " + conf.maxSize)
```

Pooling stateless service objects is not usually necessary. We do not believe it should be the default choice, as most stateless objects are naturally thread safe, and instance pooling is problematic if resources are cached.

Simpler pooling is available by using auto-proxying. You can set the `TargetSource` implementations used by any auto-proxy creator.

Prototype Target Sources

Setting up a "prototype" target source is similar to setting up a pooling `TargetSource`. In this case, a new instance of the target is created on every method invocation. Although the cost of creating a new object is not high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus, you should not use this approach without very good reason.

To do this, you could modify the `poolTargetSource` definition shown earlier as follows (we also changed the name, for clarity):

```
<bean id="prototypeTargetSource"
class="org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

The only property is the name of the target bean. Inheritance is used in the `TargetSource` implementations to ensure consistent naming. As with the pooling target source, the target bean must be a prototype bean definition.

`ThreadLocal` Target Sources

`ThreadLocal` target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a `ThreadLocal` provides a JDK-wide facility to transparently store a resource alongside a thread. Setting up a `ThreadLocalTargetSource` is pretty much the same as was explained for the other types of target source, as the following example shows:

```
<bean id="threadlocalTargetSource"
class="org.springframework.aop.target.ThreadLocalTargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

`ThreadLocal` instances come with serious issues (potentially resulting in memory leaks) when incorrectly using them in multi-threaded and multi-classloader environments. You should always consider wrapping a threadlocal in some other class and never directly use the `ThreadLocal` itself (except in the wrapper class). Also, you should always remember to correctly set and unset (where the latter simply involves a call to `ThreadLocal.set(null)`) the resource local to the thread. Unsetting should be done in any case, since not unsetting it might result in problematic behavior. Spring's `ThreadLocal` support does this for you and should always be considered in favor of using `ThreadLocal` instances without other proper handling code.

2.6.10. Defining New Advice Types

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to the interception around advice, before, throws advice, and after returning advice.

The `org.springframework.aop.framework.adapter` package is an SPI package that lets support for new custom advice types be added without changing the core framework. The only constraint on a custom `Advice` type is that it must implement the `org.aopalliance.aop.Advice` marker interface.

See the `org.springframework.aop.framework.adapter` javadoc for further information.

2.7. Null-safety

Although Java does not let you express null-safety with its type system, the Spring Framework now provides the following annotations in the `org.springframework.lang` package to let you declare nullability of APIs and fields:

- `@Nullable`: Annotation to indicate that a specific parameter, return value, or field can be `null`.
- `@NonNull`: Annotation to indicate that a specific parameter, return value, or field cannot be `null` (not needed on parameters / return values and fields where `@NonNullApi` and `@NonNullFields` apply, respectively).
- `@NonNullApi`: Annotation at the package level that declares non-null as the default semantics for parameters and return values.
- `@NonNullFields`: Annotation at the package level that declares non-null as the default semantics for fields.

The Spring Framework itself leverages these annotations, but they can also be used in any Spring-based Java project to declare null-safe APIs and optionally null-safe fields. Generic type arguments, varargs and array elements nullability are not supported yet but should be in an upcoming release, see SPR-15942 for up-to-date information. Nullability declarations are expected to be fine-tuned between Spring Framework releases, including minor ones. Nullability of types used inside method bodies is outside of the scope of this feature.

Other common libraries such as Reactor and Spring Data provide null-safe APIs that use a similar nullability arrangement, delivering a consistent overall experience for Spring application developers.

2.7.1. Use cases

In addition to providing an explicit declaration for Spring Framework API nullability, these annotations can be used by an IDE (such as IDEA or Eclipse) to provide useful warnings related to null-safety in order to avoid `NullPointerException` at runtime.

They are also used to make Spring API null-safe in Kotlin projects, since Kotlin natively supports null-safety. More details are available in the Kotlin support documentation.

2.7.2. JSR-305 meta-annotations

Spring annotations are meta-annotated with JSR 305 annotations (a dormant but wide-spread JSR). JSR-305 meta-annotations let tooling vendors like IDEA or Kotlin provide null-safety support in a generic way, without having to hard-code support for Spring annotations.

It is not necessary nor recommended to add a JSR-305 dependency to the project classpath to take advantage of Spring null-safe API. Only projects such as Spring-based libraries that use null-safety annotations in their codebase should add `com.google.code.findbugs:jsr305:3.0.2` with `compileOnly` Gradle configuration or Maven `provided` scope to avoid compile warnings.

2.8. Data Buffers and Codecs

Java NIO provides `ByteBuffer` but many libraries build their own byte buffer API on top, especially for network operations where reusing buffers and/or using direct buffers is beneficial for performance. For example Netty has the `ByteBuf` hierarchy, Undertow uses XNIO, Jetty uses pooled byte buffers with a callback to be released, and so on. The `spring-core` module provides a set of abstractions to work with various byte buffer APIs as follows:

- `DataBufferFactory` abstracts the creation of a data buffer.
- `DataBuffer` represents a byte buffer, which may be pooled.
- `DataBufferUtils` offers utility methods for data buffers.
- Codecs decode or encode data buffer streams into higher level objects.

2.8.1. `DataBufferFactory`

`DataBufferFactory` is used to create data buffers in one of two ways:

1. Allocate a new data buffer, optionally specifying capacity upfront, if known, which is more efficient even though implementations of `DataBuffer` can grow and shrink on demand.
2. Wrap an existing `byte[]` or `java.nio.ByteBuffer`, which decorates the given data with a `DataBuffer` implementation and that does not involve allocation.

Note that WebFlux applications do not create a `DataBufferFactory` directly but instead access it through the `ServerHttpResponse` or the `ClientHttpRequest` on the client side. The type of factory depends on the underlying client or server, e.g. `NettyDataBufferFactory` for Reactor Netty, `DefaultDataBufferFactory` for others.

2.8.2. `DataBuffer`

The `DataBuffer` interface offers similar operations as `java.nio.ByteBuffer` but also brings a few additional benefits some of which are inspired by the Netty `ByteBuf`. Below is a partial list of benefits:

- Read and write with independent positions, i.e. not requiring a call to `flip()` to alternate between read and write.
- Capacity expanded on demand as with `java.lang.StringBuilder`.
- Pooled buffers and reference counting via `PooledDataBuffer`.
- View a buffer as `java.nio.ByteBuffer`, `InputStream`, or `OutputStream`.
- Determine the index, or the last index, for a given byte.

2.8.3. `PooledDataBuffer`

As explained in the Javadoc for ByteBuffer, byte buffers can be direct or non-direct. Direct buffers may reside outside the Java heap which eliminates the need for copying for native I/O operations. That makes direct buffers particularly useful for receiving and sending data over a socket, but they're also more expensive to create and release, which leads to the idea of pooling buffers.

`PooledDataBuffer` is an extension of `DataBuffer` that helps with reference counting which is essential for byte buffer pooling. How does it work? When a `PooledDataBuffer` is allocated the reference count is at 1. Calls to `retain()` increment the count, while calls to `release()` decrement it. As long as the count is above 0, the buffer is guaranteed not to be released. When the count is decreased to 0, the pooled buffer can be released, which in practice could mean the reserved memory for the buffer is returned to the memory pool.

Note that instead of operating on `PooledDataBuffer` directly, in most cases it's better to use the convenience methods in `DataBufferUtils` that apply release or retain to a `DataBuffer` only if it is an instance of `PooledDataBuffer`.

2.8.4. `DataBufferUtils`

`DataBufferUtils` offers a number of utility methods to operate on data buffers:

- Join a stream of data buffers into a single buffer possibly with zero copy, e.g. via composite buffers, if that's supported by the underlying byte buffer API.
- Turn `InputStream` or NIO `Channel` into `Flux<DataBuffer>`, and vice versa a `Publisher<DataBuffer>` into `OutputStream` or NIO `Channel`.
- Methods to release or retain a `DataBuffer` if the buffer is an instance of `PooledDataBuffer`.
- Skip or take from a stream of bytes until a specific byte count.

2.8.5. Codecs

The `org.springframework.core.codec` package provides the following strategy interfaces:

- `Encoder` to encode `Publisher<T>` into a stream of data buffers.
- `Decoder` to decode `Publisher<DataBuffer>` into a stream of higher level objects.

The `spring-core` module provides `byte[]`, `ByteBuffer`, `DataBuffer`, `Resource`, and `String` encoder and decoder implementations. The `spring-web` module adds Jackson JSON, Jackson Smile, JAXB2, Protocol Buffers and other encoders and decoders. See Codecs in the WebFlux section.

2.8.6. Using `DataBuffer`

When working with data buffers, special care must be taken to ensure buffers are released since they may be pooled. We'll use codecs to illustrate how that works but the concepts apply more generally. Let's see what codecs must do internally to manage data buffers.

A `Decoder` is the last to read input data buffers, before creating higher level objects, and therefore it must release them as follows:

1. If a `Decoder` simply reads each input buffer and is ready to release it immediately, it can do so via `DataBufferUtils.release(dataBuffer)`.
2. If a `Decoder` is using `Flux` or `Mono` operators such as `flatMap`, `reduce`, and others that prefetch and cache data items internally, or is using operators such as `filter`, `skip`, and others that leave out items, then `doOnDiscard(DataBuffer.class, DataBufferUtils::release)` must be added to the

composition chain to ensure such buffers are released prior to being discarded, possibly also as a result of an error or cancellation signal.

3. If a `Decoder` holds on to one or more data buffers in any other way, it must ensure they are released when fully read, or in case of an error or cancellation signals that take place before the cached data buffers have been read and released.

Note that `DataBufferUtils#join` offers a safe and efficient way to aggregate a data buffer stream into a single data buffer. Likewise `skipUntilByteCount` and `takeUntilByteCount` are additional safe methods for decoders to use.

An `Encoder` allocates data buffers that others must read (and release). So an `Encoder` doesn't have much to do. However an `Encoder` must take care to release a data buffer if a serialization error occurs while populating the buffer with data. For example:

Java

```
DataBuffer buffer = factory.allocateBuffer();
boolean release = true;
try {
    // serialize and populate buffer..
    release = false;
}
finally {
    if (release) {
        DataBufferUtils.release(buffer);
    }
}
return buffer;
```

Kotlin

```
val buffer = factory.allocateBuffer()
var release = true
try {
    // serialize and populate buffer..
    release = false
} finally {
    if (release) {
        DataBufferUtils.release(buffer)
    }
}
return buffer
```

The consumer of an `Encoder` is responsible for releasing the data buffers it receives. In a WebFlux application, the output of the `Encoder` is used to write to the HTTP server response, or to the client HTTP request, in which case releasing the data buffers is the responsibility of the code writing to the server response, or to the client request.

Note that when running on Netty, there are debugging options for troubleshooting buffer leaks.

2.9. Logging

Since Spring Framework 5.0, Spring comes with its own Commons Logging bridge implemented in the `spring-jcl` module. The implementation checks for the presence of the Log4j 2.x API and the SLF4J 1.7 API in the classpath and uses the first one of those found as the logging implementation, falling back to the Java platform's core logging facilities (also known as *JUL* or `java.util.logging`) if neither Log4j 2.x nor SLF4J is available.

Put Log4j 2.x or Logback (or another SLF4J provider) in your classpath, without any extra bridges, and let the framework auto-adapt to your choice. For further information see the Spring Boot Logging Reference Documentation.

Spring's Commons Logging variant is only meant to be used for infrastructure logging purposes in the core framework and in extensions.

For logging needs within application code, prefer direct use of Log4j 2.x, SLF4J, or JUL.

A `Log` implementation may be retrieved via `org.apache.commons.logging.LogFactory` as in the following example.

Java

```
public class MyBean {
    private final Log log = LogFactory.getLog(getClass());
    // ...
}
```

Kotlin

```
class MyBean {
  private val log = LogFactory.getLog(javaClass)
  // ...
}
```

2.10. Ahead of Time Optimizations

This chapter covers Spring's Ahead of Time (AOT) optimizations.

For AOT support specific to integration tests, see Ahead of Time Support for Tests.

2.10.1. Introduction to Ahead of Time Optimizations

Spring's support for AOT optimizations is meant to inspect an `ApplicationContext` at build time and apply decisions and discovery logic that usually happens at runtime. Doing so allows building an application startup arrangement that is more straightforward and focused on a fixed set of features based mainly on the classpath and the `Environment`.

Applying such optimizations early implies the following restrictions:

- The classpath is fixed and fully defined at build time.
- The beans defined in your application cannot change at runtime, meaning:
  - `@Profile`, in particular profile-specific configuration needs to be chosen at build time.
  - Environment properties that impact the presence of a bean (`@Conditional`) are only considered at build time.

When these restrictions are in place, it becomes possible to perform ahead-of-time processing at build time and generate additional assets. A Spring AOT processed application typically generates:

- Java source code
- Bytecode (usually for dynamic proxies)
- `RuntimeHints` for the use of reflection, resource loading, serialization, and JDK proxies.

At the moment, AOT is focused on allowing Spring applications to be deployed as native images using GraalVM. We intend to support more JVM-based use cases in future generations.

2.10.2. AOT engine overview

The entry point of the AOT engine for processing an `ApplicationContext` arrangement is `ApplicationContextAotGenerator`. It takes care of the following steps, based on a `GenericApplicationContext` that represents the application to optimize and a `GenerationContext`:

- Refresh an `ApplicationContext` for AOT processing. Contrary to a traditional refresh, this version only creates bean definitions, not bean instances.
- Invoke the available `BeanFactoryInitializationAotProcessor` implementations and apply their contributions against the `GenerationContext`. For instance, a core implementation iterates over all candidate bean definitions and generates the necessary code to restore the state of the `BeanFactory`.

Once this process completes, the `GenerationContext` will have been updated with the generated code, resources, and classes that are necessary for the application to run. The `RuntimeHints` instance can also be used to generate the relevant GraalVM native image configuration files.

`ApplicationContextAotGenerator#processAheadOfTime` returns the class name of the `ApplicationContextInitializer` entry point that allows the context to be started with AOT optimizations.

Those steps are covered in greater detail in the sections below.

2.10.3. Refresh for AOT Processing

Refresh for AOT processing is supported on all `GenericApplicationContext` implementations. An application context is created with any number of entry points, usually in the form of `@Configuration`-annotated classes.

Let's look at a basic example:

```
@Configuration(proxyBeanMethods=false)
@ComponentScan
@Import({DataSourceConfiguration.class, ContainerConfiguration.class})
public class MyApplication {
}
```

Starting this application with the regular runtime involves a number of steps including classpath scanning, configuration class parsing, bean instantiation, and lifecycle callback handling. Refresh for AOT processing only applies a subset of what happens with a regular `refresh`. AOT processing can be triggered as follows:

```
RuntimeHints hints = new RuntimeHints();
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.register(MyApplication.class);
context.refreshForAotProcessing(hints);
```

In this mode, `BeanFactoryPostProcessor` implementations are invoked as usual. This includes configuration class parsing, import selectors, classpath scanning, etc. Such steps make sure that the `BeanRegistry` contains the relevant bean definitions for the application. If bean definitions are guarded by conditions (such as `@Profile`), these are discarded at this stage.

Because this mode does not actually create bean instances, `BeanPostProcessor` implementations are not invoked, except for specific variants that are relevant for AOT processing. These are:

- `MergedBeanDefinitionPostProcessor` implementations post-process bean definitions to extract additional settings, such as `init` and `destroy` methods.
- `SmartInstantiationAwareBeanPostProcessor` implementations determine a more precise bean type if necessary. This makes sure to create any proxy that will be required at runtime.

One this part completes, the `BeanFactory` contains the bean definitions that are necessary for the application to run. It does not trigger bean instantiation but allows the AOT engine to inspect the beans that will be created at runtime.

2.10.4. Bean Factory Initialization AOT Contributions

Components that want to participate in this step can implement the `BeanFactoryInitializationAotProcessor` interface. Each implementation can return an AOT contribution, based on the state of the bean factory.

An AOT contribution is a component that contributes generated code that reproduces a particular behavior. It can also contribute `RuntimeHints` to indicate the need for reflection, resource loading, serialization, or JDK proxies.

A `BeanFactoryInitializationAotProcessor` implementation can be registered in `META-INF/spring/aot.factories` with a key equal to the fully qualified name of the interface.

A `BeanFactoryInitializationAotProcessor` can also be implemented directly by a bean. In this mode, the bean provides an AOT contribution equivalent to the feature it provides with a regular runtime. Consequently, such a bean is automatically excluded from the AOT-optimized context.

If a bean implements the `BeanFactoryInitializationAotProcessor` interface, the bean and **all** of its dependencies will be initialized during AOT processing. We generally recommend that this interface is only implemented by infrastructure beans such as `BeanFactoryPostProcessor` which have limited dependencies and are already initialized early in the bean factory lifecycle. If such a bean is registered using an `@Bean` factory method, ensure the method is `static` so that its enclosing `@Configuration` class does not have to be initialized.

Bean Registration AOT Contributions

A core `BeanFactoryInitializationAotProcessor` implementation is responsible for collecting the necessary contributions for each candidate `BeanDefinition`. It does so using a dedicated `BeanRegistrationAotProcessor`.

This interface is used as follows:

- Implemented by a `BeanPostProcessor` bean, to replace its runtime behavior. For instance `AutowiredAnnotationBeanPostProcessor` implements this interface to generate code that injects members annotated with `@Autowired`.
- Implemented by a type registered in `META-INF/spring/aot.factories` with a key equal to the fully qualified name of the interface. Typically used when the bean definition needs to be tuned for specific features of the core framework.

If a bean implements the `BeanRegistrationAotProcessor` interface, the bean and **all** of its dependencies will be initialized during AOT processing. We generally recommend that this interface is only implemented by infrastructure beans such as `BeanFactoryPostProcessor` which have limited dependencies and are already initialized early in the bean factory lifecycle. If such a bean is registered using an `@Bean` factory method, ensure the method is `static` so that its enclosing `@Configuration` class does not have to be initialized.

If no `BeanRegistrationAotProcessor` handles a particular registered bean, a default implementation processes it. This is the default behavior, since tuning the generated code for a bean definition should be restricted to corner cases.

Taking our previous example, let's assume that `DataSourceConfiguration` is as follows:

Java

```
@Configuration(proxyBeanMethods = false)
public class DataSourceConfiguration {

    @Bean
    public SimpleDataSource dataSource() {
        return new SimpleDataSource();
    }

}
```

Since there isn't any particular condition on this class, `dataSourceConfiguration` and `dataSource` are identified as candidates. The AOT engine will convert the configuration class above to code similar to the following:

Java

```
/**
 * Bean definitions for {@link DataSourceConfiguration}
 */
public class DataSourceConfiguration__BeanDefinitions {
    /**
     * Get the bean definition for 'dataSourceConfiguration'
     */
    public static BeanDefinition getDataSourceConfigurationBeanDefinition() {
        Class<?> beanType = DataSourceConfiguration.class;
        RootBeanDefinition beanDefinition = new RootBeanDefinition(beanType);
        beanDefinition.setInstanceSupplier(DataSourceConfiguration::new);
        return beanDefinition;
    }

    /**
     * Get the bean instance supplier for 'dataSource'.
     */
    private static BeanInstanceSupplier<SimpleDataSource>
getDataSourceInstanceSupplier() {
        return
BeanInstanceSupplier.<SimpleDataSource>forFactoryMethod(DataSourceConfiguration.class,
"dataSource")
                .withGenerator((registeredBean) ->
registeredBean.getBeanFactory().getBean(DataSourceConfiguration.class).dataSource());
    }

    /**
     * Get the bean definition for 'dataSource'
     */
    public static BeanDefinition getDataSourceBeanDefinition() {
        Class<?> beanType = SimpleDataSource.class;
        RootBeanDefinition beanDefinition = new RootBeanDefinition(beanType);
        beanDefinition.setInstanceSupplier(getDataSourceInstanceSupplier());
        return beanDefinition;
    }
}
```

The exact code generated may differ depending on the exact nature of your bean definitions.

The generated code above creates bean definitions equivalent to the `@Configuration` class, but in a direct way and without the use of reflection if at all possible. There is a bean definition for `dataSourceConfiguration` and one for `dataSourceBean`. When a `datasource` instance is required, a `BeanInstanceSupplier` is called. This supplier invokes the `dataSource()` method on the `dataSourceConfiguration` bean.

2.10.5. Runtime Hints

Running an application as a native image requires additional information compared to a regular JVM runtime. For instance, GraalVM needs to know ahead of time if a component uses reflection. Similarly, classpath resources are not shipped in a native image unless specified explicitly. Consequently, if the application needs to load a resource, it must be referenced from the corresponding GraalVM native image configuration file.

The `RuntimeHints` API collects the need for reflection, resource loading, serialization, and JDK proxies at runtime. The following example makes sure that `config/app.properties` can be loaded from the classpath at runtime within a native image:

Java

```
runtimeHints.resources().registerPattern("config/app.properties");
```

A number of contracts are handled automatically during AOT processing. For instance, the return type of a `@Controller` method is inspected, and relevant reflection hints are added if Spring detects that the type should be serialized (typically to JSON).

For cases that the core container cannot infer, you can register such hints programmatically. A number of convenient annotations are also provided for common use cases.

`@ImportRuntimeHints`

`RuntimeHintsRegistrar` implementations allow you to get a callback to the `RuntimeHints` instance managed by the AOT engine. Implementations of this interface can be registered using `@ImportRuntimeHints` on any Spring bean or `@Bean` factory method. `RuntimeHintsRegistrar` implementations are detected and invoked at build time.

```
/*
 * Copyright 2002-2022 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.docs.core.aot.hints.importruntimehints;

import java.util.Locale;

import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.hint.RuntimeHintsRegistrar;
import org.springframework.context.annotation.ImportRuntimeHints;
import org.springframework.core.io.ClassPathResource;
import org.springframework.stereotype.Component;

@Component
@ImportRuntimeHints(SpellCheckService.SpellCheckServiceRuntimeHints.class)
public class SpellCheckService {

    public void loadDictionary(Locale locale) {
        ClassPathResource resource = new ClassPathResource("dicts/" +
locale.getLanguage() + ".txt");
        //...
    }

    static class SpellCheckServiceRuntimeHints implements RuntimeHintsRegistrar {

        @Override
        public void registerHints(RuntimeHints hints, ClassLoader classLoader) {
            hints.resources().registerPattern("dicts/*");
        }
    }

}
```

If at all possible, `@ImportRuntimeHints` should be used as close as possible to the component that requires the hints. This way, if the component is not contributed to the `BeanFactory`, the hints won't be contributed either.

It is also possible to register an implementation statically by adding an entry in `META-INF/spring/aot.factories` with a key equal to the fully qualified name of the `RuntimeHintsRegistrar` interface.

`@Reflective`

`@Reflective` provides an idiomatic way to flag the need for reflection on an annotated element. For instance, `@EventListener` is meta-annotated with `@Reflective` since the underlying implementation invokes the annotated method using reflection.

By default, only Spring beans are considered and an invocation hint is registered for the annotated element. This can be tuned by specifying a custom `ReflectiveProcessor` implementation via the `@Reflective` annotation.

Library authors can reuse this annotation for their own purposes. If components other than Spring beans need to be processed, a `BeanFactoryInitializationAotProcessor` can detect the relevant types and use `ReflectiveRuntimeHintsRegistrar` to process them.

`@RegisterReflectionForBinding`

`@RegisterReflectionForBinding` is a specialization of `@Reflective` that registers the need for serializing arbitrary types. A typical use case is the use of DTOs that the container cannot infer, such as using a web client within a method body.

`@RegisterReflectionForBinding` can be applied to any Spring bean at the class level, but it can also be applied directly to a method, field, or constructor to better indicate where the hints are actually required. The following example registers `Account` for serialization.

Java

```
@Component
public class OrderService {

    @RegisterReflectionForBinding(Account.class)
    public void process(Order order) {
        // ...
    }

}
```

Testing Runtime Hints

Spring Core also ships `RuntimeHintsPredicates`, a utility for checking that existing hints match a particular use case. This can be used in your own tests to validate that a `RuntimeHintsRegistrar` contains the expected results. We can write a test for our `SpellCheckService` and ensure that we will be able to load a dictionary at runtime:

```
@Test
void shouldRegisterResourceHints() {
    RuntimeHints hints = new RuntimeHints();
    new SpellCheckServiceRuntimeHints().registerHints(hints,
getClass().getClassLoader());
    assertThat(RuntimeHintsPredicates.resource().forResource("dicts/en.txt"))
            .accepts(hints);
}
```

With `RuntimeHintsPredicates`, we can check for reflection, resource, serialization, or proxy generation hints. This approach works well for unit tests but implies that the runtime behavior of a component is well known.

You can learn more about the global runtime behavior of an application by running its test suite (or the app itself) with the GraalVM tracing agent. This agent will record all relevant calls requiring GraalVM hints at runtime and write them out as JSON configuration files.

For more targeted discovery and testing, Spring Framework ships a dedicated module with core AOT testing utilities, `"org.springframework:spring-core-test"`. This module contains the RuntimeHints Agent, a Java agent that records all method invocations that are related to runtime hints and helps you to assert that a given `RuntimeHints` instance covers all recorded invocations. Let's consider a piece of infrastructure for which we'd like to test the hints we're contributing during the AOT processing phase.

```
/*
 * Copyright 2002-2022 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.docs.core.aot.hints.testing;

import java.lang.reflect.Method;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.springframework.util.ClassUtils;

public class SampleReflection {

    private final Log logger = LogFactory.getLog(SampleReflection.class);

    public void performReflection() {
        try {
            Class<?> springVersion =
ClassUtils.forName("org.springframework.core.SpringVersion", null);
            Method getVersion = ClassUtils.getMethod(springVersion, "getVersion");
            String version = (String) getVersion.invoke(null);
            logger.info("Spring version:" + version);
        }
        catch (Exception exc) {
            logger.error("reflection failed", exc);
        }
    }

}
```

We can then write a unit test (no native compilation required) that checks our contributed hints:

```
/*
 * Copyright 2002-2022 the original author or authors.
 *
```

```
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.docs.core.aot.hints.testing;

import java.util.List;

import org.junit.jupiter.api.Test;

import org.springframework.aot.hint.ExecutableMode;
import org.springframework.aot.hint.RuntimeHints;
import org.springframework.aot.test.agent.EnabledIfRuntimeHintsAgent;
import org.springframework.aot.test.agent.RuntimeHintsInvocations;
import org.springframework.aot.test.agent.RuntimeHintsRecorder;
import org.springframework.core.SpringVersion;

import static org.assertj.core.api.Assertions.assertThat;

// @EnabledIfRuntimeHintsAgent signals that the annotated test class or test
// method is only enabled if the RuntimeHintsAgent is loaded on the current JVM.
// It also tags tests with the "RuntimeHints" JUnit tag.
@EnabledIfRuntimeHintsAgent
class SampleReflectionRuntimeHintsTests {

    @Test
    void shouldRegisterReflectionHints() {
        RuntimeHints runtimeHints = new RuntimeHints();
        // Call a RuntimeHintsRegistrar that contributes hints like:
        runtimeHints.reflection().registerType(SpringVersion.class, typeHint -> {
            typeHint.withMethod("getVersion", List.of(), ExecutableMode.INVOKE);
        });

        // Invoke the relevant piece of code we want to test within a recording lambda
        RuntimeHintsInvocations invocations = RuntimeHintsRecorder.record(() -> {
            SampleReflection sample = new SampleReflection();
            sample.performReflection();
        });
        // assert that the recorded invocations are covered by the contributed hints
        assertThat(invocations).match(runtimeHints);
    }
```

```
}
```

If you forgot to contribute a hint, the test will fail and provide some details about the invocation:

```
org.springframework.docs.core.aot.hints.testing.SampleReflection performReflection
INFO: Spring version:6.0.0-SNAPSHOT

Missing <"ReflectionHints"> for invocation <java.lang.Class#forName>
with arguments ["org.springframework.core.SpringVersion",
    false,
    jdk.internal.loader.ClassLoaders$AppClassLoader@251a69d7].
Stacktrace:
<"org.springframework.util.ClassUtils#forName, Line 284
io.spring.runtimehintstesting.SampleReflection#performReflection, Line 19
io.spring.runtimehintstesting.SampleReflectionRuntimeHintsTests#lambda$shouldRegisterR
eflectionHints$0, Line 25
```

There are various ways to configure this Java agent in your build, so please refer to the documentation of your build tool and test execution plugin. The agent itself can be configured to instrument specific packages (by default, only `org.springframework` is instrumented). You'll find more details in the Spring Framework `buildSrc` README file.

2.11. Appendix

2.11.1. XML Schemas

This part of the appendix lists XML schemas related to the core container.

The `util` Schema

As the name implies, the `util` tags deal with common, utility configuration issues, such as configuring collections, referencing constants, and so forth. To use the tags in the `util` schema, you need to have the following preamble at the top of your Spring XML configuration file (the text in the snippet references the correct schema so that the tags in the `util` namespace are available to you):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
https://www.springframework.org/schema/util/spring-util.xsd">

        <!-- bean definitions here -->

</beans>
```

Using `<util:constant/>`

Consider the following bean definition:

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

The preceding configuration uses a Spring `FactoryBean` implementation (the `FieldRetrievingFactoryBean`) to set the value of the `isolation` property on a bean to the value of the `java.sql.Connection.TRANSACTION_SERIALIZABLE` constant. This is all well and good, but it is verbose and (unnecessarily) exposes Spring's internal plumbing to the end user.

The following XML Schema-based version is more concise, clearly expresses the developer's intent ("inject this constant value"), and it reads better:

```
<bean id="..." class="...">
    <property name="isolation">
        <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
    </property>
</bean>
```

Setting a Bean Property or Constructor Argument from a Field Value

`FieldRetrievingFactoryBean` is a `FactoryBean` that retrieves a `static` or non-static field value. It is typically used for retrieving `public static final` constants, which may then be used to set a property value or constructor argument for another bean.

The following example shows how a `static` field is exposed, by using the `staticField` property:

```
<bean id="myField"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField"
value="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</bean>
```

There is also a convenience usage form where the `static` field is specified as the bean name, as the following example shows:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

This does mean that there is no longer any choice in what the bean `id` is (so any other bean that refers to it also has to use this longer name), but this form is very concise to define and very convenient to use as an inner bean since the `id` does not have to be specified for the bean reference, as the following example shows:

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

You can also access a non-static (instance) field of another bean, as described in the API documentation for the `FieldRetrievingFactoryBean` class.

Injecting enumeration values into beans as either property or constructor arguments is easy to do in Spring. You do not actually have to do anything or know anything about the Spring internals (or even about classes such as the `FieldRetrievingFactoryBean`). The following example enumeration shows how easy injecting an enum value is:

Java

```
package jakarta.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED
}
```

Kotlin

```
package jakarta.persistence

enum class PersistenceContextType {

    TRANSACTION,
    EXTENDED
}
```

Now consider the following setter of type `PersistenceContextType` and the corresponding bean definition:

Java

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }
}
```

Kotlin

```
package example

class Client {

    lateinit var persistenceContextType: PersistenceContextType
}
```

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION"/>
</bean>
```

Using `<util:property-path/>`

Consider the following example:

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age"
class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

The preceding configuration uses a Spring `FactoryBean` implementation (the `PropertyPathFactoryBean`) to create a bean (of type `int`) called `testBean.age` that has a value equal to the `age` property of the `testBean` bean.

Now consider the following example, which adds a `<util:property-path/>` element:

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

The value of the `path` attribute of the `<property-path/>` element follows the form of `beanName.beanProperty`. In this case, it picks up the `age` property of the bean named `testBean`. The value of that `age` property is `10`.

Using `<util:property-path/>` to Set a Bean Property or Constructor Argument

`PropertyPathFactoryBean` is a `FactoryBean` that evaluates a property path on a given target object. The target object can be specified directly or by a bean name. You can then use this value in another bean definition as a property value or constructor argument.

The following example shows a path being used against another bean, by name:

```
<!-- target bean to be referenced by name -->
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- results in 11, which is the value of property 'spouse.age' of bean 'person' -->
<bean id="theAge"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetBeanName" value="person"/>
    <property name="propertyPath" value="spouse.age"/>
</bean>
```

In the following example, a path is evaluated against an inner bean:

```
<!-- results in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetObject">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="12"/>
        </bean>
    </property>
    <property name="propertyPath" value="age"/>
</bean>
```

There is also a shortcut form, where the bean name is the property path. The following example shows the shortcut form:

```
<!-- results in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

This form does mean that there is no choice in the name of the bean. Any reference to it also has to use the same `id`, which is the path. If used as an inner bean, there is no need to refer to it at all, as the following example shows:

```
<bean id="..." class="...">
    <property name="age">
        <bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
    </property>
</bean>
```

You can specifically set the result type in the actual definition. This is not necessary for most use cases, but it can sometimes be useful. See the javadoc for more info on this feature.

Using `<util:properties/>`

Consider the following example:

```
<!-- creates a java.util.Properties instance with values loaded from the supplied
location -->
<bean id="jdbcConfiguration"
class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

The preceding configuration uses a Spring `FactoryBean` implementation (the `PropertiesFactoryBean`) to instantiate a `java.util.Properties` instance with values loaded from the supplied `Resource` location).

The following example uses a `util:properties` element to make a more concise representation:

```
<!-- creates a java.util.Properties instance with values loaded from the supplied
location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-
production.properties"/>
```

Using `<util:list/>`

Consider the following example:

```
<!-- creates a java.util.List instance with values loaded from the supplied
'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
    <property name="sourceList">
        <list>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </list>
    </property>
</bean>
```

The preceding configuration uses a Spring `FactoryBean` implementation (the `ListFactoryBean`) to create a `java.util.List` instance and initialize it with values taken from the supplied `sourceList`.

The following example uses a `<util:list/>` element to make a more concise representation:

```
<!-- creates a java.util.List instance with the supplied values -->
<util:list id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

You can also explicitly control the exact type of `List` that is instantiated and populated by using the `list-class` attribute on the `<util:list/>` element. For example, if we really need a `java.util.LinkedList` to be instantiated, we could use the following configuration:

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

If no `list-class` attribute is supplied, the container chooses a `List` implementation.

Using `<util:map/>`

Consider the following example:

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap'
-->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="sourceMap">
        <map>
            <entry key="pechorin" value="pechorin@hero.org"/>
            <entry key="raskolnikov" value="raskolnikov@slums.org"/>
            <entry key="stavrogin" value="stavrogin@gov.org"/>
            <entry key="porfiry" value="porfiry@gov.org"/>
        </map>
    </property>
</bean>
```

The preceding configuration uses a Spring `FactoryBean` implementation (the `MapFactoryBean`) to create a `java.util.Map` instance initialized with key-value pairs taken from the supplied `'sourceMap'`.

The following example uses a `<util:map/>` element to make a more concise representation:

```
<!-- creates a java.util.Map instance with the supplied key-value pairs -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

You can also explicitly control the exact type of `Map` that is instantiated and populated by using the `'map-class'` attribute on the `<util:map/>` element. For example, if we really need a `java.util.TreeMap` to be instantiated, we could use the following configuration:

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

If no `'map-class'` attribute is supplied, the container chooses a `Map` implementation.

Using `<util:set/>`

Consider the following example:

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet'
-->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="sourceSet">
        <set>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </set>
    </property>
</bean>
```

The preceding configuration uses a Spring `FactoryBean` implementation (the `SetFactoryBean`) to create a `java.util.Set` instance initialized with values taken from the supplied `sourceSet`.

The following example uses a `<util:set/>` element to make a more concise representation:

```
<!-- creates a java.util.Set instance with the supplied values -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

You can also explicitly control the exact type of `Set` that is instantiated and populated by using the `set-class` attribute on the `<util:set/>` element. For example, if we really need a `java.util.TreeSet` to be instantiated, we could use the following configuration:

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

If no `set-class` attribute is supplied, the container chooses a `Set` implementation.

The `aop` Schema

The `aop` tags deal with configuring all things AOP in Spring, including Spring's own proxy-based AOP framework and Spring's integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled Aspect Oriented Programming with Spring.

In the interest of completeness, to use the tags in the `aop` schema, you need to have the following preamble at the top of your Spring XML configuration file (the text in the snippet references the

correct schema so that the tags in the aop namespace are available to you):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- bean definitions here -->

</beans>
```

The `context` Schema

The `context` tags deal with `ApplicationContext` configuration that relates to plumbing — that is, not usually beans that are important to an end-user but rather beans that do a lot of the "grunt" work in Spring, such as `BeanfactoryPostProcessors`. The following snippet references the correct schema so that the elements in the `context` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <!-- bean definitions here -->

</beans>
```

Using `<property-placeholder/>`

This element activates the replacement of `${…}` placeholders, which are resolved against a specified properties file (as a Spring resource location). This element is a convenience mechanism that sets up a `PropertySourcesPlaceholderConfigurer` for you. If you need more control over the specific `PropertySourcesPlaceholderConfigurer` setup, you can explicitly define it as a bean yourself.

Using `<annotation-config/>`

This element activates the Spring infrastructure to detect annotations in bean classes:

- Spring's `@Configuration` model

- `@Autowired`/`@Inject`, `@Value`, and `@Lookup`
- JSR-250's `@Resource`, `@PostConstruct`, and `@PreDestroy` (if available)
- JAX-WS's `@WebServiceRef` and EJB 3's `@EJB` (if available)
- JPA's `@PersistenceContext` and `@PersistenceUnit` (if available)
- Spring's `@EventListener`

Alternatively, you can choose to explicitly activate the individual `BeanPostProcessors` for those annotations.

This element does not activate processing of Spring's `@Transactional` annotation; you can use the `<tx:annotation-driven/>` element for that purpose. Similarly, Spring's caching annotations need to be explicitly enabled as well.

Using `<component-scan/>`

This element is detailed in the section on annotation-based container configuration.

Using `<load-time-weaver/>`

This element is detailed in the section on load-time weaving with AspectJ in the Spring Framework.

Using `<spring-configured/>`

This element is detailed in the section on using AspectJ to dependency inject domain objects with Spring.

Using `<mbean-export/>`

This element is detailed in the section on configuring annotation-based MBean export.

The Beans Schema

Last but not least, we have the elements in the `beans` schema. These elements have been in Spring since the very dawn of the framework. Examples of the various elements in the `beans` schema are not shown here because they are quite comprehensively covered in dependencies and configuration in detail (and, indeed, in that entire chapter).

Note that you can add zero or more key-value pairs to `<bean/>` XML definitions. What, if anything, is done with this extra metadata is totally up to your own custom logic (and so is typically only of use if you write your own custom elements as described in the appendix entitled XML Schema Authoring).

The following example shows the `<meta/>` element in the context of a surrounding `<bean/>` (note that, without any logic to interpret it, the metadata is effectively useless as it stands).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="foo" class="x.y.Foo">
        <meta key="cacheName" value="foo"/> ①
        <property name="name" value="Rick"/>
    </bean>

</beans>
```

① This is the example `meta` element

In the case of the preceding example, you could assume that there is some logic that consumes the bean definition and sets up some caching infrastructure that uses the supplied metadata.

2.11.2. XML Schema Authoring

Since version 2.0, Spring has featured a mechanism for adding schema-based extensions to the basic Spring XML format for defining and configuring beans. This section covers how to write your own custom XML bean definition parsers and integrate such parsers into the Spring IoC container.

To facilitate authoring configuration files that use a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, you should first read the previous section on XML Schemas.

To create new XML configuration extensions:

1. Author an XML schema to describe your custom element(s).
2. Code a custom `NamespaceHandler` implementation.
3. Code one or more `BeanDefinitionParser` implementations (this is where the real work is done).
4. Register your new artifacts with Spring.

For a unified example, we create an XML extension (a custom XML element) that lets us configure objects of the type `SimpleDateFormat` (from the `java.text` package). When we are done, we will be able to define bean definitions of type `SimpleDateFormat` as follows:

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

(We include much more detailed examples follow later in this appendix. The intent of this first simple example is to walk you through the basic steps of making a custom extension.)

Authoring the Schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. For our example, we use the following schema to configure `SimpleDateFormat` objects:

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.example/schema/myns"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:beans="http://www.springframework.org/schema/beans"
        targetNamespace="http://www.mycompany.example/schema/myns"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.springframework.org/schema/beans"/>

    <xsd:element name="dateformat">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="beans:identifiedType"> ①
                    <xsd:attribute name="lenient" type="xsd:boolean"/>
                    <xsd:attribute name="pattern" type="xsd:string" use="required"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

① The indicated line contains an extension base for all identifiable tags (meaning they have an `id` attribute that we can use as the bean identifier in the container). We can use this attribute because we imported the Spring-provided `beans` namespace.

The preceding schema lets us configure `SimpleDateFormat` objects directly in an XML application context file by using the `<myns:dateformat/>` element, as the following example shows:

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

Note that, after we have created the infrastructure classes, the preceding snippet of XML is essentially the same as the following XML snippet:

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-MM-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

The second of the two preceding snippets creates a bean in the container (identified by the name `dateFormat` of type `SimpleDateFormat`) with a couple of properties set.

The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. By using a properly authored schema, you can use autocompletion to let a user choose between several configuration options defined in the enumeration.

Coding a `NamespaceHandler`

In addition to the schema, we need a `NamespaceHandler` to parse all elements of this specific namespace that Spring encounters while parsing configuration files. For this example, the `NamespaceHandler` should take care of the parsing of the `myns:dateformat` element.

The `NamespaceHandler` interface features three methods:

- `init()`: Allows for initialization of the `NamespaceHandler` and is called by Spring before the handler is used.
- `BeanDefinition parse(Element, ParserContext)`: Called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can itself register bean definitions, return a bean definition, or both.
- `BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext)`: Called when Spring encounters an attribute or nested element of a different namespace. The decoration of one or more bean definitions is used (for example) with the scopes that Spring supports. We start by highlighting a simple example, without using decoration, after which we show decoration in a somewhat more advanced example.

Although you can code your own `NamespaceHandler` for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where a single `<myns:dateformat/>` element results in a single `SimpleDateFormat` bean definition). Spring features a number of convenience classes that support this scenario. In the following example, we use the `NamespaceHandlerSupport` class:

Java

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat", new
SimpleDateFormatBeanDefinitionParser());
    }
}
```

Kotlin

```
package org.springframework.samples.xml

import org.springframework.beans.factory.xml.NamespaceHandlerSupport

class MyNamespaceHandler : NamespaceHandlerSupport {

    override fun init() {
        registerBeanDefinitionParser("dateformat",
SimpleDateFormatBeanDefinitionParser())
    }
}
```

You may notice that there is not actually a whole lot of parsing logic in this class. Indeed, the `NamespaceHandlerSupport` class has a built-in notion of delegation. It supports the registration of any number of `BeanDefinitionParser` instances, to which it delegates to when it needs to parse an element in its namespace. This clean separation of concerns lets a `NamespaceHandler` handle the orchestration of the parsing of all of the custom elements in its namespace while delegating to `BeanDefinitionParsers` to do the grunt work of the XML parsing. This means that each `BeanDefinitionParser` contains only the logic for parsing a single custom element, as we can see in the next step.

Using `BeanDefinitionParser`

A `BeanDefinitionParser` is used if the `NamespaceHandler` encounters an XML element of the type that has been mapped to the specific bean definition parser (`dateformat` in this case). In other words, the `BeanDefinitionParser` is responsible for parsing one distinct top-level XML element defined in the schema. In the parser, we' have access to the XML element (and thus to its subelements, too) so that we can parse our custom XML content, as you can see in the following example:

Java

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends
AbstractSingleBeanDefinitionParser { ①

    protected Class getBeanClass(Element element) {
        return SimpleDateFormat.class; ②
    }

    protected void doParse(Element element, BeanDefinitionBuilder bean) {
        // this will never be null since the schema explicitly requires that a value
be supplied
        String pattern = element.getAttribute("pattern");
        bean.addConstructorArgValue(pattern);

        // this however is an optional property
        String lenient = element.getAttribute("lenient");
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
        }
    }

}
```

① We use the Spring-provided `AbstractSingleBeanDefinitionParser` to handle a lot of the basic grunt work of creating a single `BeanDefinition`.

② We supply the `AbstractSingleBeanDefinitionParser` superclass with the type that our single `BeanDefinition` represents.

Kotlin

```
package org.springframework.samples.xml

import org.springframework.beans.factory.support.BeanDefinitionBuilder
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser
import org.springframework.util.StringUtils
import org.w3c.dom.Element

import java.text.SimpleDateFormat

class SimpleDateFormatBeanDefinitionParser : AbstractSingleBeanDefinitionParser() { ①

    override fun getBeanClass(element: Element): Class<*>? { ②
        return SimpleDateFormat::class.java
    }

    override fun doParse(element: Element, bean: BeanDefinitionBuilder) {
        // this will never be null since the schema explicitly requires that a value
be supplied
        val pattern = element.getAttribute("pattern")
        bean.addConstructorArgValue(pattern)

        // this however is an optional property
        val lenient = element.getAttribute("lenient")
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", java.lang.Boolean.valueOf(lenient))
        }
    }
}
```

① We use the Spring-provided `AbstractSingleBeanDefinitionParser` to handle a lot of the basic grunt work of creating a single `BeanDefinition`.

② We supply the `AbstractSingleBeanDefinitionParser` superclass with the type that our single `BeanDefinition` represents.

In this simple case, this is all that we need to do. The creation of our single `BeanDefinition` is handled by the `AbstractSingleBeanDefinitionParser` superclass, as is the extraction and setting of the bean definition's unique identifier.

Registering the Handler and the Schema

The coding is finished. All that remains to be done is to make the Spring XML parsing infrastructure aware of our custom element. We do so by registering our custom `namespaceHandler` and custom XSD file in two special-purpose properties files. These properties files are both placed in a `META-INF` directory in your application and can, for example, be distributed alongside your binary classes in a JAR file. The Spring XML parsing infrastructure automatically picks up your new extension by consuming these special properties files, the formats of which are detailed in the next two sections.

Writing `META-INF/spring.handlers`

The properties file called `spring.handlers` contains a mapping of XML Schema URIs to namespace handler classes. For our example, we need to write the following:

```
http\://www.mycompany.example/schema/myns=org.springframework.samples.xml.MyNamespaceH
andler
```

(The `:` character is a valid delimiter in the Java properties format, so `:` character in the URI needs to be escaped with a backslash.)

The first part (the key) of the key-value pair is the URI associated with your custom namespace extension and needs to exactly match exactly the value of the `targetNamespace` attribute, as specified in your custom XSD schema.

Writing 'META-INF/spring.schemas'

The properties file called `spring.schemas` contains a mapping of XML Schema locations (referred to, along with the schema declaration, in XML files that use the schema as part of the `xsi:schemaLocation` attribute) to classpath resources. This file is needed to prevent Spring from absolutely having to use a default `EntityResolver` that requires Internet access to retrieve the schema file. If you specify the mapping in this properties file, Spring searches for the schema (in this case, `myns.xsd` in the `org.springframework.samples.xml` package) on the classpath. The following snippet shows the line we need to add for our custom schema:

```
http\://www.mycompany.example/schema/myns/myns.xsd=org/springframework/samples/xml/myn
s.xsd
```

(Remember that the `:` character must be escaped.)

You are encouraged to deploy your XSD file (or files) right alongside the `NamespaceHandler` and `BeanDefinitionParser` classes on the classpath.

Using a Custom Extension in Your Spring XML Configuration

Using a custom extension that you yourself have implemented is no different from using one of the “custom” extensions that Spring provides. The following example uses the custom `<dateformat/>` element developed in the previous steps in a Spring XML configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:myns="http://www.mycompany.example/schema/myns"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.mycompany.example/schema/myns
http://www.mycompany.com/schema/myns/myns.xsd">

    <!-- as a top-level bean -->
    <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm"
lenient="true"/> ①

    <bean id="jobDetailTemplate" abstract="true">
        <property name="dateFormat">
            <!-- as an inner bean -->
            <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
        </property>
    </bean>

</beans>
```

① Our custom bean.

More Detailed Examples

This section presents some more detailed examples of custom XML extensions.

Nesting Custom Elements within Custom Elements

The example presented in this section shows how you to write the various artifacts required to satisfy a target of the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:foo="http://www.foo.example/schema/component"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.foo.example/schema/component
http://www.foo.example/schema/component/component.xsd">

    <foo:component id="bionic-family" name="Bionic-1">
        <foo:component name="Mother-1">
            <foo:component name="Karate-1"/>
            <foo:component name="Sport-1"/>
        </foo:component>
        <foo:component name="Rock-1"/>
    </foo:component>

</beans>
```

The preceding configuration nests custom extensions within each other. The class that is actually configured by the `<foo:component/>` element is the `Component` class (shown in the next example). Notice how the `Component` class does not expose a setter method for the `components` property. This makes it hard (or rather impossible) to configure a bean definition for the `Component` class by using setter injection. The following listing shows the `Component` class:

Java

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

    private String name;
    private List<Component> components = new ArrayList<Component> ();

    // mmm, there is no setter method for the 'components'
    public void addComponent(Component component) {
        this.components.add(component);
    }

    public List<Component> getComponents() {
        return components;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Kotlin

```
package com.foo

import java.util.ArrayList

class Component {

    var name: String? = null
    private val components = ArrayList<Component>()

    // mmm, there is no setter method for the 'components'
    fun addComponent(component: Component) {
        this.components.add(component)
    }

    fun getComponents(): List<Component> {
        return components
    }
}
```

The typical solution to this issue is to create a custom `FactoryBean` that exposes a setter property for the `components` property. The following listing shows such a custom `FactoryBean`:

Java

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

    private Component parent;
    private List<Component> children;

    public void setParent(Component parent) {
        this.parent = parent;
    }

    public void setChildren(List<Component> children) {
        this.children = children;
    }

    public Component getObject() throws Exception {
        if (this.children != null && this.children.size() > 0) {
            for (Component child : children) {
                this.parent.addComponent(child);
            }
        }
        return this.parent;
    }

    public Class<Component> getObjectType() {
        return Component.class;
    }

    public boolean isSingleton() {
        return true;
    }
}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.FactoryBean
import org.springframework.stereotype.Component

class ComponentFactoryBean : FactoryBean<Component> {

    private var parent: Component? = null
    private var children: List<Component>? = null

    fun setParent(parent: Component) {
        this.parent = parent
    }

    fun setChildren(children: List<Component>) {
        this.children = children
    }

    override fun getObject(): Component? {
        if (this.children != null && this.children!!.isNotEmpty()) {
            for (child in children!!) {
                this.parent!!.addComponent(child)
            }
        }
        return this.parent
    }

    override fun getObjectType(): Class<Component>? {
        return Component::class.java
    }

    override fun isSingleton(): Boolean {
        return true
    }
}
```

This works nicely, but it exposes a lot of Spring plumbing to the end user. What we are going to do is write a custom extension that hides away all of this Spring plumbing. If we stick to the steps described previously, we start off by creating the XSD schema to define the structure of our custom tag, as the following listing shows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.example/schema/component"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.example/schema/component"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified">

    <xsd:element name="component">
        <xsd:complexType>
            <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="component"/>
            </xsd:choice>
            <xsd:attribute name="id" type="xsd:ID"/>
            <xsd:attribute name="name" use="required" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Again following the process described earlier, we then create a custom `NamespaceHandler`:

Java

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("component", new
ComponentBeanDefinitionParser());
    }
}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.xml.NamespaceHandlerSupport

class ComponentNamespaceHandler : NamespaceHandlerSupport() {

    override fun init() {
        registerBeanDefinitionParser("component", ComponentBeanDefinitionParser())
    }
}
```

Next up is the custom `BeanDefinitionParser`. Remember that we are creating a `BeanDefinition` that describes a `ComponentFactoryBean`. The following listing shows our custom `BeanDefinitionParser` implementation:

Java

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

    protected AbstractBeanDefinition parseInternal(Element element, ParserContext
parserContext) {
        return parseComponentElement(element);
    }

    private static AbstractBeanDefinition parseComponentElement(Element element) {
        BeanDefinitionBuilder factory =
BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);
        factory.addPropertyValue("parent", parseComponent(element));

        List<Element> childElements = DomUtils.getChildElementsByTagName(element,
"component");
        if (childElements != null && childElements.size() > 0) {
            parseChildComponents(childElements, factory);
        }

        return factory.getBeanDefinition();
    }

    private static BeanDefinition parseComponent(Element element) {
        BeanDefinitionBuilder component =
BeanDefinitionBuilder.rootBeanDefinition(Component.class);
        component.addPropertyValue("name", element.getAttribute("name"));
        return component.getBeanDefinition();
    }

    private static void parseChildComponents(List<Element> childElements,
BeanDefinitionBuilder factory) {
        ManagedList<BeanDefinition> children = new
ManagedList<BeanDefinition>(childElements.size());
```

```
        for (Element element : childElements) {
            children.add(parseComponentElement(element));
        }
        factory.addPropertyValue("children", children);
    }
}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.config.BeanDefinition
import org.springframework.beans.factory.support.AbstractBeanDefinition
import org.springframework.beans.factory.support.BeanDefinitionBuilder
import org.springframework.beans.factory.support.ManagedList
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser
import org.springframework.beans.factory.xml.ParserContext
import org.springframework.util.xml.DomUtils
import org.w3c.dom.Element

import java.util.List

class ComponentBeanDefinitionParser : AbstractBeanDefinitionParser() {

    override fun parseInternal(element: Element, parserContext: ParserContext):
AbstractBeanDefinition? {
        return parseComponentElement(element)
    }

    private fun parseComponentElement(element: Element): AbstractBeanDefinition {
        val factory =
BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean::class.java)
        factory.addPropertyValue("parent", parseComponent(element))

        val childElements = DomUtils.getChildElementsByTagName(element, "component")
        if (childElements != null && childElements.size > 0) {
            parseChildComponents(childElements, factory)
        }

        return factory.getBeanDefinition()
    }

    private fun parseComponent(element: Element): BeanDefinition {
        val component =
BeanDefinitionBuilder.rootBeanDefinition(Component::class.java)
        component.addPropertyValue("name", element.getAttribute("name"))
        return component.beanDefinition
    }

    private fun parseChildComponents(childElements: List<Element>, factory:
BeanDefinitionBuilder) {
        val children = ManagedList<BeanDefinition>(childElements.size)
        for (element in childElements) {
            children.add(parseComponentElement(element))
        }
        factory.addPropertyValue("children", children)
    }
}
```

Finally, the various artifacts need to be registered with the Spring XML infrastructure, by modifying the `META-INF/spring.handlers` and `META-INF/spring.schemas` files, as follows:

```
# in 'META-INF/spring.handlers'
http\://www.foo.example/schema/component=com.foo.ComponentNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.example/schema/component/component.xsd=com/foo/component.xsd
```

Custom Attributes on "Normal" Elements

Writing your own custom parser and the associated artifacts is not hard. However, it is sometimes not the right thing to do. Consider a scenario where you need to add metadata to already existing bean definitions. In this case, you certainly do not want to have to write your own entire custom extension. Rather, you merely want to add an additional attribute to the existing bean definition element.

By way of another example, suppose that you define a bean definition for a service object that (unknown to it) accesses a clustered JCache, and you want to ensure that the named JCache instance is eagerly started within the surrounding cluster. The following listing shows such a definition:

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"
        jcache:cache-name="checking.account">
    <!-- other dependencies here... -->
</bean>
```

We can then create another `BeanDefinition` when the `'jcache:cache-name'` attribute is parsed. This `BeanDefinition` then initializes the named JCache for us. We can also modify the existing `BeanDefinition` for the `'checkingAccountService'` so that it has a dependency on this new JCache-initializing `BeanDefinition`. The following listing shows our `JCacheInitializer`:

Java

```
package com.foo;

public class JCacheInitializer {

    private String name;

    public JCacheInitializer(String name) {
        this.name = name;
    }

    public void initialize() {
        // lots of JCache API calls to initialize the named cache...
    }
}
```

Kotlin

```
package com.foo

class JCacheInitializer(private val name: String) {

    fun initialize() {
        // lots of JCache API calls to initialize the named cache...
    }
}
```

Now we can move onto the custom extension. First, we need to author the XSD schema that describes the custom attribute, as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.example/schema/jcache"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.foo.example/schema/jcache"
        elementFormDefault="qualified">

    <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

Next, we need to create the associated `NamespaceHandler`, as follows:

Java

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            new JCacheInitializingBeanDefinitionDecorator());
    }

}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.xml.NamespaceHandlerSupport

class JCacheNamespaceHandler : NamespaceHandlerSupport() {

    override fun init() {
        super.registerBeanDefinitionDecoratorForAttribute("cache-name",
                JCacheInitializingBeanDefinitionDecorator())
    }

}
```

Next, we need to create the parser. Note that, in this case, because we are going to parse an XML attribute, we write a `BeanDefinitionDecorator` rather than a `BeanDefinitionParser`. The following listing shows our `BeanDefinitionDecorator` implementation:

Java

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
public class JCacheInitializingBeanDefinitionDecorator implements
BeanDefinitionDecorator {

    private static final String[] EMPTY_STRING_ARRAY = new String[0];

    public BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,
            ParserContext ctx) {
        String initializerBeanName = registerJCacheInitializer(source, ctx);
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,
            String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition)
holder.getBeanDefinition());
        String[] dependsOn = definition.getDependsOn();
        if (dependsOn == null) {
            dependsOn = new String[]{initializerBeanName};
        } else {
            List dependencies = new ArrayList(Arrays.asList(dependsOn));
            dependencies.add(initializerBeanName);
            dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
        }
        definition.setDependsOn(dependsOn);
    }

    private String registerJCacheInitializer(Node source, ParserContext ctx) {
        String cacheName = ((Attr) source).getValue();
        String beanName = cacheName + "-initializer";
        if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
            BeanDefinitionBuilder initializer =
BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class);
            initializer.addConstructorArg(cacheName);
            ctx.getRegistry().registerBeanDefinition(beanName,
initializer.getBeanDefinition());
        }
        return beanName;
    }
}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.config.BeanDefinitionHolder
import org.springframework.beans.factory.support.AbstractBeanDefinition
import org.springframework.beans.factory.support.BeanDefinitionBuilder
import org.springframework.beans.factory.xml.BeanDefinitionDecorator
import org.springframework.beans.factory.xml.ParserContext
import org.w3c.dom.Attr
import org.w3c.dom.Node

import java.util.ArrayList

class JCacheInitializingBeanDefinitionDecorator : BeanDefinitionDecorator {

    override fun decorate(source: Node, holder: BeanDefinitionHolder,
                          ctx: ParserContext): BeanDefinitionHolder {
        val initializerBeanName = registerJCacheInitializer(source, ctx)
        createDependencyOnJCacheInitializer(holder, initializerBeanName)
        return holder
    }

    private fun createDependencyOnJCacheInitializer(holder: BeanDefinitionHolder,
                                                    initializerBeanName: String) {
        val definition = holder.beanDefinition as AbstractBeanDefinition
        var dependsOn = definition.dependsOn
        dependsOn = if (dependsOn == null) {
            arrayOf(initializerBeanName)
        } else {
            val dependencies = ArrayList(listOf(*dependsOn))
            dependencies.add(initializerBeanName)
            dependencies.toTypedArray()
        }
        definition.setDependsOn(*dependsOn)
    }

    private fun registerJCacheInitializer(source: Node, ctx: ParserContext): String {
        val cacheName = (source as Attr).value
        val beanName = "$cacheName-initializer"
        if (!ctx.registry.containsBeanDefinition(beanName)) {
            val initializer =
BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer::class.java)
            initializer.addConstructorArg(cacheName)
            ctx.registry.registerBeanDefinition(beanName,
initializer.getBeanDefinition())
        }
        return beanName
    }
}
```

Finally, we need to register the various artifacts with the Spring XML infrastructure by modifying the `META-INF/spring.handlers` and `META-INF/spring.schemas` files, as follows:

```
# in 'META-INF/spring.handlers'
http\://www.foo.example/schema/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.example/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

2.11.3. Application Startup Steps

This part of the appendix lists the existing `StartupSteps` that the core container is instrumented with.

The name and detailed information about each startup step is not part of the public contract and is subject to change; this is considered as an implementation detail of the core container and will follow its behavior changes.

Table 15. Application startup steps defined in the core container

Name	Description	Tags
`spring.beans.instantiate`	Instantiation of a bean and its dependencies.	`beanName` the name of the bean, `beanType` the type required at the injection point.
`spring.beans.smart-initialize`	Initialization of `SmartInitializingSingleton` beans.	`beanName` the name of the bean.
`spring.context.annotated-bean-reader.create`	Creation of the `AnnotatedBeanDefinitionReader`.	
`spring.context.base-packages.scan`	Scanning of base packages.	`packages` array of base packages for scanning.
`spring.context.beans.post-process`	Beans post-processing phase.	
`spring.context.bean-factory.post-process`	Invocation of the `BeanFactoryPostProcessor` beans.	`postProcessor` the current post-processor.
`spring.context.beandef-registry.post-process`	Invocation of the `BeanDefinitionRegistryPostProcessor` beans.	`postProcessor` the current post-processor.
`spring.context.component-classes.register`	Registration of component classes through `AnnotationConfigApplicationContext#register`.	`classes` array of given classes for registration.

Name	Description	Tags
`spring.context.config-classes.enhance`	Enhancement of configuration classes with CGLIB proxies.	`classCount` count of enhanced classes.
`spring.context.config-classes.parse`	Configuration classes parsing phase with the `ConfigurationClassPostProcessor`.	`classCount` count of processed classes.
`spring.context.refresh`	Application context refresh phase.	

Chapter 3. Testing

This chapter covers Spring's support for integration testing and best practices for unit testing. The Spring team advocates test-driven development (TDD). The Spring team has found that the correct use of inversion of control (IoC) certainly does make both unit and integration testing easier (in that the presence of setter methods and appropriate constructors on classes makes them easier to wire together in a test without having to set up service locator registries and similar structures).

3.1. Introduction to Spring Testing

Testing is an integral part of enterprise software development. This chapter focuses on the value added by the IoC principle to unit testing and on the benefits of the Spring Framework's support for integration testing. (A thorough treatment of testing in the enterprise is beyond the scope of this reference manual.)

3.2. Unit Testing

Dependency injection should make your code less dependent on the container than it would be with traditional J2EE / Java EE development. The POJOs that make up your application should be testable in JUnit or TestNG tests, with objects instantiated by using the `new` operator, without Spring or any other container. You can use mock objects (in conjunction with other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations for Spring, the resulting clean layering and componentization of your codebase facilitate easier unit testing. For example, you can test service layer objects by stubbing or mocking DAO or repository interfaces, without needing to access persistent data while running unit tests.

True unit tests typically run extremely quickly, as there is no runtime infrastructure to set up. Emphasizing true unit tests as part of your development methodology can boost your productivity. You may not need this section of the testing chapter to help you write effective unit tests for your IoC-based applications. For certain unit testing scenarios, however, the Spring Framework provides mock objects and testing support classes, which are described in this chapter.

3.2.1. Mock Objects

Spring includes a number of packages dedicated to mocking:

- Environment
- JNDI
- Servlet API
- Spring Web Reactive

Environment

The `org.springframework.mock.env` package contains mock implementations of the `Environment` and `PropertySource` abstractions (see Bean Definition Profiles and `PropertySource` Abstraction). `MockEnvironment` and `MockPropertySource` are useful for developing out-of-container tests for code

that depends on environment-specific properties.

JNDI

The `org.springframework.mock.jndi` package contains a partial implementation of the JNDI SPI, which you can use to set up a simple JNDI environment for test suites or stand-alone applications. If, for example, JDBC `DataSource` instances get bound to the same JNDI names in test code as they do in a Jakarta EE container, you can reuse both application code and configuration in testing scenarios without modification.

The mock JNDI support in the `org.springframework.mock.jndi` package is officially deprecated as of Spring Framework 5.2 in favor of complete solutions from third parties such as Simple-JNDI.

Servlet API

The `org.springframework.mock.web` package contains a comprehensive set of Servlet API mock objects that are useful for testing web contexts, controllers, and filters. These mock objects are targeted at usage with Spring's Web MVC framework and are generally more convenient to use than dynamic mock objects (such as EasyMock) or alternative Servlet API mock objects (such as MockObjects).

Since Spring Framework 6.0, the mock objects in `org.springframework.mock.web` are based on the Servlet 6.0 API.

The Spring MVC Test framework builds on the mock Servlet API objects to provide an integration testing framework for Spring MVC. See MockMvc.

Spring Web Reactive

The `org.springframework.mock.http.server.reactive` package contains mock implementations of `ServerHttpRequest` and `ServerHttpResponse` for use in WebFlux applications. The `org.springframework.mock.web.server` package contains a mock `ServerWebExchange` that depends on those mock request and response objects.

Both `MockServerHttpRequest` and `MockServerHttpResponse` extend from the same abstract base classes as server-specific implementations and share behavior with them. For example, a mock request is immutable once created, but you can use the `mutate()` method from `ServerHttpRequest` to create a modified instance.

In order for the mock response to properly implement the write contract and return a write completion handle (that is, `Mono<Void>`), it by default uses a `Flux` with `cache().then()`, which buffers the data and makes it available for assertions in tests. Applications can set a custom write function (for example, to test an infinite stream).

The WebTestClient builds on the mock request and response to provide support for testing WebFlux applications without an HTTP server. The client can also be used for end-to-end tests with a running server.

3.2.2. Unit Testing Support Classes

Spring includes a number of classes that can help with unit testing. They fall into two categories:

- General Testing Utilities
- Spring MVC Testing Utilities

General Testing Utilities

The `org.springframework.test.util` package contains several general purpose utilities for use in unit and integration testing.

`AopTestUtils` is a collection of AOP-related utility methods. You can use these methods to obtain a reference to the underlying target object hidden behind one or more Spring proxies. For example, if you have configured a bean as a dynamic mock by using a library such as EasyMock or Mockito, and the mock is wrapped in a Spring proxy, you may need direct access to the underlying mock to configure expectations on it and perform verifications. For Spring's core AOP utilities, see `AopUtils` and `AopProxyUtils`.

`ReflectionTestUtils` is a collection of reflection-based utility methods. You can use these methods in testing scenarios where you need to change the value of a constant, set a non-`public` field, invoke a non-`public` setter method, or invoke a non-`public` configuration or lifecycle callback method when testing application code for use cases such as the following:

- ORM frameworks (such as JPA and Hibernate) that condone `private` or `protected` field access as opposed to `public` setter methods for properties in a domain entity.
- Spring's support for annotations (such as `@Autowired`, `@Inject`, and `@Resource`), that provide dependency injection for `private` or `protected` fields, setter methods, and configuration methods.
- Use of annotations such as `@PostConstruct` and `@PreDestroy` for lifecycle callback methods.

`TestSocketUtils` is a simple utility for finding available TCP ports on `localhost` for use in integration testing scenarios.

`TestSocketUtils` can be used in integration tests which start an external server on an available random port. However, these utilities make no guarantee about the subsequent availability of a given port and are therefore unreliable. Instead of using `TestSocketUtils` to find an available local port for a server, it is recommended that you rely on a server's ability to start on a random ephemeral port that it selects or is assigned by the operating system. To interact with that server, you should query the server for the port it is currently using.

Spring MVC Testing Utilities

The `org.springframework.test.web` package contains `ModelAndViewAssert`, which you can use in combination with JUnit, TestNG, or any other testing framework for unit tests that deal with Spring MVC `ModelAndView` objects.

Unit testing Spring MVC Controllers

To unit test your Spring MVC `Controller` classes as POJOs, use `ModelAndViewAssert` combined with `MockHttpServletRequest`, `MockHttpSession`, and so on from Spring's Servlet API mocks. For thorough integration testing of your Spring MVC and REST `Controller` classes in conjunction with your `WebApplicationContext` configuration for Spring MVC, use the Spring MVC Test Framework instead.

3.3. Integration Testing

It is important to be able to perform some integration testing without requiring deployment to your application server or connecting to other enterprise infrastructure. Doing so lets you test things such as:

- The correct wiring of your Spring IoC container contexts.
- Data access using JDBC or an ORM tool. This can include such things as the correctness of SQL statements, Hibernate queries, JPA entity mappings, and so forth.

The Spring Framework provides first-class support for integration testing in the `spring-test` module. The name of the actual JAR file might include the release version and might also be in the long `org.springframework.test` form, depending on where you get it from (see the section on Dependency Management for an explanation). This library includes the `org.springframework.test` package, which contains valuable classes for integration testing with a Spring container. This testing does not rely on an application server or other deployment environment. Such tests are slower to run than unit tests but much faster than the equivalent Selenium tests or remote tests that rely on deployment to an application server.

Unit and integration testing support is provided in the form of the annotation-driven Spring TestContext Framework. The TestContext framework is agnostic of the actual testing framework in use, which allows instrumentation of tests in various environments, including JUnit, TestNG, and others.

The following section provides an overview of the high-level goals of Spring's integration support, and the rest of this chapter then focuses on dedicated topics:

- JDBC Testing Support
- Spring TestContext Framework
- WebTestClient
- MockMvc
- Testing Client Applications
- Annotations

3.3.1. Goals of Integration Testing

Spring's integration testing support has the following primary goals:

- To manage Spring IoC container caching between tests.

- To provide Dependency Injection of test fixture instances.
- To provide transaction management appropriate to integration testing.
- To supply Spring-specific base classes that assist developers in writing integration tests.

The next few sections describe each goal and provide links to implementation and configuration details.

Context Management and Caching

The Spring TestContext Framework provides consistent loading of Spring `ApplicationContext` instances and `WebApplicationContext` instances as well as caching of those contexts. Support for the caching of loaded contexts is important, because startup time can become an issue — not because of the overhead of Spring itself, but because the objects instantiated by the Spring container take time to instantiate. For example, a project with 50 to 100 Hibernate mapping files might take 10 to 20 seconds to load the mapping files, and incurring that cost before running every test in every test fixture leads to slower overall test runs that reduce developer productivity.

Test classes typically declare either an array of resource locations for XML or Groovy configuration metadata — often in the classpath — or an array of component classes that is used to configure the application. These locations or classes are the same as or similar to those specified in `web.xml` or other configuration files for production deployments.

By default, once loaded, the configured `ApplicationContext` is reused for each test. Thus, the setup cost is incurred only once per test suite, and subsequent test execution is much faster. In this context, the term “test suite” means all tests run in the same JVM — for example, all tests run from an Ant, Maven, or Gradle build for a given project or module. In the unlikely case that a test corrupts the application context and requires reloading (for example, by modifying a bean definition or the state of an application object) the TestContext framework can be configured to reload the configuration and rebuild the application context before executing the next test.

See Context Management and Context Caching with the TestContext framework.

Dependency Injection of Test Fixtures

When the TestContext framework loads your application context, it can optionally configure instances of your test classes by using Dependency Injection. This provides a convenient mechanism for setting up test fixtures by using preconfigured beans from your application context. A strong benefit here is that you can reuse application contexts across various testing scenarios (for example, for configuring Spring-managed object graphs, transactional proxies, `DataSource` instances, and others), thus avoiding the need to duplicate complex test fixture setup for individual test cases.

As an example, consider a scenario where we have a class (`HibernateTitleRepository`) that implements data access logic for a `Title` domain entity. We want to write integration tests that test the following areas:

- The Spring configuration: Basically, is everything related to the configuration of the `HibernateTitleRepository` bean correct and present?
- The Hibernate mapping file configuration: Is everything mapped correctly and are the correct

lazy-loading settings in place?

- The logic of the `HibernateTitleRepository`: Does the configured instance of this class perform as anticipated?

See dependency injection of test fixtures with the TestContext framework.

Transaction Management

One common issue in tests that access a real database is their effect on the state of the persistence store. Even when you use a development database, changes to the state may affect future tests. Also, many operations — such as inserting or modifying persistent data — cannot be performed (or verified) outside of a transaction.

The TestContext framework addresses this issue. By default, the framework creates and rolls back a transaction for each test. You can write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they behave correctly, according to their configured transactional semantics. In addition, if a test method deletes the contents of selected tables while running within the transaction managed for the test, the transaction rolls back by default, and the database returns to its state prior to execution of the test. Transactional support is provided to a test by using a `PlatformTransactionManager` bean defined in the test's application context.

If you want a transaction to commit (unusual, but occasionally useful when you want a particular test to populate or modify the database), you can tell the TestContext framework to cause the transaction to commit instead of roll back by using the `@Commit` annotation.

See transaction management with the TestContext framework.

Support Classes for Integration Testing

The Spring TestContext Framework provides several `abstract` support classes that simplify the writing of integration tests. These base test classes provide well-defined hooks into the testing framework as well as convenient instance variables and methods, which let you access:

- The `ApplicationContext`, for performing explicit bean lookups or testing the state of the context as a whole.
- A `JdbcTemplate`, for executing SQL statements to query the database. You can use such queries to confirm database state both before and after execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives.

In addition, you may want to create your own custom, application-wide superclass with instance variables and methods specific to your project.

See support classes for the TestContext framework.

3.4. JDBC Testing Support

3.4.1. JdbcTestUtils

The `org.springframework.test.jdbc` package contains `JdbcTestUtils`, which is a collection of JDBC-related utility functions intended to simplify standard database testing scenarios. Specifically, `JdbcTestUtils` provides the following static utility methods.

- `countRowsInTable(..)`: Counts the number of rows in the given table.
- `countRowsInTableWhere(..)`: Counts the number of rows in the given table by using the provided `WHERE` clause.
- `deleteFromTables(..)`: Deletes all rows from the specified tables.
- `deleteFromTableWhere(..)`: Deletes rows from the given table by using the provided `WHERE` clause.
- `dropTables(..)`: Drops the specified tables.

`AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests` provide convenience methods that delegate to the aforementioned methods in `JdbcTestUtils`.

3.4.2. Embedded Databases

The `spring-jdbc` module provides support for configuring and launching an embedded database, which you can use in integration tests that interact with a database. For details, see Embedded Database Support and Testing Data Access Logic with an Embedded Database.

3.5. Spring TestContext Framework

The Spring TestContext Framework (located in the `org.springframework.test.context` package) provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use. The TestContext framework also places a great deal of importance on convention over configuration, with reasonable defaults that you can override through annotation-based configuration.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit 4, JUnit Jupiter (AKA JUnit 5), and TestNG. For JUnit 4 and TestNG, Spring provides `abstract` support classes. Furthermore, Spring provides a custom JUnit `Runner` and custom JUnit `Rules` for JUnit 4 and a custom `Extension` for JUnit Jupiter that let you write so-called POJO test classes. POJO test classes are not required to extend a particular class hierarchy, such as the `abstract` support classes.

The following section provides an overview of the internals of the TestContext framework. If you are interested only in using the framework and are not interested in extending it with your own custom listeners or custom loaders, feel free to go directly to the configuration (context management, dependency injection, transaction management), support classes, and annotation support sections.

3.5.1. Key Abstractions

The core of the framework consists of the `TestContextManager` class and the `TestContext`, `TestExecutionListener`, and `SmartContextLoader` interfaces. A `TestContextManager` is created for each test class (for example, for the execution of all test methods within a single test class in JUnit Jupiter). The `TestContextManager`, in turn, manages a `TestContext` that holds the context of the current test. The `TestContextManager` also updates the state of the `TestContext` as the test progresses and delegates to `TestExecutionListener` implementations, which instrument the actual test execution by providing dependency injection, managing transactions, and so on. A `SmartContextLoader` is responsible for loading an `ApplicationContext` for a given test class. See the javadoc and the Spring test suite for further information and examples of various implementations.

`TestContext`

`TestContext` encapsulates the context in which a test is run (agnostic of the actual testing framework in use) and provides context management and caching support for the test instance for which it is responsible. The `TestContext` also delegates to a `SmartContextLoader` to load an `ApplicationContext` if requested.

`TestContextManager`

`TestContextManager` is the main entry point into the Spring TestContext Framework and is responsible for managing a single `TestContext` and signaling events to each registered `TestExecutionListener` at well-defined test execution points:

- Prior to any “before class” or “before all” methods of a particular testing framework.
- Test instance post-processing.
- Prior to any “before” or “before each” methods of a particular testing framework.
- Immediately before execution of the test method but after test setup.
- Immediately after execution of the test method but before test tear down.
- After any “after” or “after each” methods of a particular testing framework.
- After any “after class” or “after all” methods of a particular testing framework.

`TestExecutionListener`

`TestExecutionListener` defines the API for reacting to test-execution events published by the `TestContextManager` with which the listener is registered. See `TestExecutionListener` Configuration.

Context Loaders

`ContextLoader` is a strategy interface for loading an `ApplicationContext` for an integration test managed by the Spring TestContext Framework. You should implement `SmartContextLoader` instead of this interface to provide support for component classes, active bean definition profiles, test property sources, context hierarchies, and `WebApplicationContext` support.

`SmartContextLoader` is an extension of the `ContextLoader` interface that supersedes the original minimal `ContextLoader` SPI. Specifically, a `SmartContextLoader` can choose to process resource locations, component classes, or context initializers. Furthermore, a `SmartContextLoader` can set

active bean definition profiles and test property sources in the context that it loads.

Spring provides the following implementations:

- `DelegatingSmartContextLoader`: One of two default loaders, it delegates internally to an `AnnotationConfigContextLoader`, a `GenericXmlContextLoader`, or a `GenericGroovyXmlContextLoader`, depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. Groovy support is enabled only if Groovy is on the classpath.
- `WebDelegatingSmartContextLoader`: One of two default loaders, it delegates internally to an `AnnotationConfigWebContextLoader`, a `GenericXmlWebContextLoader`, or a `GenericGroovyXmlWebContextLoader`, depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. A web `ContextLoader` is used only if `@WebAppConfiguration` is present on the test class. Groovy support is enabled only if Groovy is on the classpath.
- `AnnotationConfigContextLoader`: Loads a standard `ApplicationContext` from component classes.
- `AnnotationConfigWebContextLoader`: Loads a `WebApplicationContext` from component classes.
- `GenericGroovyXmlContextLoader`: Loads a standard `ApplicationContext` from resource locations that are either Groovy scripts or XML configuration files.
- `GenericGroovyXmlWebContextLoader`: Loads a `WebApplicationContext` from resource locations that are either Groovy scripts or XML configuration files.
- `GenericXmlContextLoader`: Loads a standard `ApplicationContext` from XML resource locations.
- `GenericXmlWebContextLoader`: Loads a `WebApplicationContext` from XML resource locations.

3.5.2. Bootstrapping the TestContext Framework

The default configuration for the internals of the Spring TestContext Framework is sufficient for all common use cases. However, there are times when a development team or third party framework would like to change the default `ContextLoader`, implement a custom `TestContext` or `ContextCache`, augment the default sets of `ContextCustomizerFactory` and `TestExecutionListener` implementations, and so on. For such low-level control over how the TestContext framework operates, Spring provides a bootstrapping strategy.

`TestContextBootstrapper` defines the SPI for bootstrapping the TestContext framework. A `TestContextBootstrapper` is used by the `TestContextManager` to load the `TestExecutionListener` implementations for the current test and to build the `TestContext` that it manages. You can configure a custom bootstrapping strategy for a test class (or test class hierarchy) by using `@BootstrapWith`, either directly or as a meta-annotation. If a bootstrapper is not explicitly configured by using `@BootstrapWith`, either the `DefaultTestContextBootstrapper` or the `WebTestContextBootstrapper` is used, depending on the presence of `@WebAppConfiguration`.

Since the `TestContextBootstrapper` SPI is likely to change in the future (to accommodate new requirements), we strongly encourage implementers not to implement this interface directly but rather to extend `AbstractTestContextBootstrapper` or one of its concrete subclasses instead.

3.5.3. `TestExecutionListener` Configuration

Spring provides the following `TestExecutionListener` implementations that are registered by default, exactly in the following order:

- `ServletTestExecutionListener`: Configures Servlet API mocks for a `WebApplicationContext`.
- `DirtiesContextBeforeModesTestExecutionListener`: Handles the `@DirtiesContext` annotation for "before" modes.
- `ApplicationEventsTestExecutionListener`: Provides support for `ApplicationEvents`.
- `DependencyInjectionTestExecutionListener`: Provides dependency injection for the test instance.
- `DirtiesContextTestExecutionListener`: Handles the `@DirtiesContext` annotation for "after" modes.
- `TransactionalTestExecutionListener`: Provides transactional test execution with default rollback semantics.
- `SqlScriptsTestExecutionListener`: Runs SQL scripts configured by using the `@Sql` annotation.
- `EventPublishingTestExecutionListener`: Publishes test execution events to the test's `ApplicationContext` (see Test Execution Events).

Registering `TestExecutionListener` Implementations

You can register `TestExecutionListener` implementations explicitly for a test class, its subclasses, and its nested classes by using the `@TestExecutionListeners` annotation. See annotation support and the javadoc for `@TestExecutionListeners` for details and examples.

Switching to default `TestExecutionListener` implementations

If you extend a class that is annotated with `@TestExecutionListeners` and you need to switch to using the default set of listeners, you can annotate your class with the following.

Java

```
// Switch to default listeners
@TestExecutionListeners(
    listeners = {},
    inheritListeners = false,
    mergeMode = MERGE_WITH_DEFAULTS)
class MyTest extends BaseTest {
    // class body...
}
```

Kotlin

```
// Switch to default listeners
@TestExecutionListeners(
    listeners = [],
    inheritListeners = false,
    mergeMode = MERGE_WITH_DEFAULTS)
class MyTest : BaseTest {
    // class body...
}
```

Automatic Discovery of Default `TestExecutionListener` Implementations

Registering `TestExecutionListener` implementations by using `@TestExecutionListeners` is suitable for custom listeners that are used in limited testing scenarios. However, it can become cumbersome if a custom listener needs to be used across an entire test suite. This issue is addressed through support for automatic discovery of default `TestExecutionListener` implementations through the `SpringFactoriesLoader` mechanism.

Specifically, the `spring-test` module declares all core default `TestExecutionListener` implementations under the `org.springframework.test.context.TestExecutionListener` key in its `META-INF/spring.factories` properties file. Third-party frameworks and developers can contribute their own `TestExecutionListener` implementations to the list of default listeners in the same manner through their own `META-INF/spring.factories` properties file.

Ordering `TestExecutionListener` Implementations

When the TestContext framework discovers default `TestExecutionListener` implementations through the aforementioned `SpringFactoriesLoader` mechanism, the instantiated listeners are sorted by using Spring's `AnnotationAwareOrderComparator`, which honors Spring's `Ordered` interface and `@Order` annotation for ordering. `AbstractTestExecutionListener` and all default `TestExecutionListener` implementations provided by Spring implement `Ordered` with appropriate

values. Third-party frameworks and developers should therefore make sure that their default `TestExecutionListener` implementations are registered in the proper order by implementing `Ordered` or declaring `@Order`. See the javadoc for the `getOrder()` methods of the core default `TestExecutionListener` implementations for details on what values are assigned to each core listener.

Merging `TestExecutionListener` Implementations

If a custom `TestExecutionListener` is registered via `@TestExecutionListeners`, the default listeners are not registered. In most common testing scenarios, this effectively forces the developer to manually declare all default listeners in addition to any custom listeners. The following listing demonstrates this style of configuration:

Java

```
@ContextConfiguration
@TestExecutionListeners({
    MyCustomTestExecutionListener.class,
    ServletTestExecutionListener.class,
    DirtiesContextBeforeModesTestExecutionListener.class,
    DependencyInjectionTestExecutionListener.class,
    DirtiesContextTestExecutionListener.class,
    TransactionalTestExecutionListener.class,
    SqlScriptsTestExecutionListener.class
})
class MyTest {
    // class body...
}
```

Kotlin

```
@ContextConfiguration
@TestExecutionListeners(
    MyCustomTestExecutionListener::class,
    ServletTestExecutionListener::class,
    DirtiesContextBeforeModesTestExecutionListener::class,
    DependencyInjectionTestExecutionListener::class,
    DirtiesContextTestExecutionListener::class,
    TransactionalTestExecutionListener::class,
    SqlScriptsTestExecutionListener::class
)
class MyTest {
    // class body...
}
```

The challenge with this approach is that it requires that the developer know exactly which listeners are registered by default. Moreover, the set of default listeners can change from release to release — for example, `SqlScriptsTestExecutionListener` was introduced in Spring Framework 4.1, and `DirtiesContextBeforeModesTestExecutionListener` was introduced in Spring Framework 4.2.

Furthermore, third-party frameworks like Spring Boot and Spring Security register their own default `TestExecutionListener` implementations by using the aforementioned automatic discovery mechanism.

To avoid having to be aware of and re-declare all default listeners, you can set the `mergeMode` attribute of `@TestExecutionListeners` to `MergeMode.MERGE_WITH_DEFAULTS`. `MERGE_WITH_DEFAULTS` indicates that locally declared listeners should be merged with the default listeners. The merging algorithm ensures that duplicates are removed from the list and that the resulting set of merged listeners is sorted according to the semantics of `AnnotationAwareOrderComparator`, as described in Ordering `TestExecutionListener` Implementations. If a listener implements `Ordered` or is annotated with `@Order`, it can influence the position in which it is merged with the defaults. Otherwise, locally declared listeners are appended to the list of default listeners when merged.

For example, if the `MyCustomTestExecutionListener` class in the previous example configures its `order` value (for example, `500`) to be less than the order of the `ServletTestExecutionListener` (which happens to be `1000`), the `MyCustomTestExecutionListener` can then be automatically merged with the list of defaults in front of the `ServletTestExecutionListener`, and the previous example could be replaced with the following:

Java

```
@ContextConfiguration
@TestExecutionListeners(
    listeners = MyCustomTestExecutionListener.class,
    mergeMode = MERGE_WITH_DEFAULTS
)
class MyTest {
    // class body...
}
```

Kotlin

```
@ContextConfiguration
@TestExecutionListeners(
        listeners = [MyCustomTestExecutionListener::class],
        mergeMode = MERGE_WITH_DEFAULTS
)
class MyTest {
    // class body...
}
```

3.5.4. Application Events

Since Spring Framework 5.3.3, the TestContext framework provides support for recording application events published in the `ApplicationContext` so that assertions can be performed against those events within tests. All events published during the execution of a single test are made available via the `ApplicationEvents` API which allows you to process the events as a `java.util.Stream`.

To use `ApplicationEvents` in your tests, do the following.

- Ensure that your test class is annotated or meta-annotated with `@RecordApplicationEvents`.
- Ensure that the `ApplicationEventsTestExecutionListener` is registered. Note, however, that `ApplicationEventsTestExecutionListener` is registered by default and only needs to be manually registered if you have custom configuration via `@TestExecutionListeners` that does not include the default listeners.
- Annotate a field of type `ApplicationEvents` with `@Autowired` and use that instance of `ApplicationEvents` in your test and lifecycle methods (such as `@BeforeEach` and `@AfterEach` methods in JUnit Jupiter).
    - When using the SpringExtension for JUnit Jupiter, you may declare a method parameter of type `ApplicationEvents` in a test or lifecycle method as an alternative to an `@Autowired` field in the test class.

The following test class uses the `SpringExtension` for JUnit Jupiter and AssertJ to assert the types of application events published while invoking a method in a Spring-managed component:

Java

```
@SpringJUnitConfig(/* ... */)
@RecordApplicationEvents ①
class OrderServiceTests {

    @Autowired
    OrderService orderService;

    @Autowired
    ApplicationEvents events; ②

    @Test
    void submitOrder() {
        // Invoke method in OrderService that publishes an event
        orderService.submitOrder(new Order(/* ... */));
        // Verify that an OrderSubmitted event was published
        long numEvents = events.stream(OrderSubmitted.class).count(); ③
        assertThat(numEvents).isEqualTo(1);
    }
}
```

① Annotate the test class with `@RecordApplicationEvents`.

② Inject the `ApplicationEvents` instance for the current test.

③ Use the `ApplicationEvents` API to count how many `OrderSubmitted` events were published.

Kotlin

```
@SpringJUnitConfig(/* ... */)
@RecordApplicationEvents ①
class OrderServiceTests {

    @Autowired
    lateinit var orderService: OrderService

    @Autowired
    lateinit var events: ApplicationEvents ②

    @Test
    fun submitOrder() {
        // Invoke method in OrderService that publishes an event
        orderService.submitOrder(Order(/* ... */))
        // Verify that an OrderSubmitted event was published
        val numEvents = events.stream(OrderSubmitted::class).count() ③
        assertThat(numEvents).isEqualTo(1)
    }
}
```

① Annotate the test class with `@RecordApplicationEvents`.

② Inject the `ApplicationEvents` instance for the current test.

③ Use the `ApplicationEvents` API to count how many `OrderSubmitted` events were published.

See the `ApplicationEvents` javadoc for further details regarding the `ApplicationEvents` API.

3.5.5. Test Execution Events

The `EventPublishingTestExecutionListener` introduced in Spring Framework 5.2 offers an alternative approach to implementing a custom `TestExecutionListener`. Components in the test's `ApplicationContext` can listen to the following events published by the `EventPublishingTestExecutionListener`, each of which corresponds to a method in the `TestExecutionListener` API.

- `BeforeTestClassEvent`
- `PrepareTestInstanceEvent`
- `BeforeTestMethodEvent`
- `BeforeTestExecutionEvent`
- `AfterTestExecutionEvent`
- `AfterTestMethodEvent`
- `AfterTestClassEvent`

These events may be consumed for various reasons, such as resetting mock beans or tracing test execution. One advantage of consuming test execution events rather than implementing a custom `TestExecutionListener` is that test execution events may be consumed by any Spring bean registered

in the test `ApplicationContext`, and such beans may benefit directly from dependency injection and other features of the `ApplicationContext`. In contrast, a `TestExecutionListener` is not a bean in the `ApplicationContext`.

The `EventPublishingTestExecutionListener` is registered by default; however, it only publishes events if the `ApplicationContext` has *already been loaded*. This prevents the `ApplicationContext` from being loaded unnecessarily or too early.

Consequently, a `BeforeTestClassEvent` will not be published until after the `ApplicationContext` has been loaded by another `TestExecutionListener`. For example, with the default set of `TestExecutionListener` implementations registered, a `BeforeTestClassEvent` will not be published for the first test class that uses a particular test `ApplicationContext`, but a `BeforeTestClassEvent` *will* be published for any subsequent test class in the same test suite that uses the same test `ApplicationContext` since the context will already have been loaded when subsequent test classes run (as long as the context has not been removed from the `ContextCache` via `@DirtiesContext` or the max-size eviction policy).

If you wish to ensure that a `BeforeTestClassEvent` is always published for every test class, you need to register a `TestExecutionListener` that loads the `ApplicationContext` in the `beforeTestClass` callback, and that `TestExecutionListener` must be registered *before* the `EventPublishingTestExecutionListener`.

Similarly, if `@DirtiesContext` is used to remove the `ApplicationContext` from the context cache after the last test method in a given test class, the `AfterTestClassEvent` will not be published for that test class.

In order to listen to test execution events, a Spring bean may choose to implement the `org.springframework.context.ApplicationListener` interface. Alternatively, listener methods can be annotated with `@EventListener` and configured to listen to one of the particular event types listed above (see Annotation-based Event Listeners). Due to the popularity of this approach, Spring provides the following dedicated `@EventListener` annotations to simplify registration of test execution event listeners. These annotations reside in the `org.springframework.test.context.event.annotation` package.

- `@BeforeTestClass`
- `@PrepareTestInstance`
- `@BeforeTestMethod`
- `@BeforeTestExecution`
- `@AfterTestExecution`
- `@AfterTestMethod`
- `@AfterTestClass`

Exception Handling

By default, if a test execution event listener throws an exception while consuming an event, that exception will propagate to the underlying testing framework in use (such as JUnit or TestNG). For

example, if the consumption of a `BeforeTestMethodEvent` results in an exception, the corresponding test method will fail as a result of the exception. In contrast, if an asynchronous test execution event listener throws an exception, the exception will not propagate to the underlying testing framework. For further details on asynchronous exception handling, consult the class-level javadoc for `@EventListener`.

Asynchronous Listeners

If you want a particular test execution event listener to process events asynchronously, you can use Spring's regular `@Async` support. For further details, consult the class-level javadoc for `@EventListener`.

3.5.6. Context Management

Each `TestContext` provides context management and caching support for the test instance for which it is responsible. Test instances do not automatically receive access to the configured `ApplicationContext`. However, if a test class implements the `ApplicationContextAware` interface, a reference to the `ApplicationContext` is supplied to the test instance. Note that `AbstractJUnit4SpringContextTests` and `AbstractTestNGSpringContextTests` implement `ApplicationContextAware` and, therefore, provide access to the `ApplicationContext` automatically.

@Autowired ApplicationContext

As an alternative to implementing the `ApplicationContextAware` interface, you can inject the application context for your test class through the `@Autowired` annotation on either a field or setter method, as the following example shows:

Java

```
@SpringJUnitConfig
class MyTest {

    @Autowired ①
    ApplicationContext applicationContext;

    // class body...
}
```

① Injecting the `ApplicationContext`.

Kotlin

```
@SpringJUnitConfig
class MyTest {

    @Autowired ①
    lateinit var applicationContext: ApplicationContext

    // class body...
}
```

① Injecting the `ApplicationContext`.

Similarly, if your test is configured to load a `WebApplicationContext`, you can inject the web application context into your test, as follows:

Java

```
@SpringJUnitWebConfig ①
class MyWebAppTest {

    @Autowired ②
    WebApplicationContext wac;

    // class body...
}
```

① Configuring the `WebApplicationContext`.

② Injecting the `WebApplicationContext`.

Kotlin

```
@SpringJUnitWebConfig ①
class MyWebAppTest {

    @Autowired ②
    lateinit var wac: WebApplicationContext
    // class body...
}
```

① Configuring the `WebApplicationContext`.

② Injecting the `WebApplicationContext`.

Dependency injection by using `@Autowired` is provided by the `DependencyInjectionTestExecutionListener`, which is configured by default (see Dependency Injection of Test Fixtures).

Test classes that use the TestContext framework do not need to extend any particular class or implement a specific interface to configure their application context. Instead, configuration is achieved by declaring the `@ContextConfiguration` annotation at the class level. If your test class does not explicitly declare application context resource locations or component classes, the configured `ContextLoader` determines how to load a context from a default location or default configuration classes. In addition to context resource locations and component classes, an application context can also be configured through application context initializers.

The following sections explain how to use Spring's `@ContextConfiguration` annotation to configure a test `ApplicationContext` by using XML configuration files, Groovy scripts, component classes (typically `@Configuration` classes), or context initializers. Alternatively, you can implement and configure your own custom `SmartContextLoader` for advanced use cases.

- Context Configuration with XML resources
- Context Configuration with Groovy Scripts
- Context Configuration with Component Classes
- Mixing XML, Groovy Scripts, and Component Classes
- Context Configuration with Context Initializers
- Context Configuration Inheritance
- Context Configuration with Environment Profiles
- Context Configuration with Test Property Sources
- Context Configuration with Dynamic Property Sources
- Loading a `WebApplicationContext`
- Context Caching
- Context Hierarchies

Context Configuration with XML resources

To load an `ApplicationContext` for your tests by using XML configuration files, annotate your test class with `@ContextConfiguration` and configure the `locations` attribute with an array that contains the resource locations of XML configuration metadata. A plain or relative path (for example, `context.xml`) is treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash is treated as an absolute classpath location (for example, `/org/example/config.xml`). A path that represents a resource URL (i.e., a path prefixed with `classpath:`, `file:`, `http:`, etc.) is used *as is*.

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "/app-config.xml" and
// "/test-config.xml" in the root of the classpath
@ContextConfiguration(locations={"/app-config.xml", "/test-config.xml"}) ①
class MyTest {
    // class body...
}
```

① Setting the locations attribute to a list of XML files.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "/app-config.xml" and
// "/test-config.xml" in the root of the classpath
@ContextConfiguration("/app-config.xml", "/test-config.xml") ①
class MyTest {
    // class body...
}
```

① Setting the locations attribute to a list of XML files.

`@ContextConfiguration` supports an alias for the `locations` attribute through the standard Java `value` attribute. Thus, if you do not need to declare additional attributes in `@ContextConfiguration`, you can omit the declaration of the `locations` attribute name and declare the resource locations by using the shorthand format demonstrated in the following example:

Java

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-config.xml"}) ①
class MyTest {
    // class body...
}
```

① Specifying XML files without using the `location` attribute.

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextConfiguration("/app-config.xml", "/test-config.xml") ①
class MyTest {
    // class body...
}
```

① Specifying XML files without using the `location` attribute.

If you omit both the `locations` and the `value` attributes from the `@ContextConfiguration` annotation, the TestContext framework tries to detect a default XML resource location. Specifically, `GenericXmlContextLoader` and `GenericXmlWebContextLoader` detect a default location based on the name of the test class. If your class is named `com.example.MyTest`, `GenericXmlContextLoader` loads your application context from `"classpath:com/example/MyTest-context.xml"`. The following example shows how to do so:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTest-context.xml"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTest-context.xml"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Context Configuration with Groovy Scripts

To load an `ApplicationContext` for your tests by using Groovy scripts that use the Groovy Bean Definition DSL, you can annotate your test class with `@ContextConfiguration` and configure the `locations` or `value` attribute with an array that contains the resource locations of Groovy scripts. Resource lookup semantics for Groovy scripts are the same as those described for XML configuration files.

Enabling Groovy script support

Support for using Groovy scripts to load an `ApplicationContext` in the Spring TestContext Framework is enabled automatically if Groovy is on the classpath.

The following example shows how to specify Groovy configuration files:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "/AppConfig.groovy" and
// "/TestConfig.groovy" in the root of the classpath
@ContextConfiguration({"/AppConfig.groovy", "/TestConfig.Groovy"}) ①
class MyTest {
    // class body...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "/AppConfig.groovy" and
// "/TestConfig.groovy" in the root of the classpath
@ContextConfiguration("/AppConfig.groovy", "/TestConfig.Groovy") ①
class MyTest {
    // class body...
}
```

① Specifying the location of Groovy configuration files.

If you omit both the `locations` and `value` attributes from the `@ContextConfiguration` annotation, the

TestContext framework tries to detect a default Groovy script. Specifically, `GenericGroovyXmlContextLoader` and `GenericGroovyXmlWebContextLoader` detect a default location based on the name of the test class. If your class is named `com.example.MyTest`, the Groovy context loader loads your application context from `"classpath:com/example/MyTestContext.groovy"`. The following example shows how to use the default:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from
// "classpath:com/example/MyTestContext.groovy"
@ContextConfiguration ①
class MyTest {
    // class body...
}
```

① Loading configuration from the default location.

Declaring XML configuration and Groovy scripts simultaneously

You can declare both XML configuration files and Groovy scripts simultaneously by using the `locations` or `value` attribute of `@ContextConfiguration`. If the path to a configured resource location ends with `.xml`, it is loaded by using an `XmlBeanDefinitionReader`. Otherwise, it is loaded by using a `GroovyBeanDefinitionReader`.

The following listing shows how to combine both in an integration test:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from
// "/app-config.xml" and "/TestConfig.groovy"
@ContextConfiguration({ "/app-config.xml", "/TestConfig.groovy" })
class MyTest {
    // class body...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from
// "/app-config.xml" and "/TestConfig.groovy"
@ContextConfiguration("/app-config.xml", "/TestConfig.groovy")
class MyTest {
    // class body...
}
```

Context Configuration with Component Classes

To load an `ApplicationContext` for your tests by using component classes (see Java-based container configuration), you can annotate your test class with `@ContextConfiguration` and configure the `classes` attribute with an array that contains references to component classes. The following example shows how to do so:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@ContextConfiguration(classes = {AppConfig.class, TestConfig.class}) ①
class MyTest {
    // class body...
}
```

① Specifying component classes.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from AppConfig and TestConfig
@ContextConfiguration(classes = [AppConfig::class, TestConfig::class]) ①
class MyTest {
    // class body...
}
```

① Specifying component classes.

Component Classes

The term “component class” can refer to any of the following:

- A class annotated with `@Configuration`.
- A component (that is, a class annotated with `@Component`, `@Service`, `@Repository`, or other stereotype annotations).
- A JSR-330 compliant class that is annotated with `jakarta.inject` annotations.
- Any class that contains `@Bean`-methods.
- Any other class that is intended to be registered as a Spring component (i.e., a Spring bean in the `ApplicationContext`), potentially taking advantage of automatic autowiring of a single constructor without the use of Spring annotations.

See the javadoc of `@Configuration` and `@Bean` for further information regarding the configuration and semantics of component classes, paying special attention to the discussion of `@Bean` Lite Mode.

If you omit the `classes` attribute from the `@ContextConfiguration` annotation, the TestContext framework tries to detect the presence of default configuration classes. Specifically, `AnnotationConfigContextLoader` and `AnnotationConfigWebContextLoader` detect all `static` nested classes of the test class that meet the requirements for configuration class implementations, as specified in the `@Configuration` javadoc. Note that the name of the configuration class is arbitrary. In addition, a test class can contain more than one `static` nested configuration class if desired. In the following example, the `OrderServiceTest` class declares a `static` nested configuration class named `Config` that is automatically used to load the `ApplicationContext` for the test class:

Java

```
@SpringJUnitConfig ①
// ApplicationContext will be loaded from the
// static nested Config class
class OrderServiceTest {

    @Configuration
    static class Config {

        // this bean will be injected into the OrderServiceTest class
        @Bean
        OrderService orderService() {
            OrderService orderService = new OrderServiceImpl();
            // set properties, etc.
            return orderService;
        }
    }

    @Autowired
    OrderService orderService;

    @Test
    void testOrderService() {
        // test the orderService
    }

}
```

① Loading configuration information from the nested `Config` class.

Kotlin

```
@SpringJUnitConfig ①
// ApplicationContext will be loaded from the nested Config class
class OrderServiceTest {

    @Autowired
    lateinit var orderService: OrderService

    @Configuration
    class Config {

        // this bean will be injected into the OrderServiceTest class
        @Bean
        fun orderService(): OrderService {
            // set properties, etc.
            return OrderServiceImpl()
        }
    }

    @Test
    fun testOrderService() {
        // test the orderService
    }
}
```

① Loading configuration information from the nested `Config` class.

Mixing XML, Groovy Scripts, and Component Classes

It may sometimes be desirable to mix XML configuration files, Groovy scripts, and component classes (typically `@Configuration` classes) to configure an `ApplicationContext` for your tests. For example, if you use XML configuration in production, you may decide that you want to use `@Configuration` classes to configure specific Spring-managed components for your tests, or vice versa.

Furthermore, some third-party frameworks (such as Spring Boot) provide first-class support for loading an `ApplicationContext` from different types of resources simultaneously (for example, XML configuration files, Groovy scripts, and `@Configuration` classes). The Spring Framework, historically, has not supported this for standard deployments. Consequently, most of the `SmartContextLoader` implementations that the Spring Framework delivers in the `spring-test` module support only one resource type for each test context. However, this does not mean that you cannot use both. One exception to the general rule is that the `GenericGroovyXmlContextLoader` and `GenericGroovyXmlWebContextLoader` support both XML configuration files and Groovy scripts simultaneously. Furthermore, third-party frameworks may choose to support the declaration of both `locations` and `classes` through `@ContextConfiguration`, and, with the standard testing support in the TestContext framework, you have the following options.

If you want to use resource locations (for example, XML or Groovy) and `@Configuration` classes to configure your tests, you must pick one as the entry point, and that one must include or import the

other. For example, in XML or Groovy scripts, you can include `@Configuration` classes by using component scanning or defining them as normal Spring beans, whereas, in a `@Configuration` class, you can use `@ImportResource` to import XML configuration files or Groovy scripts. Note that this behavior is semantically equivalent to how you configure your application in production: In production configuration, you define either a set of XML or Groovy resource locations or a set of `@Configuration` classes from which your production `ApplicationContext` is loaded, but you still have the freedom to include or import the other type of configuration.

Context Configuration with Context Initializers

To configure an `ApplicationContext` for your tests by using context initializers, annotate your test class with `@ContextConfiguration` and configure the `initializers` attribute with an array that contains references to classes that implement `ApplicationContextInitializer`. The declared context initializers are then used to initialize the `ConfigurableApplicationContext` that is loaded for your tests. Note that the concrete `ConfigurableApplicationContext` type supported by each declared initializer must be compatible with the type of `ApplicationContext` created by the `SmartContextLoader` in use (typically a `GenericApplicationContext`). Furthermore, the order in which the initializers are invoked depends on whether they implement Spring's `Ordered` interface or are annotated with Spring's `@Order` annotation or the standard `@Priority` annotation. The following example shows how to use initializers:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@ContextConfiguration(
    classes = TestConfig.class,
    initializers = TestAppCtxInitializer.class) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using a configuration class and an initializer.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from TestConfig
// and initialized by TestAppCtxInitializer
@ContextConfiguration(
        classes = [TestConfig::class],
        initializers = [TestAppCtxInitializer::class]) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using a configuration class and an initializer.

You can also omit the declaration of XML configuration files, Groovy scripts, or component classes

in `@ContextConfiguration` entirely and instead declare only `ApplicationContextInitializer` classes, which are then responsible for registering beans in the context — for example, by programmatically loading bean definitions from XML files or configuration classes. The following example shows how to do so:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
@ContextConfiguration(initializers = EntireAppInitializer.class) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using only an initializer.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be initialized by EntireAppInitializer
// which presumably registers beans in the context
@ContextConfiguration(initializers = [EntireAppInitializer::class]) ①
class MyTest {
    // class body...
}
```

① Specifying configuration by using only an initializer.

Context Configuration Inheritance

`@ContextConfiguration` supports boolean `inheritLocations` and `inheritInitializers` attributes that denote whether resource locations or component classes and context initializers declared by superclasses should be inherited. The default value for both flags is `true`. This means that a test class inherits the resource locations or component classes as well as the context initializers declared by any superclasses. Specifically, the resource locations or component classes for a test class are appended to the list of resource locations or annotated classes declared by superclasses. Similarly, the initializers for a given test class are added to the set of initializers defined by test superclasses. Thus, subclasses have the option of extending the resource locations, component classes, or context initializers.

If the `inheritLocations` or `inheritInitializers` attribute in `@ContextConfiguration` is set to `false`, the resource locations or component classes and the context initializers, respectively, for the test class shadow and effectively replace the configuration defined by superclasses.

As of Spring Framework 5.3, test configuration may also be inherited from enclosing classes. See `@Nested` test class configuration for details.

In the next example, which uses XML resource locations, the `ApplicationContext` for `ExtendedTest` is loaded from `base-config.xml` and `extended-config.xml`, in that order. Beans defined in `extended-`

`config.xml` can, therefore, override (that is, replace) those defined in `base-config.xml`. The following example shows how one class can extend another and use both its own configuration file and the superclass's configuration file:

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "/base-config.xml"
// in the root of the classpath
@ContextConfiguration("/base-config.xml") ①
class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from "/base-config.xml" and
// "/extended-config.xml" in the root of the classpath
@ContextConfiguration("/extended-config.xml") ②
class ExtendedTest extends BaseTest {
    // class body...
}
```

① Configuration file defined in the superclass.

② Configuration file defined in the subclass.

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "/base-config.xml"
// in the root of the classpath
@ContextConfiguration("/base-config.xml") ①
open class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from "/base-config.xml" and
// "/extended-config.xml" in the root of the classpath
@ContextConfiguration("/extended-config.xml") ②
class ExtendedTest : BaseTest() {
    // class body...
}
```

① Configuration file defined in the superclass.

② Configuration file defined in the subclass.

Similarly, in the next example, which uses component classes, the `ApplicationContext` for `ExtendedTest` is loaded from the `BaseConfig` and `ExtendedConfig` classes, in that order. Beans defined in `ExtendedConfig` can, therefore, override (that is, replace) those defined in `BaseConfig`. The following example shows how one class can extend another and use both its own configuration class and the superclass's configuration class:

Java

```
// ApplicationContext will be loaded from BaseConfig
@SpringJUnitConfig(BaseConfig.class) ①
class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig
@SpringJUnitConfig(ExtendedConfig.class) ②
class ExtendedTest extends BaseTest {
    // class body...
}
```

① Configuration class defined in the superclass.

② Configuration class defined in the subclass.

Kotlin

```
// ApplicationContext will be loaded from BaseConfig
@SpringJUnitConfig(BaseConfig::class) ①
open class BaseTest {
    // class body...
}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig
@SpringJUnitConfig(ExtendedConfig::class) ②
class ExtendedTest : BaseTest() {
    // class body...
}
```

① Configuration class defined in the superclass.

② Configuration class defined in the subclass.

In the next example, which uses context initializers, the `ApplicationContext` for `ExtendedTest` is initialized by using `BaseInitializer` and `ExtendedInitializer`. Note, however, that the order in which the initializers are invoked depends on whether they implement Spring's `Ordered` interface or are annotated with Spring's `@Order` annotation or the standard `@Priority` annotation. The following example shows how one class can extend another and use both its own initializer and the superclass's initializer:

Java

```
// ApplicationContext will be initialized by BaseInitializer
@SpringJUnitConfig(initializers = BaseInitializer.class) ①
class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
@SpringJUnitConfig(initializers = ExtendedInitializer.class) ②
class ExtendedTest extends BaseTest {
    // class body...
}
```

① Initializer defined in the superclass.

② Initializer defined in the subclass.

Kotlin

```
// ApplicationContext will be initialized by BaseInitializer
@SpringJUnitConfig(initializers = [BaseInitializer::class]) ①
open class BaseTest {
    // class body...
}

// ApplicationContext will be initialized by BaseInitializer
// and ExtendedInitializer
@SpringJUnitConfig(initializers = [ExtendedInitializer::class]) ②
class ExtendedTest : BaseTest() {
    // class body...
}
```

① Initializer defined in the superclass.

② Initializer defined in the subclass.

Context Configuration with Environment Profiles

The Spring Framework has first-class support for the notion of environments and profiles (AKA "bean definition profiles"), and integration tests can be configured to activate particular bean definition profiles for various testing scenarios. This is achieved by annotating a test class with the `@ActiveProfiles` annotation and supplying a list of profiles that should be activated when loading the `ApplicationContext` for the test.

You can use `@ActiveProfiles` with any implementation of the `SmartContextLoader` SPI, but `@ActiveProfiles` is not supported with implementations of the older `ContextLoader` SPI.

Consider two examples with XML configuration and `@Configuration` classes:

```
<!-- app-config.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <bean id="transferService"
            class="com.bank.service.internal.DefaultTransferService">
        <constructor-arg ref="accountRepository"/>
        <constructor-arg ref="feePolicy"/>
    </bean>

    <bean id="accountRepository"
            class="com.bank.repository.internal.JdbcAccountRepository">
        <constructor-arg ref="dataSource"/>
    </bean>

    <bean id="feePolicy"
        class="com.bank.service.internal.ZeroFeePolicy"/>

    <beans profile="dev">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script
                location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script
                location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
    </beans>

    <beans profile="default">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script
                location="classpath:com/bank/config/sql/schema.sql"/>
        </jdbc:embedded-database>
    </beans>

</beans>
```

Java

```
@ExtendWith(SpringExtension.class)
// ApplicationContext will be loaded from "classpath:/app-config.xml"
@ContextConfiguration("/app-config.xml")
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    TransferService transferService;

    @Test
    void testTransferService() {
        // test the transferService
    }
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
// ApplicationContext will be loaded from "classpath:/app-config.xml"
@ContextConfiguration("/app-config.xml")
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    lateinit var transferService: TransferService

    @Test
    fun testTransferService() {
        // test the transferService
    }
}
```

When `TransferServiceTest` is run, its `ApplicationContext` is loaded from the `app-config.xml` configuration file in the root of the classpath. If you inspect `app-config.xml`, you can see that the `accountRepository` bean has a dependency on a `dataSource` bean. However, `dataSource` is not defined as a top-level bean. Instead, `dataSource` is defined three times: in the `production` profile, in the `dev` profile, and in the `default` profile.

By annotating `TransferServiceTest` with `@ActiveProfiles("dev")`, we instruct the Spring TestContext Framework to load the `ApplicationContext` with the active profiles set to `{"dev"}`. As a result, an embedded database is created and populated with test data, and the `accountRepository` bean is wired with a reference to the development `DataSource`. That is likely what we want in an integration test.

It is sometimes useful to assign beans to a `default` profile. Beans within the default profile are included only when no other profile is specifically activated. You can use this to define “fallback” beans to be used in the application’s default state. For example, you may explicitly provide a data

source for `dev` and `production` profiles, but define an in-memory data source as a default when neither of these is active.

The following code listings demonstrate how to implement the same configuration and integration test with `@Configuration` classes instead of XML:

Java

```
@Configuration
@Profile("dev")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

Kotlin

```
@Configuration
@Profile("dev")
class StandaloneDataConfig {

    @Bean
    fun dataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/bank/config/sql/schema.sql")
                .addScript("classpath:com/bank/config/sql/test-data.sql")
                .build()
    }
}
```

Java

```
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

Kotlin

```
@Configuration
@Profile("production")
class JndiDataConfig {

    @Bean(destroyMethod = "")
    fun dataSource(): DataSource {
        val ctx = InitialContext()
        return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource
    }
}
```

Java

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

Kotlin

```
@Configuration
@Profile("default")
class DefaultDataConfig {

    @Bean
    fun dataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.HSQL)
                .addScript("classpath:com/bank/config/sql/schema.sql")
                .build()
    }
}
```

Java

```
@Configuration
public class TransferServiceConfig {

    @Autowired DataSource dataSource;

    @Bean
    public TransferService transferService() {
        return new DefaultTransferService(accountRepository(), feePolicy());
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public FeePolicy feePolicy() {
        return new ZeroFeePolicy();
    }
}
```

Kotlin

```
@Configuration
class TransferServiceConfig {

    @Autowired
    lateinit var dataSource: DataSource

    @Bean
    fun transferService(): TransferService {
        return DefaultTransferService(accountRepository(), feePolicy())
    }

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }

    @Bean
    fun feePolicy(): FeePolicy {
        return ZeroFeePolicy()
    }
}
```

Java

```
@SpringJUnitConfig({
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
        JndiDataConfig.class,
        DefaultDataConfig.class})
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    TransferService transferService;

    @Test
    void testTransferService() {
        // test the transferService
    }
}
```

Kotlin

```
@SpringJUnitConfig(
        TransferServiceConfig::class,
        StandaloneDataConfig::class,
        JndiDataConfig::class,
        DefaultDataConfig::class)
@ActiveProfiles("dev")
class TransferServiceTest {

    @Autowired
    lateinit var transferService: TransferService

    @Test
    fun testTransferService() {
        // test the transferService
    }
}
```

In this variation, we have split the XML configuration into four independent `@Configuration` classes:

- `TransferServiceConfig`: Acquires a `dataSource` through dependency injection by using `@Autowired`.
- `StandaloneDataConfig`: Defines a `dataSource` for an embedded database suitable for developer tests.
- `JndiDataConfig`: Defines a `dataSource` that is retrieved from JNDI in a production environment.
- `DefaultDataConfig`: Defines a `dataSource` for a default embedded database, in case no profile is active.

As with the XML-based configuration example, we still annotate `TransferServiceTest` with `@ActiveProfiles("dev")`, but this time we specify all four configuration classes by using the `@ContextConfiguration` annotation. The body of the test class itself remains completely unchanged.

It is often the case that a single set of profiles is used across multiple test classes within a given project. Thus, to avoid duplicate declarations of the `@ActiveProfiles` annotation, you can declare `@ActiveProfiles` once on a base class, and subclasses automatically inherit the `@ActiveProfiles` configuration from the base class. In the following example, the declaration of `@ActiveProfiles` (as well as other annotations) has been moved to an abstract superclass, `AbstractIntegrationTest`:

As of Spring Framework 5.3, test configuration may also be inherited from enclosing classes. See `@Nested` test class configuration for details.

Java

```
@SpringJUnitConfig({
        TransferServiceConfig.class,
        StandaloneDataConfig.class,
        JndiDataConfig.class,
        DefaultDataConfig.class})
@ActiveProfiles("dev")
abstract class AbstractIntegrationTest {
}
```

Kotlin

```
@SpringJUnitConfig(
        TransferServiceConfig::class,
        StandaloneDataConfig::class,
        JndiDataConfig::class,
        DefaultDataConfig::class)
@ActiveProfiles("dev")
abstract class AbstractIntegrationTest {
}
```

Java

```
// "dev" profile inherited from superclass
class TransferServiceTest extends AbstractIntegrationTest {

    @Autowired
    TransferService transferService;

    @Test
    void testTransferService() {
        // test the transferService
    }
}
```

Kotlin

```
// "dev" profile inherited from superclass
class TransferServiceTest : AbstractIntegrationTest() {

    @Autowired
    lateinit var transferService: TransferService

    @Test
    fun testTransferService() {
        // test the transferService
    }
}
```

`@ActiveProfiles` also supports an `inheritProfiles` attribute that can be used to disable the inheritance of active profiles, as the following example shows:

Java

```
// "dev" profile overridden with "production"
@ActiveProfiles(profiles = "production", inheritProfiles = false)
class ProductionTransferServiceTest extends AbstractIntegrationTest {
    // test body
}
```

Kotlin

```
// "dev" profile overridden with "production"
@ActiveProfiles("production", inheritProfiles = false)
class ProductionTransferServiceTest : AbstractIntegrationTest() {
    // test body
}
```

Furthermore, it is sometimes necessary to resolve active profiles for tests programmatically instead of declaratively — for example, based on:

- The current operating system.
- Whether tests are being run on a continuous integration build server.
- The presence of certain environment variables.
- The presence of custom class-level annotations.
- Other concerns.

To resolve active bean definition profiles programmatically, you can implement a custom `ActiveProfilesResolver` and register it by using the `resolver` attribute of `@ActiveProfiles`. For further information, see the corresponding javadoc. The following example demonstrates how to implement and register a custom `OperatingSystemActiveProfilesResolver`:

Java

```
// "dev" profile overridden programmatically via a custom resolver
@ActiveProfiles(
        resolver = OperatingSystemActiveProfilesResolver.class,
        inheritProfiles = false)
class TransferServiceTest extends AbstractIntegrationTest {
    // test body
}
```

Kotlin

```
// "dev" profile overridden programmatically via a custom resolver
@ActiveProfiles(
        resolver = OperatingSystemActiveProfilesResolver::class,
        inheritProfiles = false)
class TransferServiceTest : AbstractIntegrationTest() {
    // test body
}
```

Java

```
public class OperatingSystemActiveProfilesResolver implements ActiveProfilesResolver {

    @Override
    public String[] resolve(Class<?> testClass) {
        String profile = ...;
        // determine the value of profile based on the operating system
        return new String[] {profile};
    }
}
```

Kotlin

```
class OperatingSystemActiveProfilesResolver : ActiveProfilesResolver {

    override fun resolve(testClass: Class<*>): Array<String> {
        val profile: String = ...
        // determine the value of profile based on the operating system
        return arrayOf(profile)
    }
}
```

Context Configuration with Test Property Sources

The Spring Framework has first-class support for the notion of an environment with a hierarchy of property sources, and you can configure integration tests with test-specific property sources. In contrast to the `@PropertySource` annotation used on `@Configuration` classes, you can declare the

`@TestPropertySource` annotation on a test class to declare resource locations for test properties files or inlined properties. These test property sources are added to the set of `PropertySources` in the `Environment` for the `ApplicationContext` loaded for the annotated integration test.

You can use `@TestPropertySource` with any implementation of the `SmartContextLoader` SPI, but `@TestPropertySource` is not supported with implementations of the older `ContextLoader` SPI.

Implementations of `SmartContextLoader` gain access to merged test property source values through the `getPropertySourceLocations()` and `getPropertySourceProperties()` methods in `MergedContextConfiguration`.

Declaring Test Property Sources

You can configure test properties files by using the `locations` or `value` attribute of `@TestPropertySource`.

Both traditional and XML-based properties file formats are supported — for example, `"classpath:/com/example/test.properties"` or `"file:///path/to/file.xml"`.

Each path is interpreted as a Spring `Resource`. A plain path (for example, `"test.properties"`) is treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash is treated as an absolute classpath resource (for example: `"/org/example/test.xml"`). A path that references a URL (for example, a path prefixed with `classpath:`, `file:`, or `http:`) is loaded by using the specified resource protocol. Resource location wildcards (such as `*/.properties`) are not permitted: Each location must evaluate to exactly one `.properties` or `.xml` resource.

The following example uses a test properties file:

Java

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Specifying a properties file with an absolute path.

Kotlin

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Specifying a properties file with an absolute path.

You can configure inlined properties in the form of key-value pairs by using the `properties` attribute of `@TestPropertySource`, as shown in the next example. All key-value pairs are added to the enclosing `Environment` as a single test `PropertySource` with the highest precedence.

The supported syntax for key-value pairs is the same as the syntax defined for entries in a Java properties file:

- `key=value`
- `key:value`
- `key value`

The following example sets two inlined properties:

Java

```
@ContextConfiguration
@TestPropertySource(properties = {"timezone = GMT", "port: 4242"}) ①
class MyIntegrationTests {
    // class body...
}
```

① Setting two properties by using two variations of the key-value syntax.

Kotlin

```
@ContextConfiguration
@TestPropertySource(properties = ["timezone = GMT", "port: 4242"]) ①
class MyIntegrationTests {
    // class body...
}
```

① Setting two properties by using two variations of the key-value syntax.

As of Spring Framework 5.2, `@TestPropertySource` can be used as *repeatable annotation*. That means that you can have multiple declarations of `@TestPropertySource` on a single test class, with the `locations` and `properties` from later `@TestPropertySource` annotations overriding those from previous `@TestPropertySource` annotations.

In addition, you may declare multiple composed annotations on a test class that are each meta-annotated with `@TestPropertySource`, and all of those `@TestPropertySource` declarations will contribute to your test property sources.

Directly present `@TestPropertySource` annotations always take precedence over meta-present `@TestPropertySource` annotations. In other words, `locations` and `properties` from a directly present `@TestPropertySource` annotation will override the `locations` and `properties` from a `@TestPropertySource` annotation used as a meta-annotation.

Default Properties File Detection

If `@TestPropertySource` is declared as an empty annotation (that is, without explicit values for the `locations` or `properties` attributes), an attempt is made to detect a default properties file relative to the class that declared the annotation. For example, if the annotated test class is `com.example.MyTest`, the corresponding default properties file is `classpath:com/example/MyTest.properties`. If the default cannot be detected, an `IllegalStateException` is thrown.

Precedence

Test properties have higher precedence than those defined in the operating system's environment, Java system properties, or property sources added by the application declaratively by using `@PropertySource` or programmatically. Thus, test properties can be used to selectively override properties loaded from system and application property sources. Furthermore, inlined properties have higher precedence than properties loaded from resource locations. Note, however, that properties registered via `@DynamicPropertySource` have higher precedence than those loaded via `@TestPropertySource`.

In the next example, the `timezone` and `port` properties and any properties defined in `"/test.properties"` override any properties of the same name that are defined in system and application property sources. Furthermore, if the `"/test.properties"` file defines entries for the `timezone` and `port` properties those are overridden by the inlined properties declared by using the `properties` attribute. The following example shows how to specify properties both in a file and inline:

Java

```
@ContextConfiguration
@TestPropertySource(
    locations = "/test.properties",
    properties = {"timezone = GMT", "port: 4242"}
)
class MyIntegrationTests {
    // class body...
}
```

Kotlin

```
@ContextConfiguration
@TestPropertySource("/test.properties",
        properties = ["timezone = GMT", "port: 4242"]
)
class MyIntegrationTests {
    // class body...
}
```

Inheriting and Overriding Test Property Sources

`@TestPropertySource` supports boolean `inheritLocations` and `inheritProperties` attributes that denote whether resource locations for properties files and inlined properties declared by

superclasses should be inherited. The default value for both flags is `true`. This means that a test class inherits the locations and inlined properties declared by any superclasses. Specifically, the locations and inlined properties for a test class are appended to the locations and inlined properties declared by superclasses. Thus, subclasses have the option of extending the locations and inlined properties. Note that properties that appear later shadow (that is, override) properties of the same name that appear earlier. In addition, the aforementioned precedence rules apply for inherited test property sources as well.

If the `inheritLocations` or `inheritProperties` attribute in `@TestPropertySource` is set to `false`, the locations or inlined properties, respectively, for the test class shadow and effectively replace the configuration defined by superclasses.

As of Spring Framework 5.3, test configuration may also be inherited from enclosing classes. See `@Nested` test class configuration for details.

In the next example, the `ApplicationContext` for `BaseTest` is loaded by using only the `base.properties` file as a test property source. In contrast, the `ApplicationContext` for `ExtendedTest` is loaded by using the `base.properties` and `extended.properties` files as test property source locations. The following example shows how to define properties in both a subclass and its superclass by using `properties` files:

Java

```
@TestPropertySource("base.properties")
@ContextConfiguration
class BaseTest {
    // ...
}

@TestPropertySource("extended.properties")
@ContextConfiguration
class ExtendedTest extends BaseTest {
    // ...
}
```

Kotlin

```
@TestPropertySource("base.properties")
@ContextConfiguration
open class BaseTest {
    // ...
}

@TestPropertySource("extended.properties")
@ContextConfiguration
class ExtendedTest : BaseTest() {
    // ...
}
```

In the next example, the `ApplicationContext` for `BaseTest` is loaded by using only the inlined `key1` property. In contrast, the `ApplicationContext` for `ExtendedTest` is loaded by using the inlined `key1` and `key2` properties. The following example shows how to define properties in both a subclass and its superclass by using inline properties:

Java

```
@TestPropertySource(properties = "key1 = value1")
@ContextConfiguration
class BaseTest {
    // ...
}

@TestPropertySource(properties = "key2 = value2")
@ContextConfiguration
class ExtendedTest extends BaseTest {
    // ...
}
```

Kotlin

```
@TestPropertySource(properties = ["key1 = value1"])
@ContextConfiguration
open class BaseTest {
    // ...
}

@TestPropertySource(properties = ["key2 = value2"])
@ContextConfiguration
class ExtendedTest : BaseTest() {
    // ...
}
```

Context Configuration with Dynamic Property Sources

As of Spring Framework 5.2.5, the TestContext framework provides support for *dynamic* properties via the `@DynamicPropertySource` annotation. This annotation can be used in integration tests that need to add properties with dynamic values to the set of `PropertySources` in the `Environment` for the `ApplicationContext` loaded for the integration test.

The `@DynamicPropertySource` annotation and its supporting infrastructure were originally designed to allow properties from Testcontainers based tests to be exposed easily to Spring integration tests. However, this feature may also be used with any form of external resource whose lifecycle is maintained outside the test's `ApplicationContext`.

In contrast to the `@TestPropertySource` annotation that is applied at the class level, `@DynamicPropertySource` must be applied to a `static` method that accepts a single `DynamicPropertyRegistry` argument which is used to add *name-value* pairs to the `Environment`. Values

are dynamic and provided via a `Supplier` which is only invoked when the property is resolved. Typically, method references are used to supply values, as can be seen in the following example which uses the Testcontainers project to manage a Redis container outside of the Spring `ApplicationContext`. The IP address and port of the managed Redis container are made available to components within the test's `ApplicationContext` via the `redis.host` and `redis.port` properties. These properties can be accessed via Spring's `Environment` abstraction or injected directly into Spring-managed components – for example, via `@Value("${redis.host}")` and `@Value("${redis.port}")`, respectively.

If you use `@DynamicPropertySource` in a base class and discover that tests in subclasses fail because the dynamic properties change between subclasses, you may need to annotate your base class with `@DirtiesContext` to ensure that each subclass gets its own `ApplicationContext` with the correct dynamic properties.

Java

```
@SpringJUnitConfig(/* ... */)
@Testcontainers
class ExampleIntegrationTests {

    @Container
    static RedisContainer redis = new RedisContainer();

    @DynamicPropertySource
    static void redisProperties(DynamicPropertyRegistry registry) {
        registry.add("redis.host", redis::getHost);
        registry.add("redis.port", redis::getMappedPort);
    }

    // tests ...

}
```

Kotlin

```
@SpringJUnitConfig(/* ... */)
@Testcontainers
class ExampleIntegrationTests {

    companion object {

        @Container
        @JvmStatic
        val redis: RedisContainer = RedisContainer()

        @DynamicPropertySource
        @JvmStatic
        fun redisProperties(registry: DynamicPropertyRegistry) {
            registry.add("redis.host", redis::getHost)
            registry.add("redis.port", redis::getMappedPort)
        }
    }

    // tests ...

}
```

Precedence

Dynamic properties have higher precedence than those loaded from `@TestPropertySource`, the operating system's environment, Java system properties, or property sources added by the application declaratively by using `@PropertySource` or programmatically. Thus, dynamic properties can be used to selectively override properties loaded via `@TestPropertySource`, system property sources, and application property sources.

Loading a `WebApplicationContext`

To instruct the TestContext framework to load a `WebApplicationContext` instead of a standard `ApplicationContext`, you can annotate the respective test class with `@WebAppConfiguration`.

The presence of `@WebAppConfiguration` on your test class instructs the TestContext framework (TCF) that a `WebApplicationContext` (WAC) should be loaded for your integration tests. In the background, the TCF makes sure that a `MockServletContext` is created and supplied to your test's WAC. By default, the base resource path for your `MockServletContext` is set to `src/main/webapp`. This is interpreted as a path relative to the root of your JVM (normally the path to your project). If you are familiar with the directory structure of a web application in a Maven project, you know that `src/main/webapp` is the default location for the root of your WAR. If you need to override this default, you can provide an alternate path to the `@WebAppConfiguration` annotation (for example, `@WebAppConfiguration("src/test/webapp")`). If you wish to reference a base resource path from the classpath instead of the file system, you can use Spring's `classpath:` prefix.

Note that Spring's testing support for `WebApplicationContext` implementations is on par with its support for standard `ApplicationContext` implementations. When testing with a

`WebApplicationContext`, you are free to declare XML configuration files, Groovy scripts, or `@Configuration` classes by using `@ContextConfiguration`. You are also free to use any other test annotations, such as `@ActiveProfiles`, `@TestExecutionListeners`, `@Sql`, `@Rollback`, and others.

The remaining examples in this section show some of the various configuration options for loading a `WebApplicationContext`. The following example shows the TestContext framework's support for convention over configuration:

Java

```
@ExtendWith(SpringExtension.class)

// defaults to "file:src/main/webapp"
@WebAppConfiguration

// detects "WacTests-context.xml" in the same package
// or static nested @Configuration classes
@ContextConfiguration
class WacTests {
    //...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)

// defaults to "file:src/main/webapp"
@WebAppConfiguration

// detects "WacTests-context.xml" in the same package
// or static nested @Configuration classes
@ContextConfiguration
class WacTests {
    //...
}
```

If you annotate a test class with `@WebAppConfiguration` without specifying a resource base path, the resource path effectively defaults to `file:src/main/webapp`. Similarly, if you declare `@ContextConfiguration` without specifying resource `locations`, component `classes`, or context `initializers`, Spring tries to detect the presence of your configuration by using conventions (that is, `WacTests-context.xml` in the same package as the `WacTests` class or static nested `@Configuration` classes).

The following example shows how to explicitly declare a resource base path with `@WebAppConfiguration` and an XML resource location with `@ContextConfiguration`:

Java

```
@ExtendWith(SpringExtension.class)

// file system resource
@WebAppConfiguration("webapp")

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")
class WacTests {
    //...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)

// file system resource
@WebAppConfiguration("webapp")

// classpath resource
@ContextConfiguration("/spring/test-servlet-config.xml")
class WacTests {
    //...
}
```

The important thing to note here is the different semantics for paths with these two annotations. By default, `@WebAppConfiguration` resource paths are file system based, whereas `@ContextConfiguration` resource locations are classpath based.

The following example shows that we can override the default resource semantics for both annotations by specifying a Spring resource prefix:

Java

```
@ExtendWith(SpringExtension.class)

// classpath resource
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")
class WacTests {
    //...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)

// classpath resource
@WebAppConfiguration("classpath:test-web-resources")

// file system resource
@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")
class WacTests {
    //...
}
```

Contrast the comments in this example with the previous example.

Working with Web Mocks

To provide comprehensive web testing support, the TestContext framework has a `ServletTestExecutionListener` that is enabled by default. When testing against a `WebApplicationContext`, this `TestExecutionListener` sets up default thread-local state by using Spring Web's `RequestContextHolder` before each test method and creates a `MockHttpServletRequest`, a `MockHttpServletResponse`, and a `ServletWebRequest` based on the base resource path configured with `@WebAppConfiguration`. `ServletTestExecutionListener` also ensures that the `MockHttpServletResponse` and `ServletWebRequest` can be injected into the test instance, and, once the test is complete, it cleans up thread-local state.

Once you have a `WebApplicationContext` loaded for your test, you might find that you need to interact with the web mocks — for example, to set up your test fixture or to perform assertions after invoking your web component. The following example shows which mocks can be autowired into your test instance. Note that the `WebApplicationContext` and `MockServletContext` are both cached across the test suite, whereas the other mocks are managed per test method by the `ServletTestExecutionListener`.

Java

```
@SpringJUnitWebConfig
class WacTests {

    @Autowired
    WebApplicationContext wac; // cached

    @Autowired
    MockServletContext servletContext; // cached

    @Autowired
    MockHttpSession session;

    @Autowired
    MockHttpServletRequest request;

    @Autowired
    MockHttpServletResponse response;

    @Autowired
    ServletWebRequest webRequest;

    //...
}
```

Kotlin

```
@SpringJUnitWebConfig
class WacTests {

    @Autowired
    lateinit var wac: WebApplicationContext // cached

    @Autowired
    lateinit var servletContext: MockServletContext // cached

    @Autowired
    lateinit var session: MockHttpSession

    @Autowired
    lateinit var request: MockHttpServletRequest

    @Autowired
    lateinit var response: MockHttpServletResponse

    @Autowired
    lateinit var webRequest: ServletWebRequest

    //...
}
```

Context Caching

Once the TestContext framework loads an `ApplicationContext` (or `WebApplicationContext`) for a test, that context is cached and reused for all subsequent tests that declare the same unique context configuration within the same test suite. To understand how caching works, it is important to understand what is meant by "unique" and "test suite."

An `ApplicationContext` can be uniquely identified by the combination of configuration parameters that is used to load it. Consequently, the unique combination of configuration parameters is used to generate a key under which the context is cached. The TestContext framework uses the following configuration parameters to build the context cache key:

- `locations` (from `@ContextConfiguration`)
- `classes` (from `@ContextConfiguration`)
- `contextInitializerClasses` (from `@ContextConfiguration`)
- `contextCustomizers` (from `ContextCustomizerFactory`) – this includes `@DynamicPropertySource` methods as well as various features from Spring Boot's testing support such as `@MockBean` and `@SpyBean`.
- `contextLoader` (from `@ContextConfiguration`)
- `parent` (from `@ContextHierarchy`)
- `activeProfiles` (from `@ActiveProfiles`)

- `propertySourceLocations` (from `@TestPropertySource`)
- `propertySourceProperties` (from `@TestPropertySource`)
- `resourceBasePath` (from `@WebAppConfiguration`)

For example, if `TestClassA` specifies `{"app-config.xml", "test-config.xml"}` for the `locations` (or `value`) attribute of `@ContextConfiguration`, the TestContext framework loads the corresponding `ApplicationContext` and stores it in a `static` context cache under a key that is based solely on those locations. So, if `TestClassB` also defines `{"app-config.xml", "test-config.xml"}` for its locations (either explicitly or implicitly through inheritance) but does not define `@WebAppConfiguration`, a different `ContextLoader`, different active profiles, different context initializers, different test property sources, or a different parent context, then the same `ApplicationContext` is shared by both test classes. This means that the setup cost for loading an application context is incurred only once (per test suite), and subsequent test execution is much faster.

Test suites and forked processes

The Spring TestContext framework stores application contexts in a static cache. This means that the context is literally stored in a `static` variable. In other words, if tests run in separate processes, the static cache is cleared between each test execution, which effectively disables the caching mechanism.

To benefit from the caching mechanism, all tests must run within the same process or test suite. This can be achieved by executing all tests as a group within an IDE. Similarly, when executing tests with a build framework such as Ant, Maven, or Gradle, it is important to make sure that the build framework does not fork between tests. For example, if the `forkMode` for the Maven Surefire plug-in is set to `always` or `pertest`, the TestContext framework cannot cache application contexts between test classes, and the build process runs significantly more slowly as a result.

The size of the context cache is bounded with a default maximum size of 32. Whenever the maximum size is reached, a least recently used (LRU) eviction policy is used to evict and close stale contexts. You can configure the maximum size from the command line or a build script by setting a JVM system property named `spring.test.context.cache.maxSize`. As an alternative, you can set the same property via the `SpringProperties` mechanism.

Since having a large number of application contexts loaded within a given test suite can cause the suite to take an unnecessarily long time to run, it is often beneficial to know exactly how many contexts have been loaded and cached. To view the statistics for the underlying context cache, you can set the log level for the `org.springframework.test.context.cache` logging category to `DEBUG`.

In the unlikely case that a test corrupts the application context and requires reloading (for example, by modifying a bean definition or the state of an application object), you can annotate your test class or test method with `@DirtiesContext` (see the discussion of `@DirtiesContext` in Spring Testing Annotations). This instructs Spring to remove the context from the cache and rebuild the application context before running the next test that requires the same application context. Note that support for the `@DirtiesContext` annotation is provided by the `DirtiesContextBeforeModesTestExecutionListener` and the `DirtiesContextTestExecutionListener`, which are enabled by default.

ApplicationContext lifecycle and console logging

When you need to debug a test executed with the Spring TestContext Framework, it can be useful to analyze the console output (that is, output to the `SYSOUT` and `SYSERR` streams). Some build tools and IDEs are able to associate console output with a given test; however, some console output cannot be easily associated with a given test.

With regard to console logging triggered by the Spring Framework itself or by components registered in the `ApplicationContext`, it is important to understand the lifecycle of an `ApplicationContext` that has been loaded by the Spring TestContext Framework within a test suite.

The `ApplicationContext` for a test is typically loaded when an instance of the test class is being prepared — for example, to perform dependency injection into `@Autowired` fields of the test instance. This means that any console logging triggered during the initialization of the `ApplicationContext` typically cannot be associated with an individual test method. However, if the context is closed immediately before the execution of a test method according to `@DirtiesContext` semantics, a new instance of the context will be loaded just prior to execution of the test method. In the latter scenario, an IDE or build tool may potentially associate console logging with the individual test method.

The `ApplicationContext` for a test can be closed via one of the following scenarios.

- The context is closed according to `@DirtiesContext` semantics.
- The context is closed because it has been automatically evicted from the cache according to the LRU eviction policy.
- The context is closed via a JVM shutdown hook when the JVM for the test suite terminates.

If the context is closed according to `@DirtiesContext` semantics after a particular test method, an IDE or build tool may potentially associate console logging with the individual test method. If the context is closed according to `@DirtiesContext` semantics after a test class, any console logging triggered during the shutdown of the `ApplicationContext` cannot be associated with an individual test method. Similarly, any console logging triggered during the shutdown phase via a JVM shutdown hook cannot be associated with an individual test method.

When a Spring `ApplicationContext` is closed via a JVM shutdown hook, callbacks executed during the shutdown phase are executed on a thread named `SpringContextShutdownHook`. So, if you wish to disable console logging triggered when the `ApplicationContext` is closed via a JVM shutdown hook, you may be able to register a custom filter with your logging framework that allows you to ignore any logging initiated by that thread.

Context Hierarchies

When writing integration tests that rely on a loaded Spring `ApplicationContext`, it is often sufficient

to test against a single context. However, there are times when it is beneficial or even necessary to test against a hierarchy of `ApplicationContext` instances. For example, if you are developing a Spring MVC web application, you typically have a root `WebApplicationContext` loaded by Spring's `ContextLoaderListener` and a child `WebApplicationContext` loaded by Spring's `DispatcherServlet`. This results in a parent-child context hierarchy where shared components and infrastructure configuration are declared in the root context and consumed in the child context by web-specific components. Another use case can be found in Spring Batch applications, where you often have a parent context that provides configuration for shared batch infrastructure and a child context for the configuration of a specific batch job.

You can write integration tests that use context hierarchies by declaring context configuration with the `@ContextHierarchy` annotation, either on an individual test class or within a test class hierarchy. If a context hierarchy is declared on multiple classes within a test class hierarchy, you can also merge or override the context configuration for a specific, named level in the context hierarchy. When merging configuration for a given level in the hierarchy, the configuration resource type (that is, XML configuration files or component classes) must be consistent. Otherwise, it is perfectly acceptable to have different levels in a context hierarchy configured using different resource types.

The remaining JUnit Jupiter based examples in this section show common configuration scenarios for integration tests that require the use of context hierarchies.

Single test class with context hierarchy

`ControllerIntegrationTests` represents a typical integration testing scenario for a Spring MVC web application by declaring a context hierarchy that consists of two levels, one for the root `WebApplicationContext` (loaded by using the `TestAppConfig` `@Configuration` class) and one for the dispatcher servlet `WebApplicationContext` (loaded by using the `WebConfig` `@Configuration` class). The `WebApplicationContext` that is autowired into the test instance is the one for the child context (that is, the lowest context in the hierarchy). The following listing shows this configuration scenario:

Java

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = TestAppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class ControllerIntegrationTests {

    @Autowired
    WebApplicationContext wac;

    // ...
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@WebAppConfiguration
@ContextHierarchy(
    ContextConfiguration(classes = [TestAppConfig::class]),
    ContextConfiguration(classes = [WebConfig::class]))
class ControllerIntegrationTests {

    @Autowired
    lateinit var wac: WebApplicationContext

    // ...
}
```

Class hierarchy with implicit parent context

The test classes in this example define a context hierarchy within a test class hierarchy. `AbstractWebTests` declares the configuration for a root `WebApplicationContext` in a Spring-powered web application. Note, however, that `AbstractWebTests` does not declare `@ContextHierarchy`. Consequently, subclasses of `AbstractWebTests` can optionally participate in a context hierarchy or follow the standard semantics for `@ContextConfiguration`. `SoapWebServiceTests` and `RestWebServiceTests` both extend `AbstractWebTests` and define a context hierarchy by using `@ContextHierarchy`. The result is that three application contexts are loaded (one for each declaration of `@ContextConfiguration`), and the application context loaded based on the configuration in `AbstractWebTests` is set as the parent context for each of the contexts loaded for the concrete subclasses. The following listing shows this configuration scenario:

Java

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")
public abstract class AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/soap-ws-config.xml"))
public class SoapWebServiceTests extends AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/rest-ws-config.xml"))
public class RestWebServiceTests extends AbstractWebTests {}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")
abstract class AbstractWebTests

@ContextHierarchy(ContextConfiguration("/spring/soap-ws-config.xml"))
class SoapWebServiceTests : AbstractWebTests()

@ContextHierarchy(ContextConfiguration("/spring/rest-ws-config.xml"))
class RestWebServiceTests : AbstractWebTests()
```

Class hierarchy with merged context hierarchy configuration

The classes in this example show the use of named hierarchy levels in order to merge the configuration for specific levels in a context hierarchy. `BaseTests` defines two levels in the hierarchy, `parent` and `child`. `ExtendedTests` extends `BaseTests` and instructs the Spring TestContext Framework to merge the context configuration for the `child` hierarchy level, by ensuring that the names declared in the `name` attribute in `@ContextConfiguration` are both `child`. The result is that three application contexts are loaded: one for `/app-config.xml`, one for `/user-config.xml`, and one for `{"/user-config.xml", "/order-config.xml"}`. As with the previous example, the application context loaded from `/app-config.xml` is set as the parent context for the contexts loaded from `/user-config.xml` and `{"/user-config.xml", "/order-config.xml"}`. The following listing shows this configuration scenario:

Java

```
@ExtendWith(SpringExtension.class)
@ContextHierarchy({
    @ContextConfiguration(name = "parent", locations = "/app-config.xml"),
    @ContextConfiguration(name = "child", locations = "/user-config.xml")
})
class BaseTests {}

@ContextHierarchy(
    @ContextConfiguration(name = "child", locations = "/order-config.xml")
)
class ExtendedTests extends BaseTests {}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextHierarchy(
    ContextConfiguration(name = "parent", locations = ["/app-config.xml"]),
    ContextConfiguration(name = "child", locations = ["/user-config.xml"]))
open class BaseTests {}

@ContextHierarchy(
    ContextConfiguration(name = "child", locations = ["/order-config.xml"])
)
class ExtendedTests : BaseTests() {}
```

Class hierarchy with overridden context hierarchy configuration

In contrast to the previous example, this example demonstrates how to override the configuration for a given named level in a context hierarchy by setting the `inheritLocations` flag in `@ContextConfiguration` to `false`. Consequently, the application context for `ExtendedTests` is loaded only from `/test-user-config.xml` and has its parent set to the context loaded from `/app-config.xml`. The following listing shows this configuration scenario:

Java

```
@ExtendWith(SpringExtension.class)
@ContextHierarchy({
    @ContextConfiguration(name = "parent", locations = "/app-config.xml"),
    @ContextConfiguration(name = "child", locations = "/user-config.xml")
})
class BaseTests {}

@ContextHierarchy(
    @ContextConfiguration(
        name = "child",
        locations = "/test-user-config.xml",
        inheritLocations = false
))
class ExtendedTests extends BaseTests {}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextHierarchy(
    ContextConfiguration(name = "parent", locations = ["/app-config.xml"]),
    ContextConfiguration(name = "child", locations = ["/user-config.xml"]))
open class BaseTests {}

@ContextHierarchy(
        ContextConfiguration(
                name = "child",
                locations = ["/test-user-config.xml"],
                inheritLocations = false
        ))
class ExtendedTests : BaseTests() {}
```

Dirtying a context within a context hierarchy

If you use `@DirtiesContext` in a test whose context is configured as part of a context hierarchy, you can use the `hierarchyMode` flag to control how the context cache is cleared. For further details, see the discussion of `@DirtiesContext` in Spring Testing Annotations and the `@DirtiesContext` javadoc.

3.5.7. Dependency Injection of Test Fixtures

When you use the `DependencyInjectionTestExecutionListener` (which is configured by default), the dependencies of your test instances are injected from beans in the application context that you configured with `@ContextConfiguration` or related annotations. You may use setter injection, field injection, or both, depending on which annotations you choose and whether you place them on setter methods or fields. If you are using JUnit Jupiter you may also optionally use constructor injection (see Dependency Injection with `SpringExtension`). For consistency with Spring's annotation-based injection support, you may also use Spring's `@Autowired` annotation or the `@Inject` annotation from JSR-330 for field and setter injection.

For testing frameworks other than JUnit Jupiter, the TestContext framework does not participate in instantiation of the test class. Thus, the use of `@Autowired` or `@Inject` for constructors has no effect for test classes.

Although field injection is discouraged in production code, field injection is actually quite natural in test code. The rationale for the difference is that you will never instantiate your test class directly. Consequently, there is no need to be able to invoke a `public` constructor or setter method on your test class.

Because `@Autowired` is used to perform autowiring by type, if you have multiple bean definitions of the same type, you cannot rely on this approach for those particular beans. In that case, you can use `@Autowired` in conjunction with `@Qualifier`. You can also choose to use `@Inject` in conjunction with `@Named`. Alternatively, if your test class has access to its `ApplicationContext`, you can perform an explicit lookup by using (for example) a call to `applicationContext.getBean("titleRepository",`

`TitleRepository.class)`.

If you do not want dependency injection applied to your test instances, do not annotate fields or setter methods with `@Autowired` or `@Inject`. Alternatively, you can disable dependency injection altogether by explicitly configuring your class with `@TestExecutionListeners` and omitting `DependencyInjectionTestExecutionListener.class` from the list of listeners.

Consider the scenario of testing a `HibernateTitleRepository` class, as outlined in the Goals section. The next two code listings demonstrate the use of `@Autowired` on fields and setter methods. The application context configuration is presented after all sample code listings.

The dependency injection behavior in the following code listings is not specific to JUnit Jupiter. The same DI techniques can be used in conjunction with any supported testing framework.

The following examples make calls to static assertion methods, such as `assertNotNull()`, but without prepending the call with `Assertions`. In such cases, assume that the method was properly imported through an `import static` declaration that is not shown in the example.

The first code listing shows a JUnit Jupiter based implementation of the test class that uses `@Autowired` for field injection:

Java

```
@ExtendWith(SpringExtension.class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    @Autowired
    HibernateTitleRepository titleRepository;

    @Test
    void findById() {
        Title title = titleRepository.findById(new Long(10));
        assertNotNull(title);
    }
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    @Autowired
    lateinit var titleRepository: HibernateTitleRepository

    @Test
    fun findById() {
        val title = titleRepository.findById(10)
        assertNotNull(title)
    }
}
```

Alternatively, you can configure the class to use `@Autowired` for setter injection, as follows:

Java

```
@ExtendWith(SpringExtension.class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    HibernateTitleRepository titleRepository;

    @Autowired
    void setTitleRepository(HibernateTitleRepository titleRepository) {
        this.titleRepository = titleRepository;
    }

    @Test
    void findById() {
        Title title = titleRepository.findById(new Long(10));
        assertNotNull(title);
    }
}
```

Kotlin

```
@ExtendWith(SpringExtension::class)
// specifies the Spring configuration to load for this test fixture
@ContextConfiguration("repository-config.xml")
class HibernateTitleRepositoryTests {

    // this instance will be dependency injected by type
    lateinit var titleRepository: HibernateTitleRepository

    @Autowired
    fun setTitleRepository(titleRepository: HibernateTitleRepository) {
        this.titleRepository = titleRepository
    }

    @Test
    fun findById() {
        val title = titleRepository.findById(10)
        assertNotNull(title)
    }
}
```

The preceding code listings use the same XML context file referenced by the `@ContextConfiguration` annotation (that is, `repository-config.xml`). The following shows this configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- this bean will be injected into the HibernateTitleRepositoryTests class -->
    <bean id="titleRepository"
class="com.foo.repository.hibernate.HibernateTitleRepository">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <!-- configuration elided for brevity -->
    </bean>

</beans>
```

If you are extending from a Spring-provided test base class that happens to use `@Autowired` on one of its setter methods, you might have multiple beans of the affected type defined in your application context (for example, multiple `DataSource` beans). In such a case, you can override the setter method and use the `@Qualifier` annotation to indicate a specific target bean, as follows (but make sure to delegate to the overridden method in the superclass as well):

Java

```
// ...

    @Autowired
    @Override
    public void setDataSource(@Qualifier("myDataSource") DataSource
dataSource) {
        super.setDataSource(dataSource);
    }

// ...
```

Kotlin

```
// ...

    @Autowired
    override fun setDataSource(@Qualifier("myDataSource") dataSource:
DataSource) {
        super.setDataSource(dataSource)
    }

// ...
```

The specified qualifier value indicates the specific `DataSource` bean to inject, narrowing the set of type matches to a specific bean. Its value is matched against `<qualifier>` declarations within the corresponding `<bean>` definitions. The bean name is used as a fallback qualifier value, so you can effectively also point to a specific bean by name there (as shown earlier, assuming that `myDataSource` is the bean `id`).

3.5.8. Testing Request- and Session-scoped Beans

Spring has supported Request- and session-scoped beans since the early years, and you can test your request-scoped and session-scoped beans by following these steps:

- Ensure that a `WebApplicationContext` is loaded for your test by annotating your test class with `@WebAppConfiguration`.
- Inject the mock request or session into your test instance and prepare your test fixture as appropriate.

- Invoke your web component that you retrieved from the configured `WebApplicationContext` (with dependency injection).
- Perform assertions against the mocks.

The next code snippet shows the XML configuration for a login use case. Note that the `userService` bean has a dependency on a request-scoped `loginAction` bean. Also, the `LoginAction` is instantiated by using SpEL expressions that retrieve the username and password from the current HTTP request. In our test, we want to configure these request parameters through the mock managed by the TestContext framework. The following listing shows the configuration for this use case:

Request-scoped bean configuration

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
            c:loginAction-ref="loginAction"/>

    <bean id="loginAction" class="com.example.LoginAction"
            c:username="#{request.getParameter('user')}"
            c:password="#{request.getParameter('pswd')}"
            scope="request">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

In `RequestScopedBeanTests`, we inject both the `UserService` (that is, the subject under test) and the `MockHttpServletRequest` into our test instance. Within our `requestScope()` test method, we set up our test fixture by setting request parameters in the provided `MockHttpServletRequest`. When the `loginUser()` method is invoked on our `userService`, we are assured that the user service has access to the request-scoped `loginAction` for the current `MockHttpServletRequest` (that is, the one in which we just set parameters). We can then perform assertions against the results based on the known inputs for the username and password. The following listing shows how to do so:

Java

```
@SpringJUnitWebConfig
class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

    @Test
    void requestScope() {
        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        LoginResults results = userService.loginUser();
        // assert results
    }
}
```

Kotlin

```
@SpringJUnitWebConfig
class RequestScopedBeanTests {

    @Autowired lateinit var userService: UserService
    @Autowired lateinit var request: MockHttpServletRequest

    @Test
    fun requestScope() {
        request.setParameter("user", "enigma")
        request.setParameter("pswd", "\$pr!ng")

        val results = userService.loginUser()
        // assert results
    }
}
```

The following code snippet is similar to the one we saw earlier for a request-scoped bean. However, this time, the `userService` bean has a dependency on a session-scoped `userPreferences` bean. Note that the `UserPreferences` bean is instantiated by using a SpEL expression that retrieves the theme from the current HTTP session. In our test, we need to configure a theme in the mock session managed by the TestContext framework. The following example shows how to do so:

Session-scoped bean configuration

```
<beans>

    <bean id="userService" class="com.example.SimpleUserService"
            c:userPreferences-ref="userPreferences" />

    <bean id="userPreferences" class="com.example.UserPreferences"
            c:theme="#{session.getAttribute('theme')}"
            scope="session">
        <aop:scoped-proxy/>
    </bean>

</beans>
```

In `SessionScopedBeanTests`, we inject the `UserService` and the `MockHttpSession` into our test instance. Within our `sessionScope()` test method, we set up our test fixture by setting the expected `theme` attribute in the provided `MockHttpSession`. When the `processUserPreferences()` method is invoked on our `userService`, we are assured that the user service has access to the session-scoped `userPreferences` for the current `MockHttpSession`, and we can perform assertions against the results based on the configured theme. The following example shows how to do so:

Java

```
@SpringJUnitWebConfig
class SessionScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpSession session;

    @Test
    void sessionScope() throws Exception {
        session.setAttribute("theme", "blue");

        Results results = userService.processUserPreferences();
        // assert results
    }
}
```

Kotlin

```
@SpringJUnitWebConfig
class SessionScopedBeanTests {

    @Autowired lateinit var userService: UserService
    @Autowired lateinit var session: MockHttpSession

    @Test
    fun sessionScope() {
        session.setAttribute("theme", "blue")

        val results = userService.processUserPreferences()
        // assert results
    }
}
```

3.5.9. Transaction Management

In the TestContext framework, transactions are managed by the `TransactionalTestExecutionListener`, which is configured by default, even if you do not explicitly declare `@TestExecutionListeners` on your test class. To enable support for transactions, however, you must configure a `PlatformTransactionManager` bean in the `ApplicationContext` that is loaded with `@ContextConfiguration` semantics (further details are provided later). In addition, you must declare Spring's `@Transactional` annotation either at the class or the method level for your tests.

Test-managed Transactions

Test-managed transactions are transactions that are managed declaratively by using the `TransactionalTestExecutionListener` or programmatically by using `TestTransaction` (described later). You should not confuse such transactions with Spring-managed transactions (those managed directly by Spring within the `ApplicationContext` loaded for tests) or application-managed transactions (those managed programmatically within application code that is invoked by tests). Spring-managed and application-managed transactions typically participate in test-managed transactions. However, you should use caution if Spring-managed or application-managed transactions are configured with any propagation type other than `REQUIRED` or `SUPPORTS` (see the discussion on transaction propagation for details).

Preemptive timeouts and test-managed transactions

Caution must be taken when using any form of preemptive timeouts from a testing framework in conjunction with Spring's test-managed transactions.

Specifically, Spring's testing support binds transaction state to the current thread (via a `java.lang.ThreadLocal` variable) *before* the current test method is invoked. If a testing framework invokes the current test method in a new thread in order to support a preemptive timeout, any actions performed within the current test method will *not* be invoked within the test-managed transaction. Consequently, the result of any such actions will not be rolled back with the test-managed transaction. On the contrary, such actions will be committed to the persistent store — for example, a relational database — even though the test-managed transaction is properly rolled back by Spring.

Situations in which this can occur include but are not limited to the following.

- JUnit 4's `@Test(timeout = ...)` support and `TimeOut` rule
- JUnit Jupiter's `assertTimeoutPreemptively(...)` methods in the `org.junit.jupiter.api.Assertions` class
- TestNG's `@Test(timeOut = ...)` support

Enabling and Disabling Transactions

Annotating a test method with `@Transactional` causes the test to be run within a transaction that is, by default, automatically rolled back after completion of the test. If a test class is annotated with `@Transactional`, each test method within that class hierarchy runs within a transaction. Test methods that are not annotated with `@Transactional` (at the class or method level) are not run within a transaction. Note that `@Transactional` is not supported on test lifecycle methods — for example, methods annotated with JUnit Jupiter's `@BeforeAll`, `@BeforeEach`, etc. Furthermore, tests that are annotated with `@Transactional` but have the `propagation` attribute set to `NOT_SUPPORTED` or `NEVER` are not run within a transaction.

Table 16. `@Transactional` attribute support

Attribute	Supported for test-managed transactions
`value` and `transactionManager`	yes
`propagation`	only `Propagation.NOT_SUPPORTED` and `Propagation.NEVER` are supported
`isolation`	no
`timeout`	no
`readOnly`	no
`rollbackFor` and `rollbackForClassName`	no: use `TestTransaction.flagForRollback()` instead
`noRollbackFor` and `noRollbackForClassName`	no: use `TestTransaction.flagForCommit()` instead

Method-level lifecycle methods — for example, methods annotated with JUnit Jupiter's `@BeforeEach` or `@AfterEach` — are run within a test-managed transaction. On the other hand, suite-level and class-level lifecycle methods — for example, methods annotated with JUnit Jupiter's `@BeforeAll` or `@AfterAll` and methods annotated with TestNG's `@BeforeSuite`, `@AfterSuite`, `@BeforeClass`, or `@AfterClass` — are *not* run within a test-managed transaction.

If you need to run code in a suite-level or class-level lifecycle method within a transaction, you may wish to inject a corresponding `PlatformTransactionManager` into your test class and then use that with a `TransactionTemplate` for programmatic transaction management.

Note that `AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests` are preconfigured for transactional support at the class level.

The following example demonstrates a common scenario for writing an integration test for a Hibernate-based `UserRepository`:

Java

```
@SpringJUnitConfig(TestConfig.class)
@Transactional
class HibernateUserRepositoryTests {

    @Autowired
    HibernateUserRepository repository;

    @Autowired
    SessionFactory sessionFactory;

    JdbcTemplate jdbcTemplate;

    @Autowired
    void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    void createUser() {
        // track initial state in test database:
        final int count = countRowsInTable("user");

        User user = new User(...);
        repository.save(user);

        // Manual flush is required to avoid false positive in test
        sessionFactory.getCurrentSession().flush();
        assertNumUsers(count + 1);
    }

    private int countRowsInTable(String tableName) {
        return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    private void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"));
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
@Transactional
class HibernateUserRepositoryTests {

    @Autowired
    lateinit var repository: HibernateUserRepository

    @Autowired
    lateinit var sessionFactory: SessionFactory

    lateinit var jdbcTemplate: JdbcTemplate

    @Autowired
    fun setDataSource(dataSource: DataSource) {
        this.jdbcTemplate = JdbcTemplate(dataSource)
    }

    @Test
    fun createUser() {
        // track initial state in test database:
        val count = countRowsInTable("user")

        val user = User()
        repository.save(user)

        // Manual flush is required to avoid false positive in test
        sessionFactory.getCurrentSession().flush()
        assertNumUsers(count + 1)
    }

    private fun countRowsInTable(tableName: String): Int {
        return JdbcTestUtils.countRowsInTable(jdbcTemplate, tableName)
    }

    private fun assertNumUsers(expected: Int) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"))
    }
}
```

As explained in Transaction Rollback and Commit Behavior, there is no need to clean up the database after the `createUser()` method runs, since any changes made to the database are automatically rolled back by the `TransactionalTestExecutionListener`.

Transaction Rollback and Commit Behavior

By default, test transactions will be automatically rolled back after completion of the test; however, transactional commit and rollback behavior can be configured declaratively via the `@Commit` and

`@Rollback` annotations. See the corresponding entries in the annotation support section for further details.

Programmatic Transaction Management

You can interact with test-managed transactions programmatically by using the static methods in `TestTransaction`. For example, you can use `TestTransaction` within test methods, before methods, and after methods to start or end the current test-managed transaction or to configure the current test-managed transaction for rollback or commit. Support for `TestTransaction` is automatically available whenever the `TransactionalTestExecutionListener` is enabled.

The following example demonstrates some of the features of `TestTransaction`. See the javadoc for `TestTransaction` for further details.

Java

```
@ContextConfiguration(classes = TestConfig.class)
public class ProgrammaticTransactionManagementTests extends
        AbstractTransactionalJUnit4SpringContextTests {

    @Test
    public void transactionalTest() {
        // assert initial state in test database:
        assertNumUsers(2);

        deleteFromTables("user");

        // changes to the database will be committed!
        TestTransaction.flagForCommit();
        TestTransaction.end();
        assertFalse(TestTransaction.isActive());
        assertNumUsers(0);

        TestTransaction.start();
        // perform other actions against the database that will
        // be automatically rolled back after the test completes...
    }

    protected void assertNumUsers(int expected) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"));
    }
}
```

Kotlin

```
@ContextConfiguration(classes = [TestConfig::class])
class ProgrammaticTransactionManagementTests :
AbstractTransactionalJUnit4SpringContextTests() {

    @Test
    fun transactionalTest() {
        // assert initial state in test database:
        assertNumUsers(2)

        deleteFromTables("user")

        // changes to the database will be committed!
        TestTransaction.flagForCommit()
        TestTransaction.end()
        assertFalse(TestTransaction.isActive())
        assertNumUsers(0)

        TestTransaction.start()
        // perform other actions against the database that will
        // be automatically rolled back after the test completes...
    }

    protected fun assertNumUsers(expected: Int) {
        assertEquals("Number of rows in the [user] table.", expected,
countRowsInTable("user"))
    }
}
```

Running Code Outside of a Transaction

Occasionally, you may need to run certain code before or after a transactional test method but outside the transactional context — for example, to verify the initial database state prior to running your test or to verify expected transactional commit behavior after your test runs (if the test was configured to commit the transaction). `TransactionalTestExecutionListener` supports the `@BeforeTransaction` and `@AfterTransaction` annotations for exactly such scenarios. You can annotate any `void` method in a test class or any `void` default method in a test interface with one of these annotations, and the `TransactionalTestExecutionListener` ensures that your before transaction method or after transaction method runs at the appropriate time.

Any before methods (such as methods annotated with JUnit Jupiter's `@BeforeEach`) and any after methods (such as methods annotated with JUnit Jupiter's `@AfterEach`) are run within a transaction. In addition, methods annotated with `@BeforeTransaction` or `@AfterTransaction` are not run for test methods that are not configured to run within a transaction.

Configuring a Transaction Manager

`TransactionalTestExecutionListener` expects a `PlatformTransactionManager` bean to be defined in the Spring `ApplicationContext` for the test. If there are multiple instances of `PlatformTransactionManager` within the test's `ApplicationContext`, you can declare a qualifier by using `@Transactional("myTxMgr")` or `@Transactional(transactionManager = "myTxMgr")`, or `TransactionManagementConfigurer` can be implemented by an `@Configuration` class. Consult the javadoc for `TestContextTransactionUtils.retrieveTransactionManager()` for details on the algorithm used to look up a transaction manager in the test's `ApplicationContext`.

Demonstration of All Transaction-related Annotations

The following JUnit Jupiter based example displays a fictitious integration testing scenario that highlights all transaction-related annotations. The example is not intended to demonstrate best practices but rather to demonstrate how these annotations can be used. See the annotation support section for further information and configuration examples. Transaction management for `@Sql` contains an additional example that uses `@Sql` for declarative SQL script execution with default transaction rollback semantics. The following example shows the relevant annotations:

Java

```
@SpringJUnitConfig
@Transactional(transactionManager = "txMgr")
@Commit
class FictitiousTransactionalTest {

    @BeforeTransaction
    void verifyInitialDatabaseState() {
        // logic to verify the initial state before a transaction is started
    }

    @BeforeEach
    void setUpTestDataWithinTransaction() {
        // set up test data within the transaction
    }

    @Test
    // overrides the class-level @Commit setting
    @Rollback
    void modifyDatabaseWithinTransaction() {
        // logic which uses the test data and modifies database state
    }

    @AfterEach
    void tearDownWithinTransaction() {
        // run "tear down" logic within the transaction
    }

    @AfterTransaction
    void verifyFinalDatabaseState() {
        // logic to verify the final state after transaction has rolled back
    }

}
```

Kotlin

```
@SpringJUnitConfig
@Transactional(transactionManager = "txMgr")
@Commit
class FictitiousTransactionalTest {

    @BeforeTransaction
    fun verifyInitialDatabaseState() {
        // logic to verify the initial state before a transaction is started
    }

    @BeforeEach
    fun setUpTestDataWithinTransaction() {
        // set up test data within the transaction
    }

    @Test
    // overrides the class-level @Commit setting
    @Rollback
    fun modifyDatabaseWithinTransaction() {
        // logic which uses the test data and modifies database state
    }

    @AfterEach
    fun tearDownWithinTransaction() {
        // run "tear down" logic within the transaction
    }

    @AfterTransaction
    fun verifyFinalDatabaseState() {
        // logic to verify the final state after transaction has rolled back
    }

}
```

Avoid false positives when testing ORM code

When you test application code that manipulates the state of a Hibernate session or JPA persistence context, make sure to flush the underlying unit of work within test methods that run that code. Failing to flush the underlying unit of work can produce false positives: Your test passes, but the same code throws an exception in a live, production environment. Note that this applies to any ORM framework that maintains an in-memory unit of work. In the following Hibernate-based example test case, one method demonstrates a false positive, and the other method correctly exposes the results of flushing the session:

Java

```
// ...

@Autowired
SessionFactory sessionFactory;

@Transactional
@Test // no expected exception!
public void falsePositive() {
    updateEntityInHibernateSession();
    // False positive: an exception will be thrown once the Hibernate
    // Session is finally flushed (i.e., in production code)
}

@Transactional
@Test(expected = ...)
public void updateWithSessionFlush() {
    updateEntityInHibernateSession();
    // Manual flush is required to avoid false positive in test
    sessionFactory.getCurrentSession().flush();
}

// ...
```

Kotlin

```
// ...

@Autowired
lateinit var sessionFactory: SessionFactory

@Transactional
@Test // no expected exception!
fun falsePositive() {
    updateEntityInHibernateSession()
    // False positive: an exception will be thrown once the Hibernate
    // Session is finally flushed (i.e., in production code)
}

@Transactional
@Test(expected = ...)
fun updateWithSessionFlush() {
    updateEntityInHibernateSession()
    // Manual flush is required to avoid false positive in test
    sessionFactory.getCurrentSession().flush()
}

// ...
```

The following example shows matching methods for JPA:

Java

```
// ...

@PersistenceContext
EntityManager entityManager;

@Transactional
@Test // no expected exception!
public void falsePositive() {
    updateEntityInJpaPersistenceContext();
    // False positive: an exception will be thrown once the JPA
    // EntityManager is finally flushed (i.e., in production code)
}

@Transactional
@Test(expected = ...)
public void updateWithEntityManagerFlush() {
    updateEntityInJpaPersistenceContext();
    // Manual flush is required to avoid false positive in test
    entityManager.flush();
}

// ...
```

Kotlin

```
// ...

@PersistenceContext
lateinit var entityManager:EntityManager

@Transactional
@Test // no expected exception!
fun falsePositive() {
    updateEntityInJpaPersistenceContext()
    // False positive: an exception will be thrown once the JPA
    // EntityManager is finally flushed (i.e., in production code)
}

@Transactional
@Test(expected = ...)
void updateWithEntityManagerFlush() {
    updateEntityInJpaPersistenceContext()
    // Manual flush is required to avoid false positive in test
    entityManager.flush()
}

// ...
```

Testing ORM entity lifecycle callbacks

Similar to the note about avoiding false positives when testing ORM code, if your application makes use of entity lifecycle callbacks (also known as entity listeners), make sure to flush the underlying unit of work within test methods that run that code. Failing to *flush* or *clear* the underlying unit of work can result in certain lifecycle callbacks not being invoked.

For example, when using JPA, `@PostPersist`, `@PreUpdate`, and `@PostUpdate` callbacks will not be called unless `entityManager.flush()` is invoked after an entity has been saved or updated. Similarly, if an entity is already attached to the current unit of work (associated with the current persistence context), an attempt to reload the entity will not result in a `@PostLoad` callback unless `entityManager.clear()` is invoked before the attempt to reload the entity.

The following example shows how to flush the `EntityManager` to ensure that `@PostPersist` callbacks are invoked when an entity is persisted. An entity listener with a `@PostPersist` callback method has been registered for the `Person` entity used in the example.

Java

```
// ...

@Autowired
JpaPersonRepository repo;

@PersistenceContext
EntityManager entityManager;

@Transactional
@Test
void savePerson() {
    // EntityManager#persist(...) results in @PrePersist but not
@PostPersist
    repo.save(new Person("Jane"));

    // Manual flush is required for @PostPersist callback to be invoked
    entityManager.flush();

    // Test code that relies on the @PostPersist callback
    // having been invoked...
}

// ...
```

Kotlin

```
// ...

@Autowired
lateinit var repo: JpaPersonRepository

@PersistenceContext
lateinit var entityManager: EntityManager

@Transactional
@Test
fun savePerson() {
    // EntityManager#persist(...) results in @PrePersist but not
@PostPersist
    repo.save(Person("Jane"))

    // Manual flush is required for @PostPersist callback to be invoked
    entityManager.flush()

    // Test code that relies on the @PostPersist callback
    // having been invoked...
}

// ...
```

See JpaEntityListenerTests in the Spring Framework test suite for working examples using all JPA lifecycle callbacks.

3.5.10. Executing SQL Scripts

When writing integration tests against a relational database, it is often beneficial to run SQL scripts to modify the database schema or insert test data into tables. The `spring-jdbc` module provides support for *initializing* an embedded or existing database by executing SQL scripts when the Spring `ApplicationContext` is loaded. See Embedded database support and Testing data access logic with an embedded database for details.

Although it is very useful to initialize a database for testing *once* when the `ApplicationContext` is loaded, sometimes it is essential to be able to modify the database *during* integration tests. The following sections explain how to run SQL scripts programmatically and declaratively during integration tests.

Executing SQL scripts programmatically

Spring provides the following options for executing SQL scripts programmatically within integration test methods.

- `org.springframework.jdbc.datasource.init.ScriptUtils`
- `org.springframework.jdbc.datasource.init.ResourceDatabasePopulator`

- `org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests`
- `org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests`

`ScriptUtils` provides a collection of static utility methods for working with SQL scripts and is mainly intended for internal use within the framework. However, if you require full control over how SQL scripts are parsed and run, `ScriptUtils` may suit your needs better than some of the other alternatives described later. See the javadoc for individual methods in `ScriptUtils` for further details.

`ResourceDatabasePopulator` provides an object-based API for programmatically populating, initializing, or cleaning up a database by using SQL scripts defined in external resources. `ResourceDatabasePopulator` provides options for configuring the character encoding, statement separator, comment delimiters, and error handling flags used when parsing and running the scripts. Each of the configuration options has a reasonable default value. See the javadoc for details on default values. To run the scripts configured in a `ResourceDatabasePopulator`, you can invoke either the `populate(Connection)` method to run the populator against a `java.sql.Connection` or the `execute(DataSource)` method to run the populator against a `javax.sql.DataSource`. The following example specifies SQL scripts for a test schema and test data, sets the statement separator to `@@`, and run the scripts against a `DataSource`:

Java

```
@Test
void databaseTest() {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScripts(
            new ClassPathResource("test-schema.sql"),
            new ClassPathResource("test-data.sql"));
    populator.setSeparator("@@");
    populator.execute(this.dataSource);
    // run code that uses the test schema and data
}
```

Kotlin

```
@Test
fun databaseTest() {
    val populator = ResourceDatabasePopulator()
    populator.addScripts(
            ClassPathResource("test-schema.sql"),
            ClassPathResource("test-data.sql"))
    populator.setSeparator("@@")
    populator.execute(dataSource)
    // run code that uses the test schema and data
}
```

Note that `ResourceDatabasePopulator` internally delegates to `ScriptUtils` for parsing and running SQL scripts. Similarly, the `executeSqlScript(..)` methods in `AbstractTransactionalJUnit4SpringContextTests` and `AbstractTransactionalTestNGSpringContextTests`

internally use a `ResourceDatabasePopulator` to run SQL scripts. See the Javadoc for the various `executeSqlScript(..)` methods for further details.

Executing SQL scripts declaratively with @Sql

In addition to the aforementioned mechanisms for running SQL scripts programmatically, you can declaratively configure SQL scripts in the Spring TestContext Framework. Specifically, you can declare the `@Sql` annotation on a test class or test method to configure individual SQL statements or the resource paths to SQL scripts that should be run against a given database before or after an integration test method. Support for `@Sql` is provided by the `SqlScriptsTestExecutionListener`, which is enabled by default.

Method-level `@Sql` declarations override class-level declarations by default. As of Spring Framework 5.2, however, this behavior may be configured per test class or per test method via `@SqlMergeMode`. See Merging and Overriding Configuration with `@SqlMergeMode` for further details.

Path Resource Semantics

Each path is interpreted as a Spring `Resource`. A plain path (for example, `"schema.sql"`) is treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash is treated as an absolute classpath resource (for example, `"/org/example/schema.sql"`). A path that references a URL (for example, a path prefixed with `classpath:`, `file:`, `http:`) is loaded by using the specified resource protocol.

The following example shows how to use `@Sql` at the class level and at the method level within a JUnit Jupiter based integration test class:

Java

```
@SpringJUnitConfig
@Sql("/test-schema.sql")
class DatabaseTests {

    @Test
    void emptySchemaTest() {
        // run code that uses the test schema without any test data
    }

    @Test
    @Sql({"/test-schema.sql", "/test-user-data.sql"})
    void userTest() {
        // run code that uses the test schema and test data
    }
}
```

Kotlin

```
@SpringJUnitConfig
@Sql("/test-schema.sql")
class DatabaseTests {

    @Test
    fun emptySchemaTest() {
        // run code that uses the test schema without any test data
    }

    @Test
    @Sql("/test-schema.sql", "/test-user-data.sql")
    fun userTest() {
        // run code that uses the test schema and test data
    }
}
```

Default Script Detection

If no SQL scripts or statements are specified, an attempt is made to detect a `default` script, depending on where `@Sql` is declared. If a default cannot be detected, an `IllegalStateException` is thrown.

- Class-level declaration: If the annotated test class is `com.example.MyTest`, the corresponding default script is `classpath:com/example/MyTest.sql`.
- Method-level declaration: If the annotated test method is named `testMethod()` and is defined in the class `com.example.MyTest`, the corresponding default script is `classpath:com/example/MyTest.testMethod.sql`.

Declaring Multiple `@Sql` Sets

If you need to configure multiple sets of SQL scripts for a given test class or test method but with different syntax configuration, different error handling rules, or different execution phases per set, you can declare multiple instances of `@Sql`. With Java 8, you can use `@Sql` as a repeatable annotation. Otherwise, you can use the `@SqlGroup` annotation as an explicit container for declaring multiple instances of `@Sql`.

The following example shows how to use `@Sql` as a repeatable annotation with Java 8:

Java

```
@Test
@Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`"))
@Sql("/test-user-data.sql")
void userTest() {
    // run code that uses the test schema and test data
}
```

Kotlin

```
// Repeatable annotations with non-SOURCE retention are not yet supported by Kotlin
```

In the scenario presented in the preceding example, the `test-schema.sql` script uses a different syntax for single-line comments.

The following example is identical to the preceding example, except that the `@Sql` declarations are grouped together within `@SqlGroup`. With Java 8 and above, the use of `@SqlGroup` is optional, but you may need to use `@SqlGroup` for compatibility with other JVM languages such as Kotlin.

Java

```
@Test
@SqlGroup({
    @Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),
    @Sql("/test-user-data.sql")
)}
void userTest() {
    // run code that uses the test schema and test data
}
```

Kotlin

```
@Test
@SqlGroup(
    Sql("/test-schema.sql", config = SqlConfig(commentPrefix = "`")),
    Sql("/test-user-data.sql"))
fun userTest() {
    // Run code that uses the test schema and test data
}
```

Script Execution Phases

By default, SQL scripts are run before the corresponding test method. However, if you need to run a particular set of scripts after the test method (for example, to clean up database state), you can use the `executionPhase` attribute in `@Sql`, as the following example shows:

Java

```
@Test
@Sql(
    scripts = "create-test-data.sql",
    config = @SqlConfig(transactionMode = ISOLATED)
)
@Sql(
    scripts = "delete-test-data.sql",
    config = @SqlConfig(transactionMode = ISOLATED),
    executionPhase = AFTER_TEST_METHOD
)
void userTest() {
    // run code that needs the test data to be committed
    // to the database outside of the test's transaction
}
```

Kotlin

```
@Test
@SqlGroup(
    Sql("create-test-data.sql",
        config = SqlConfig(transactionMode = ISOLATED)),
    Sql("delete-test-data.sql",
        config = SqlConfig(transactionMode = ISOLATED),
        executionPhase = AFTER_TEST_METHOD))
fun userTest() {
    // run code that needs the test data to be committed
    // to the database outside of the test's transaction
}
```

Note that `ISOLATED` and `AFTER_TEST_METHOD` are statically imported from `Sql.TransactionMode` and `Sql.ExecutionPhase`, respectively.

Script Configuration with `@SqlConfig`

You can configure script parsing and error handling by using the `@SqlConfig` annotation. When declared as a class-level annotation on an integration test class, `@SqlConfig` serves as global configuration for all SQL scripts within the test class hierarchy. When declared directly by using the `config` attribute of the `@Sql` annotation, `@SqlConfig` serves as local configuration for the SQL scripts declared within the enclosing `@Sql` annotation. Every attribute in `@SqlConfig` has an implicit default value, which is documented in the javadoc of the corresponding attribute. Due to the rules defined for annotation attributes in the Java Language Specification, it is, unfortunately, not possible to assign a value of `null` to an annotation attribute. Thus, in order to support overrides of inherited global configuration, `@SqlConfig` attributes have an explicit default value of either `""` (for Strings), `{}` (for arrays), or `DEFAULT` (for enumerations). This approach lets local declarations of `@SqlConfig` selectively override individual attributes from global declarations of `@SqlConfig` by providing a value other than `""`, `{}`, or `DEFAULT`. Global `@SqlConfig` attributes are inherited whenever local `@SqlConfig` attributes do not supply an explicit value other than `""`, `{}`, or `DEFAULT`. Explicit local

configuration, therefore, overrides global configuration.

The configuration options provided by `@Sql` and `@SqlConfig` are equivalent to those supported by `ScriptUtils` and `ResourceDatabasePopulator` but are a superset of those provided by the `<jdbc:initialize-database/>` XML namespace element. See the javadoc of individual attributes in `@Sql` and `@SqlConfig` for details.

Transaction management for `@Sql`

By default, the `SqlScriptsTestExecutionListener` infers the desired transaction semantics for scripts configured by using `@Sql`. Specifically, SQL scripts are run without a transaction, within an existing Spring-managed transaction (for example, a transaction managed by the `TransactionalTestExecutionListener` for a test annotated with `@Transactional`), or within an isolated transaction, depending on the configured value of the `transactionMode` attribute in `@SqlConfig` and the presence of a `PlatformTransactionManager` in the test's `ApplicationContext`. As a bare minimum, however, a `javax.sql.DataSource` must be present in the test's `ApplicationContext`.

If the algorithms used by `SqlScriptsTestExecutionListener` to detect a `DataSource` and `PlatformTransactionManager` and infer the transaction semantics do not suit your needs, you can specify explicit names by setting the `dataSource` and `transactionManager` attributes of `@SqlConfig`. Furthermore, you can control the transaction propagation behavior by setting the `transactionMode` attribute of `@SqlConfig` (for example, whether scripts should be run in an isolated transaction). Although a thorough discussion of all supported options for transaction management with `@Sql` is beyond the scope of this reference manual, the javadoc for `@SqlConfig` and `SqlScriptsTestExecutionListener` provide detailed information, and the following example shows a typical testing scenario that uses JUnit Jupiter and transactional tests with `@Sql`:

Java

```
@SpringJUnitConfig(TestDatabaseConfig.class)
@Transactional
class TransactionalSqlScriptsTests {

    final JdbcTemplate jdbcTemplate;

    @Autowired
    TransactionalSqlScriptsTests(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    @Test
    @Sql("/test-data.sql")
    void usersTest() {
        // verify state in test database:
        assertNumUsers(2);
        // run code that uses the test data...
    }

    int countRowsInTable(String tableName) {
        return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);
    }

    void assertNumUsers(int expected) {
        assertEquals(expected, countRowsInTable("user"),
            "Number of rows in the [user] table.");
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestDatabaseConfig::class)
@Transactional
class TransactionalSqlScriptsTests @Autowired constructor(dataSource: DataSource) {

    val jdbcTemplate: JdbcTemplate = JdbcTemplate(dataSource)

    @Test
    @Sql("/test-data.sql")
    fun usersTest() {
        // verify state in test database:
        assertNumUsers(2)
        // run code that uses the test data...
    }

    fun countRowsInTable(tableName: String): Int {
        return JdbcTestUtils.countRowsInTable(jdbcTemplate, tableName)
    }

    fun assertNumUsers(expected: Int) {
        assertEquals(expected, countRowsInTable("user"),
                "Number of rows in the [user] table.")
    }
}
```

Note that there is no need to clean up the database after the `usersTest()` method is run, since any changes made to the database (either within the test method or within the `/test-data.sql` script) are automatically rolled back by the `TransactionalTestExecutionListener` (see transaction management for details).

Merging and Overriding Configuration with `@SqlMergeMode`

As of Spring Framework 5.2, it is possible to merge method-level `@Sql` declarations with class-level declarations. For example, this allows you to provide the configuration for a database schema or some common test data once per test class and then provide additional, use case specific test data per test method. To enable `@Sql` merging, annotate either your test class or test method with `@SqlMergeMode(MERGE)`. To disable merging for a specific test method (or specific test subclass), you can switch back to the default mode via `@SqlMergeMode(OVERRIDE)`. Consult the `@SqlMergeMode` annotation documentation section for examples and further details.

3.5.11. Parallel Test Execution

Spring Framework 5.0 introduced basic support for executing tests in parallel within a single JVM when using the Spring TestContext Framework. In general, this means that most test classes or test methods can be run in parallel without any changes to test code or configuration.

For details on how to set up parallel test execution, see the documentation for your testing framework, build tool, or IDE.

Keep in mind that the introduction of concurrency into your test suite can result in unexpected side effects, strange runtime behavior, and tests that fail intermittently or seemingly randomly. The Spring Team therefore provides the following general guidelines for when not to run tests in parallel.

Do not run tests in parallel if the tests:

- Use Spring Framework's `@DirtiesContext` support.
- Use Spring Boot's `@MockBean` or `@SpyBean` support.
- Use JUnit 4's `@FixMethodOrder` support or any testing framework feature that is designed to ensure that test methods run in a particular order. Note, however, that this does not apply if entire test classes are run in parallel.
- Change the state of shared services or systems such as a database, message broker, filesystem, and others. This applies to both embedded and external systems.

If parallel test execution fails with an exception stating that the `ApplicationContext` for the current test is no longer active, this typically means that the `ApplicationContext` was removed from the `ContextCache` in a different thread.

This may be due to the use of `@DirtiesContext` or due to automatic eviction from the `ContextCache`. If `@DirtiesContext` is the culprit, you either need to find a way to avoid using `@DirtiesContext` or exclude such tests from parallel execution. If the maximum size of the `ContextCache` has been exceeded, you can increase the maximum size of the cache. See the discussion on context caching for details.

Parallel test execution in the Spring TestContext Framework is only possible if the underlying `TestContext` implementation provides a copy constructor, as explained in the javadoc for `TestContext`. The `DefaultTestContext` used in Spring provides such a constructor. However, if you use a third-party library that provides a custom `TestContext` implementation, you need to verify that it is suitable for parallel test execution.

3.5.12. TestContext Framework Support Classes

This section describes the various classes that support the Spring TestContext Framework.

Spring JUnit 4 Runner

The Spring TestContext Framework offers full integration with JUnit 4 through a custom runner (supported on JUnit 4.12 or higher). By annotating test classes with `@RunWith(SpringJUnit4ClassRunner.class)` or the shorter `@RunWith(SpringRunner.class)` variant, developers can implement standard JUnit 4-based unit and integration tests and simultaneously reap the benefits of the TestContext framework, such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on. If you want to use the Spring TestContext Framework with an alternative runner (such as JUnit 4's `Parameterized` runner) or third-party runners (such as the `MockitoJUnitRunner`), you can, optionally, use Spring's support for JUnit rules instead.

The following code listing shows the minimal requirements for configuring a test class to run with the custom Spring `Runner`:

Java

```
@RunWith(SpringRunner.class)
@TestExecutionListeners({})
public class SimpleTest {

    @Test
    public void testMethod() {
        // test logic...
    }
}
```

Kotlin

```
@RunWith(SpringRunner::class)
@TestExecutionListeners
class SimpleTest {

    @Test
    fun testMethod() {
        // test logic...
    }
}
```

In the preceding example, `@TestExecutionListeners` is configured with an empty list, to disable the default listeners, which otherwise would require an `ApplicationContext` to be configured through `@ContextConfiguration`.

Spring JUnit 4 Rules

The `org.springframework.test.context.junit4.rules` package provides the following JUnit 4 rules (supported on JUnit 4.12 or higher):

- `SpringClassRule`
- `SpringMethodRule`

`SpringClassRule` is a JUnit `TestRule` that supports class-level features of the Spring TestContext Framework, whereas `SpringMethodRule` is a JUnit `MethodRule` that supports instance-level and method-level features of the Spring TestContext Framework.

In contrast to the `SpringRunner`, Spring's rule-based JUnit support has the advantage of being independent of any `org.junit.runner.Runner` implementation and can, therefore, be combined with existing alternative runners (such as JUnit 4's `Parameterized`) or third-party runners (such as the `MockitoJUnitRunner`).

To support the full functionality of the TestContext framework, you must combine a

`SpringClassRule` with a `SpringMethodRule`. The following example shows the proper way to declare these rules in an integration test:

Java

```
// Optionally specify a non-Spring Runner via @RunWith(...)
@ContextConfiguration
public class IntegrationTest {

    @ClassRule
    public static final SpringClassRule springClassRule = new SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new SpringMethodRule();

    @Test
    public void testMethod() {
        // test logic...
    }
}
```

Kotlin

```
// Optionally specify a non-Spring Runner via @RunWith(...)
@ContextConfiguration
class IntegrationTest {

    @Rule
    val springMethodRule = SpringMethodRule()

    @Test
    fun testMethod() {
        // test logic...
    }

    companion object {
        @ClassRule
        val springClassRule = SpringClassRule()
    }
}
```

JUnit 4 Support Classes

The `org.springframework.test.context.junit4` package provides the following support classes for JUnit 4-based test cases (supported on JUnit 4.12 or higher):

- `AbstractJUnit4SpringContextTests`
- `AbstractTransactionalJUnit4SpringContextTests`

`AbstractJUnit4SpringContextTests` is an abstract base test class that integrates the Spring TestContext Framework with explicit `ApplicationContext` testing support in a JUnit 4 environment. When you extend `AbstractJUnit4SpringContextTests`, you can access a `protected applicationContext` instance variable that you can use to perform explicit bean lookups or to test the state of the context as a whole.

`AbstractTransactionalJUnit4SpringContextTests` is an abstract transactional extension of `AbstractJUnit4SpringContextTests` that adds some convenience functionality for JDBC access. This class expects a `javax.sql.DataSource` bean and a `PlatformTransactionManager` bean to be defined in the `ApplicationContext`. When you extend `AbstractTransactionalJUnit4SpringContextTests`, you can access a `protected jdbcTemplate` instance variable that you can use to run SQL statements to query the database. You can use such queries to confirm database state both before and after running database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives. As mentioned in JDBC Testing Support, `AbstractTransactionalJUnit4SpringContextTests` also provides convenience methods that delegate to methods in `JdbcTestUtils` by using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalJUnit4SpringContextTests` provides an `executeSqlScript(..)` method for running SQL scripts against the configured `DataSource`.

These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using `@RunWith(SpringRunner.class)` or Spring's JUnit rules.

SpringExtension for JUnit Jupiter

The Spring TestContext Framework offers full integration with the JUnit Jupiter testing framework, introduced in JUnit 5. By annotating test classes with `@ExtendWith(SpringExtension.class)`, you can implement standard JUnit Jupiter-based unit and integration tests and simultaneously reap the benefits of the TestContext framework, such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on.

Furthermore, thanks to the rich extension API in JUnit Jupiter, Spring provides the following features above and beyond the feature set that Spring supports for JUnit 4 and TestNG:

- Dependency injection for test constructors, test methods, and test lifecycle callback methods. See Dependency Injection with `SpringExtension` for further details.
- Powerful support for conditional test execution based on SpEL expressions, environment variables, system properties, and so on. See the documentation for `@EnabledIf` and `@DisabledIf` in Spring JUnit Jupiter Testing Annotations for further details and examples.
- Custom composed annotations that combine annotations from Spring and JUnit Jupiter. See the `@TransactionalDevTestConfig` and `@TransactionalIntegrationTest` examples in Meta-Annotation Support for Testing for further details.

The following code listing shows how to configure a test class to use the `SpringExtension` in conjunction with `@ContextConfiguration`:

Java

```
// Instructs JUnit Jupiter to extend the test with Spring support.
@ExtendWith(SpringExtension.class)
// Instructs Spring to load an ApplicationContext from TestConfig.class
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```

Kotlin

```
// Instructs JUnit Jupiter to extend the test with Spring support.
@ExtendWith(SpringExtension::class)
// Instructs Spring to load an ApplicationContext from TestConfig::class
@ContextConfiguration(classes = [TestConfig::class])
class SimpleTests {

    @Test
    fun testMethod() {
        // test logic...
    }
}
```

Since you can also use annotations in JUnit 5 as meta-annotations, Spring provides the `@SpringJUnitConfig` and `@SpringJUnitWebConfig` composed annotations to simplify the configuration of the test `ApplicationContext` and JUnit Jupiter.

The following example uses `@SpringJUnitConfig` to reduce the amount of configuration used in the previous example:

Java

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load an ApplicationContext from TestConfig.class
@SpringJUnitConfig(TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```

Kotlin

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load an ApplicationContext from TestConfig.class
@SpringJUnitConfig(TestConfig::class)
class SimpleTests {

    @Test
    fun testMethod() {
        // test logic...
    }
}
```

Similarly, the following example uses `@SpringJUnitWebConfig` to create a `WebApplicationContext` for use with JUnit Jupiter:

Java

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load a WebApplicationContext from TestWebConfig.class
@SpringJUnitWebConfig(TestWebConfig.class)
class SimpleWebTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```

Kotlin

```
// Instructs Spring to register the SpringExtension with JUnit
// Jupiter and load a WebApplicationContext from TestWebConfig::class
@SpringJUnitWebConfig(TestWebConfig::class)
class SimpleWebTests {

    @Test
    fun testMethod() {
        // test logic...
    }
}
```

See the documentation for `@SpringJUnitConfig` and `@SpringJUnitWebConfig` in Spring JUnit Jupiter Testing Annotations for further details.

Dependency Injection with `SpringExtension`

`SpringExtension` implements the `ParameterResolver` extension API from JUnit Jupiter, which lets Spring provide dependency injection for test constructors, test methods, and test lifecycle callback

methods.

Specifically, `SpringExtension` can inject dependencies from the test's `ApplicationContext` into test constructors and methods that are annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, `@AfterEach`, `@Test`, `@RepeatedTest`, `@ParameterizedTest`, and others.

Constructor Injection

If a specific parameter in a constructor for a JUnit Jupiter test class is of type `ApplicationContext` (or a sub-type thereof) or is annotated or meta-annotated with `@Autowired`, `@Qualifier`, or `@Value`, Spring injects the value for that specific parameter with the corresponding bean or value from the test's `ApplicationContext`.

Spring can also be configured to autowire all arguments for a test class constructor if the constructor is considered to be *autowirable*. A constructor is considered to be autowirable if one of the following conditions is met (in order of precedence).

- The constructor is annotated with `@Autowired`.
- `@TestConstructor` is present or meta-present on the test class with the `autowireMode` attribute set to `ALL`.
- The default *test constructor autowire mode* has been changed to `ALL`.

See `@TestConstructor` for details on the use of `@TestConstructor` and how to change the global *test constructor autowire mode*.

If the constructor for a test class is considered to be *autowirable*, Spring assumes the responsibility for resolving arguments for all parameters in the constructor. Consequently, no other `ParameterResolver` registered with JUnit Jupiter can resolve parameters for such a constructor.

Constructor injection for test classes must not be used in conjunction with JUnit Jupiter's `@TestInstance(PER_CLASS)` support if `@DirtiesContext` is used to close the test's `ApplicationContext` before or after test methods.

The reason is that `@TestInstance(PER_CLASS)` instructs JUnit Jupiter to cache the test instance between test method invocations. Consequently, the test instance will retain references to beans that were originally injected from an `ApplicationContext` that has been subsequently closed. Since the constructor for the test class will only be invoked once in such scenarios, dependency injection will not occur again, and subsequent tests will interact with beans from the closed `ApplicationContext` which may result in errors.

To use `@DirtiesContext` with "before test method" or "after test method" modes in conjunction with `@TestInstance(PER_CLASS)`, one must configure dependencies from Spring to be supplied via field or setter injection so that they can be re-injected between test method invocations.

In the following example, Spring injects the `OrderService` bean from the `ApplicationContext` loaded

from `TestConfig.class` into the `OrderServiceIntegrationTests` constructor.

Java

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    private final OrderService orderService;

    @Autowired
    OrderServiceIntegrationTests(OrderService orderService) {
        this.orderService = orderService;
    }

    // tests that use the injected OrderService
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class OrderServiceIntegrationTests @Autowired constructor(private val orderService:
OrderService){
    // tests that use the injected OrderService
}
```

Note that this feature lets test dependencies be `final` and therefore immutable.

If the `spring.test.constructor.autowire.mode` property is to `all` (see `@TestConstructor`), we can omit the declaration of `@Autowired` on the constructor in the previous example, resulting in the following.

Java

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    private final OrderService orderService;

    OrderServiceIntegrationTests(OrderService orderService) {
        this.orderService = orderService;
    }

    // tests that use the injected OrderService
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class OrderServiceIntegrationTests(val orderService:OrderService) {
    // tests that use the injected OrderService
}
```

Method Injection

If a parameter in a JUnit Jupiter test method or test lifecycle callback method is of type `ApplicationContext` (or a sub-type thereof) or is annotated or meta-annotated with `@Autowired`, `@Qualifier`, or `@Value`, Spring injects the value for that specific parameter with the corresponding bean from the test's `ApplicationContext`.

In the following example, Spring injects the `OrderService` from the `ApplicationContext` loaded from `TestConfig.class` into the `deleteOrder()` test method:

Java

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    @Test
    void deleteOrder(@Autowired OrderService orderService) {
        // use orderService from the test's ApplicationContext
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class OrderServiceIntegrationTests {

    @Test
    fun deleteOrder(@Autowired orderService: OrderService) {
        // use orderService from the test's ApplicationContext
    }
}
```

Due to the robustness of the `ParameterResolver` support in JUnit Jupiter, you can also have multiple dependencies injected into a single method, not only from Spring but also from JUnit Jupiter itself or other third-party extensions.

The following example shows how to have both Spring and JUnit Jupiter inject dependencies into the `placeOrderRepeatedly()` test method simultaneously.

Java

```
@SpringJUnitConfig(TestConfig.class)
class OrderServiceIntegrationTests {

    @RepeatedTest(10)
    void placeOrderRepeatedly(RepetitionInfo repetitionInfo,
            @Autowired OrderService orderService) {

        // use orderService from the test's ApplicationContext
        // and repetitionInfo from JUnit Jupiter
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class OrderServiceIntegrationTests {

    @RepeatedTest(10)
    fun placeOrderRepeatedly(repetitionInfo:RepetitionInfo, @Autowired
orderService:OrderService) {

        // use orderService from the test's ApplicationContext
        // and repetitionInfo from JUnit Jupiter
    }
}
```

Note that the use of `@RepeatedTest` from JUnit Jupiter lets the test method gain access to the `RepetitionInfo`.

`@Nested` test class configuration

The *Spring TestContext Framework* has supported the use of test-related annotations on `@Nested` test classes in JUnit Jupiter since Spring Framework 5.0; however, until Spring Framework 5.3 class-level test configuration annotations were not *inherited* from enclosing classes like they are from superclasses.

Spring Framework 5.3 introduces first-class support for inheriting test class configuration from enclosing classes, and such configuration will be inherited by default. To change from the default `INHERIT` mode to `OVERRIDE` mode, you may annotate an individual `@Nested` test class with `@NestedTestConfiguration(EnclosingConfiguration.OVERRIDE)`. An explicit `@NestedTestConfiguration` declaration will apply to the annotated test class as well as any of its subclasses and nested classes. Thus, you may annotate a top-level test class with `@NestedTestConfiguration`, and that will apply to all of its nested test classes recursively.

In order to allow development teams to change the default to `OVERRIDE` – for example, for compatibility with Spring Framework 5.0 through 5.2 – the default mode can be changed globally via a JVM system property or a `spring.properties` file in the root of the classpath. See the "Changing

the default enclosing configuration inheritance mode" note for details.

Although the following "Hello World" example is very simplistic, it shows how to declare common configuration on a top-level class that is inherited by its `@Nested` test classes. In this particular example, only the `TestConfig` configuration class is inherited. Each nested test class provides its own set of active profiles, resulting in a distinct `ApplicationContext` for each nested test class (see Context Caching for details). Consult the list of supported annotations to see which annotations can be inherited in `@Nested` test classes.

Java

```
@SpringJUnitConfig(TestConfig.class)
class GreetingServiceTests {

    @Nested
    @ActiveProfiles("lang_en")
    class EnglishGreetings {

        @Test
        void hello(@Autowired GreetingService service) {
            assertThat(service.greetWorld()).isEqualTo("Hello World");
        }
    }

    @Nested
    @ActiveProfiles("lang_de")
    class GermanGreetings {

        @Test
        void hello(@Autowired GreetingService service) {
            assertThat(service.greetWorld()).isEqualTo("Hallo Welt");
        }
    }
}
```

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
class GreetingServiceTests {

    @Nested
    @ActiveProfiles("lang_en")
    inner class EnglishGreetings {

        @Test
        fun hello(@Autowired service:GreetingService) {
            assertThat(service.greetWorld()).isEqualTo("Hello World")
        }
    }

    @Nested
    @ActiveProfiles("lang_de")
    inner class GermanGreetings {

        @Test
        fun hello(@Autowired service:GreetingService) {
            assertThat(service.greetWorld()).isEqualTo("Hallo Welt")
        }
    }
}
```

TestNG Support Classes

The `org.springframework.test.context.testng` package provides the following support classes for TestNG based test cases:

- `AbstractTestNGSpringContextTests`
- `AbstractTransactionalTestNGSpringContextTests`

`AbstractTestNGSpringContextTests` is an abstract base test class that integrates the Spring TestContext Framework with explicit `ApplicationContext` testing support in a TestNG environment. When you extend `AbstractTestNGSpringContextTests`, you can access a `protected applicationContext` instance variable that you can use to perform explicit bean lookups or to test the state of the context as a whole.

`AbstractTransactionalTestNGSpringContextTests` is an abstract transactional extension of `AbstractTestNGSpringContextTests` that adds some convenience functionality for JDBC access. This class expects a `javax.sql.DataSource` bean and a `PlatformTransactionManager` bean to be defined in the `ApplicationContext`. When you extend `AbstractTransactionalTestNGSpringContextTests`, you can access a `protected jdbcTemplate` instance variable that you can use to run SQL statements to query the database. You can use such queries to confirm database state both before and after running database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives. As mentioned in JDBC Testing Support,

`AbstractTransactionalTestNGSpringContextTests` also provides convenience methods that delegate to methods in `JdbcTestUtils` by using the aforementioned `jdbcTemplate`. Furthermore, `AbstractTransactionalTestNGSpringContextTests` provides an `executeSqlScript(..)` method for running SQL scripts against the configured `DataSource`.

These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using `@ContextConfiguration`, `@TestExecutionListeners`, and so on and by manually instrumenting your test class with a `TestContextManager`. See the source code of `AbstractTestNGSpringContextTests` for an example of how to instrument your test class.

3.5.13. Ahead of Time Support for Tests

This chapter covers Spring's Ahead of Time (AOT) support for integration tests using the Spring TestContext Framework.

The testing support extends Spring's core AOT support with the following features.

- Build-time detection of all integration tests in the current project that use the TestContext framework to load an `ApplicationContext`.
  - Provides explicit support for test classes based on JUnit Jupiter and JUnit 4 as well as implicit support for TestNG and other testing frameworks that use Spring's core testing annotations — as long as the tests are run using a JUnit Platform `TestEngine` that is registered for the current project.
- Build-time AOT processing: each unique test `ApplicationContext` in the current project will be refreshed for AOT processing.
- Runtime AOT support: when executing in AOT runtime mode, a Spring integration test will use an AOT-optimized `ApplicationContext` that participates transparently with the context cache.

The `@ContextHierarchy` annotation is currently not supported in AOT mode.

To provide test-specific runtime hints for use within a GraalVM native image, you have the following options.

- Implement a custom `TestRuntimeHintsRegistrar` and register it globally via `META-INF/spring/aot.factories`.
- Implement a custom `RuntimeHintsRegistrar` and register it globally via `META-INF/spring/aot.factories` or locally on a test class via `@ImportRuntimeHints`.
- Annotate a test class with `@Reflective` or `@RegisterReflectionForBinding`.
- See Runtime Hints for details on Spring's core runtime hints and annotation support.

The `TestRuntimeHintsRegistrar` API serves as a companion to the core `RuntimeHintsRegistrar` API. If you need to register global hints for testing support that are not specific to particular test classes, favor implementing `RuntimeHintsRegistrar` over the test-specific API.

If you implement a custom `ContextLoader`, it must implement `AotContextLoader` in order to provide AOT build-time processing and AOT runtime execution support. Note, however, that all context loader implementations provided by the Spring Framework and Spring Boot already implement `AotContextLoader`.

If you implement a custom `TestExecutionListener`, it must implement `AotTestExecutionListener` in order to participate in AOT processing. See the `SqlScriptsTestExecutionListener` in the `spring-test` module for an example.

3.6. WebTestClient

`WebTestClient` is an HTTP client designed for testing server applications. It wraps Spring's WebClient and uses it to perform requests but exposes a testing facade for verifying responses. `WebTestClient` can be used to perform end-to-end HTTP tests. It can also be used to test Spring MVC and Spring WebFlux applications without a running server via mock server request and response objects.

Kotlin users: See this section related to use of the `WebTestClient`.

3.6.1. Setup

To set up a `WebTestClient` you need to choose a server setup to bind to. This can be one of several mock server setup choices or a connection to a live server.

Bind to Controller

This setup allows you to test specific controller(s) via mock request and response objects, without a running server.

For WebFlux applications, use the following which loads infrastructure equivalent to the WebFlux Java config, registers the given controller(s), and creates a WebHandler chain to handle requests:

Java

```
WebTestClient client =
        WebTestClient.bindToController(new TestController()).build();
```

Kotlin

```
val client = WebTestClient.bindToController(TestController()).build()
```

For Spring MVC, use the following which delegates to the StandaloneMockMvcBuilder to load infrastructure equivalent to the WebMvc Java config, registers the given controller(s), and creates an instance of MockMvc to handle requests:

Java

```
WebTestClient client =
        MockMvcWebTestClient.bindToController(new TestController()).build();
```

Kotlin

```
val client = MockMvcWebTestClient.bindToController(TestController()).build()
```

Bind to `ApplicationContext`

This setup allows you to load Spring configuration with Spring MVC or Spring WebFlux infrastructure and controller declarations and use it to handle requests via mock request and response objects, without a running server.

For WebFlux, use the following where the Spring `ApplicationContext` is passed to WebHttpHandlerBuilder to create the WebHandler chain to handle requests:

Java

```
@SpringJUnitConfig(WebConfig.class) ①
class MyTests {

    WebTestClient client;

    @BeforeEach
    void setUp(ApplicationContext context) {  ②
        client = WebTestClient.bindToApplicationContext(context).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Kotlin

```
@SpringJUnitConfig(WebConfig::class) ①
class MyTests {

    lateinit var client: WebTestClient

    @BeforeEach
    fun setUp(context: ApplicationContext) { ②
        client = WebTestClient.bindToApplicationContext(context).build() ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

For Spring MVC, use the following where the Spring `ApplicationContext` is passed to MockMvcBuilders.webAppContextSetup to create a MockMvc instance to handle requests:

Java

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration("classpath:META-INF/web-resources") ①
@ContextHierarchy({
    @ContextConfiguration(classes = RootConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class MyTests {

    @Autowired
    WebApplicationContext wac; ②

    WebTestClient client;

    @BeforeEach
    void setUp() {
        client = MockMvcWebTestClient.bindToApplicationContext(this.wac).build(); ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Kotlin

```
@ExtendWith(SpringExtension.class)
@WebAppConfiguration("classpath:META-INF/web-resources") ①
@ContextHierarchy({
    @ContextConfiguration(classes = RootConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class MyTests {

    @Autowired
    lateinit var wac: WebApplicationContext; ②

    lateinit var client: WebTestClient

    @BeforeEach
    fun setUp() { ②
        client = MockMvcWebTestClient.bindToApplicationContext(wac).build() ③
    }
}
```

① Specify the configuration to load

② Inject the configuration

③ Create the `WebTestClient`

Bind to Router Function

This setup allows you to test functional endpoints via mock request and response objects, without a running server.

For WebFlux, use the following which delegates to `RouterFunctions.toWebHandler` to create a server setup to handle requests:

Java

```
RouterFunction<?> route = ...
client = WebTestClient.bindToRouterFunction(route).build();
```

Kotlin

```
val route: RouterFunction<*> = ...
val client = WebTestClient.bindToRouterFunction(route).build()
```

For Spring MVC there are currently no options to test WebMvc functional endpoints.

Bind to Server

This setup connects to a running server to perform full, end-to-end HTTP tests:

Java

```
client = WebTestClient.bindToServer().baseUrl("http://localhost:8080").build();
```

Kotlin

```
client = WebTestClient.bindToServer().baseUrl("http://localhost:8080").build()
```

Client Config

In addition to the server setup options described earlier, you can also configure client options, including base URL, default headers, client filters, and others. These options are readily available following `bindToServer()`. For all other configuration options, you need to use `configureClient()` to transition from server to client configuration, as follows:

Java

```
client = WebTestClient.bindToController(new TestController())
        .configureClient()
        .baseUrl("/test")
        .build();
```

Kotlin

```
client = WebTestClient.bindToController(TestController())
        .configureClient()
        .baseUrl("/test")
        .build()
```

3.6.2. Writing Tests

`WebTestClient` provides an API identical to WebClient up to the point of performing a request by using `exchange()`. See the WebClient documentation for examples on how to prepare a request with any content including form data, multipart data, and more.

After the call to `exchange()`, `WebTestClient` diverges from the `WebClient` and instead continues with a workflow to verify responses.

To assert the response status and headers, use the following:

Java

```
client.get().uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectStatus().isOk()
    .expectHeader().contentType(MediaType.APPLICATION_JSON);
```

Kotlin

```
client.get().uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectStatus().isOk()
    .expectHeader().contentType(MediaType.APPLICATION_JSON)
```

If you would like for all expectations to be asserted even if one of them fails, you can use `expectAll(..)` instead of multiple chained `expect*(..)` calls. This feature is similar to the *soft assertions* support in AssertJ and the `assertAll()` support in JUnit Jupiter.

Java

```
client.get().uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectAll(
        spec -> spec.expectStatus().isOk(),
        spec -> spec.expectHeader().contentType(MediaType.APPLICATION_JSON)
    );
```

You can then choose to decode the response body through one of the following:

- `expectBody(Class<T>)`: Decode to single object.
- `expectBodyList(Class<T>)`: Decode and collect objects to `List<T>`.
- `expectBody()`: Decode to `byte[]` for JSON Content or an empty body.

And perform assertions on the resulting higher level Object(s):

Java

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBodyList(Person.class).hasSize(3).contains(person);
```

Kotlin

```
import org.springframework.test.web.reactive.server.expectBodyList

client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBodyList<Person>().hasSize(3).contains(person)
```

If the built-in assertions are insufficient, you can consume the object instead and perform any other assertions:

Java

```
import org.springframework.test.web.reactive.server.expectBody

client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .consumeWith(result -> {
            // custom assertions (e.g. AssertJ)...
        });
```

Kotlin

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody<Person>()
        .consumeWith {
            // custom assertions (e.g. AssertJ)...
        }
```

Or you can exit the workflow and obtain an `EntityExchangeResult`:

Java

```
EntityExchangeResult<Person> result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .returnResult();
```

Kotlin

```
import org.springframework.test.web.reactive.server.expectBody

val result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk
        .expectBody<Person>()
        .returnResult()
```

When you need to decode to a target type with generics, look for the overloaded methods that accept `ParameterizedTypeReference` instead of `Class<T>`.

No Content

If the response is not expected to have content, you can assert that as follows:

Java

```
client.post().uri("/persons")
        .body(personMono, Person.class)
        .exchange()
        .expectStatus().isCreated()
        .expectBody().isEmpty();
```

Kotlin

```
client.post().uri("/persons")
        .bodyValue(person)
        .exchange()
        .expectStatus().isCreated()
        .expectBody().isEmpty()
```

If you want to ignore the response content, the following releases the content without any assertions:

Java

```
client.get().uri("/persons/123")
        .exchange()
        .expectStatus().isNotFound()
        .expectBody(Void.class);
```

Kotlin

```
client.get().uri("/persons/123")
        .exchange()
        .expectStatus().isNotFound
        .expectBody<Unit>()
```

JSON Content

You can use `expectBody()` without a target type to perform assertions on the raw content rather than through higher level Object(s).

To verify the full JSON content with JSONAssert:

Java

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .json("{\"name\":\"Jane\"}")
```

Kotlin

```
client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .json("{\"name\":\"Jane\"}")
```

To verify JSON content with JSONPath:

Java

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .jsonPath("$[0].name").isEqualTo("Jane")
        .jsonPath("$[1].name").isEqualTo("Jason");
```

Kotlin

```
client.get().uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectBody()
        .jsonPath("$[0].name").isEqualTo("Jane")
        .jsonPath("$[1].name").isEqualTo("Jason")
```

Streaming Responses

To test potentially infinite streams such as `"text/event-stream"` or `"application/x-ndjson"`, start by verifying the response status and headers, and then obtain a `FluxExchangeResult`:

Java

```
FluxExchangeResult<MyEvent> result = client.get().uri("/events")
        .accept(TEXT_EVENT_STREAM)
        .exchange()
        .expectStatus().isOk()
        .returnResult(MyEvent.class);
```

Kotlin

```
import org.springframework.test.web.reactive.server.returnResult

val result = client.get().uri("/events")
        .accept(TEXT_EVENT_STREAM)
        .exchange()
        .expectStatus().isOk()
        .returnResult<MyEvent>()
```

Now you're ready to consume the response stream with `StepVerifier` from `reactor-test`:

Java

```
Flux<Event> eventFlux = result.getResponseBody();

StepVerifier.create(eventFlux)
        .expectNext(person)
        .expectNextCount(4)
        .consumeNextWith(p -> ...)
        .thenCancel()
        .verify();
```

Kotlin

```
val eventFlux = result.getResponseBody()

StepVerifier.create(eventFlux)
        .expectNext(person)
        .expectNextCount(4)
        .consumeNextWith { p -> ... }
        .thenCancel()
        .verify()
```

MockMvc Assertions

`WebTestClient` is an HTTP client and as such it can only verify what is in the client response including status, headers, and body.

When testing a Spring MVC application with a MockMvc server setup, you have the extra choice to perform further assertions on the server response. To do that start by obtaining an `ExchangeResult` after asserting the body:

Java

```
// For a response with a body
EntityExchangeResult<Person> result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .returnResult();

// For a response without a body
EntityExchangeResult<Void> result = client.get().uri("/path")
        .exchange()
        .expectBody().isEmpty();
```

Kotlin

```
// For a response with a body
val result = client.get().uri("/persons/1")
        .exchange()
        .expectStatus().isOk()
        .expectBody(Person.class)
        .returnResult();

// For a response without a body
val result = client.get().uri("/path")
        .exchange()
        .expectBody().isEmpty();
```

Then switch to MockMvc server response assertions:

Java

```
MockMvcWebTestClient.resultActionsFor(result)
        .andExpect(model().attribute("integer", 3))
        .andExpect(model().attribute("string", "a string value"));
```

Kotlin

```
MockMvcWebTestClient.resultActionsFor(result)
        .andExpect(model().attribute("integer", 3))
        .andExpect(model().attribute("string", "a string value"));
```

3.7. MockMvc

The Spring MVC Test framework, also known as MockMvc, provides support for testing Spring MVC applications. It performs full Spring MVC request handling but via mock request and response objects instead of a running server.

MockMvc can be used on its own to perform requests and verify responses. It can also be used through the WebTestClient where MockMvc is plugged in as the server to handle requests with. The advantage of `WebTestClient` is the option to work with higher level objects instead of raw data as well as the ability to switch to full, end-to-end HTTP tests against a live server and use the same test API.

3.7.1. Overview

You can write plain unit tests for Spring MVC by instantiating a controller, injecting it with dependencies, and calling its methods. However such tests do not verify request mappings, data binding, message conversion, type conversion, validation, and nor do they involve any of the supporting `@InitBinder`, `@ModelAttribute`, or `@ExceptionHandler` methods.

The Spring MVC Test framework, also known as `MockMvc`, aims to provide more complete testing for Spring MVC controllers without a running server. It does that by invoking the `DispatcherServlet` and passing "mock" implementations of the Servlet API from the `spring-test` module which replicates the full Spring MVC request handling without a running server.

MockMvc is a server side test framework that lets you verify most of the functionality of a Spring MVC application using lightweight and targeted tests. You can use it on its own to perform requests and to verify responses, or you can also use it through the WebTestClient API with MockMvc plugged in as the server to handle requests with.

3.7.2. Static Imports

When using MockMvc directly to perform requests, you'll need static imports for:

- `MockMvcBuilders.*`
- `MockMvcRequestBuilders.*`
- `MockMvcResultMatchers.*`
- `MockMvcResultHandlers.*`

An easy way to remember that is search for `MockMvc*`. If using Eclipse be sure to also add the above as "favorite static members" in the Eclipse preferences.

When using MockMvc through the WebTestClient you do not need static imports. The `WebTestClient` provides a fluent API without static imports.

3.7.3. Setup Choices

MockMvc can be setup in one of two ways. One is to point directly to the controllers you want to test and programmatically configure Spring MVC infrastructure. The second is to point to Spring configuration with Spring MVC and controller infrastructure in it.

To set up MockMvc for testing a specific controller, use the following:

Java

```
class MyWebTests {

    MockMvc mockMvc;

    @BeforeEach
    void setup() {
        this.mockMvc = MockMvcBuilders.standaloneSetup(new
AccountController()).build();
    }

    // ...

}
```

Kotlin

```
class MyWebTests {

    lateinit var mockMvc : MockMvc

    @BeforeEach
    fun setup() {
        mockMvc = MockMvcBuilders.standaloneSetup(AccountController()).build()
    }

    // ...

}
```

Or you can also use this setup when testing through the WebTestClient which delegates to the same builder as shown above.

To set up MockMvc through Spring configuration, use the following:

Java

```
@SpringJUnitWebConfig(locations = "my-servlet-context.xml")
class MyWebTests {

    MockMvc mockMvc;

    @BeforeEach
    void setup(WebApplicationContext wac) {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    // ...

}
```

Kotlin

```
@SpringJUnitWebConfig(locations = ["my-servlet-context.xml"])
class MyWebTests {

    lateinit var mockMvc: MockMvc

    @BeforeEach
    fun setup(wac: WebApplicationContext) {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build()
    }

    // ...

}
```

Or you can also use this setup when testing through the WebTestClient which delegates to the same builder as shown above.

Which setup option should you use?

The `webAppContextSetup` loads your actual Spring MVC configuration, resulting in a more complete integration test. Since the TestContext framework caches the loaded Spring configuration, it helps keep tests running fast, even as you introduce more tests in your test suite. Furthermore, you can inject mock services into controllers through Spring configuration to remain focused on testing the web layer. The following example declares a mock service with Mockito:

```
<bean id="accountService" class="org.mockito.Mockito" factory-method="mock">
    <constructor-arg value="org.example.AccountService"/>
</bean>
```

You can then inject the mock service into the test to set up and verify your expectations, as the

following example shows:

Java

```
@SpringJUnitWebConfig(locations = "test-servlet-context.xml")
class AccountTests {

    @Autowired
    AccountService accountService;

    MockMvc mockMvc;

    @BeforeEach
    void setup(WebApplicationContext wac) {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }

    // ...

}
```

Kotlin

```
@SpringJUnitWebConfig(locations = ["test-servlet-context.xml"])
class AccountTests {

    @Autowired
    lateinit var accountService: AccountService

    lateinit mockMvc: MockMvc

    @BeforeEach
    fun setup(wac: WebApplicationContext) {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build()
    }

    // ...

}
```

The `standaloneSetup`, on the other hand, is a little closer to a unit test. It tests one controller at a time. You can manually inject the controller with mock dependencies, and it does not involve loading Spring configuration. Such tests are more focused on style and make it easier to see which controller is being tested, whether any specific Spring MVC configuration is required to work, and so on. The `standaloneSetup` is also a very convenient way to write ad-hoc tests to verify specific behavior or to debug an issue.

As with most “integration versus unit testing” debates, there is no right or wrong answer. However, using the `standaloneSetup` does imply the need for additional `webAppContextSetup` tests in order to

verify your Spring MVC configuration. Alternatively, you can write all your tests with `webAppContextSetup`, in order to always test against your actual Spring MVC configuration.

3.7.4. Setup Features

No matter which MockMvc builder you use, all `MockMvcBuilder` implementations provide some common and very useful features. For example, you can declare an `Accept` header for all requests and expect a status of 200 as well as a `Content-Type` header in all responses, as follows:

Java

```
// static import of MockMvcBuilders.standaloneSetup

MockMvc mockMvc = standaloneSetup(new MusicController())
    .defaultRequest(get("/").accept(MediaType.APPLICATION_JSON))
    .alwaysExpect(status().isOk())
    .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
    .build();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

In addition, third-party frameworks (and applications) can pre-package setup instructions, such as those in a `MockMvcConfigurer`. The Spring Framework has one such built-in implementation that helps to save and re-use the HTTP session across requests. You can use it as follows:

Java

```
// static import of SharedHttpSessionConfigurer.sharedHttpSession

MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TestController())
        .apply(sharedHttpSession())
        .build();

// Use mockMvc to perform requests...
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

See the javadoc for `ConfigurableMockMvcBuilder` for a list of all MockMvc builder features or use the IDE to explore the available options.

3.7.5. Performing Requests

This section shows how to use MockMvc on its own to perform requests and verify responses. If using MockMvc through the `WebTestClient` please see the corresponding section on Writing Tests

instead.

To perform requests that use any HTTP method, as the following example shows:

Java

```
// static import of MockMvcRequestBuilders.*

mockMvc.perform(post("/hotels/{id}", 42).accept(MediaType.APPLICATION_JSON));
```

Kotlin

```
import org.springframework.test.web.servlet.post

mockMvc.post("/hotels/{id}", 42) {
    accept = MediaType.APPLICATION_JSON
}
```

You can also perform file upload requests that internally use `MockMultipartHttpServletRequest` so that there is no actual parsing of a multipart request. Rather, you have to set it up to be similar to the following example:

Java

```
mockMvc.perform(multipart("/doc").file("a1", "ABC".getBytes("UTF-8")));
```

Kotlin

```
import org.springframework.test.web.servlet.multipart

mockMvc.multipart("/doc") {
    file("a1", "ABC".toByteArray(charset("UTF8")))
}
```

You can specify query parameters in URI template style, as the following example shows:

Java

```
mockMvc.perform(get("/hotels?thing={thing}", "somewhere"));
```

Kotlin

```
mockMvc.get("/hotels?thing={thing}", "somewhere")
```

You can also add Servlet request parameters that represent either query or form parameters, as the following example shows:

Java

```
mockMvc.perform(get("/hotels").param("thing", "somewhere"));
```

Kotlin

```
import org.springframework.test.web.servlet.get

mockMvc.get("/hotels") {
    param("thing", "somewhere")
}
```

If application code relies on Servlet request parameters and does not check the query string explicitly (as is most often the case), it does not matter which option you use. Keep in mind, however, that query parameters provided with the URI template are decoded while request parameters provided through the `param(…)` method are expected to already be decoded.

In most cases, it is preferable to leave the context path and the Servlet path out of the request URI. If you must test with the full request URI, be sure to set the `contextPath` and `servletPath` accordingly so that request mappings work, as the following example shows:

Java

```
mockMvc.perform(get("/app/main/hotels/{id}").contextPath("/app").servletPath("/main"))
```

Kotlin

```
import org.springframework.test.web.servlet.get

mockMvc.get("/app/main/hotels/{id}") {
    contextPath = "/app"
    servletPath = "/main"
}
```

In the preceding example, it would be cumbersome to set the `contextPath` and `servletPath` with every performed request. Instead, you can set up default request properties, as the following example shows:

Java

```
class MyWebTests {

    MockMvc mockMvc;

    @BeforeEach
    void setup() {
        mockMvc = standaloneSetup(new AccountController())
            .defaultRequest(get("/")
            .contextPath("/app").servletPath("/main")
            .accept(MediaType.APPLICATION_JSON)).build();
    }
}
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

The preceding properties affect every request performed through the `MockMvc` instance. If the same property is also specified on a given request, it overrides the default value. That is why the HTTP method and URI in the default request do not matter, since they must be specified on every request.

3.7.6. Defining Expectations

You can define expectations by appending one or more `andExpect(..)` calls after performing a request, as the following example shows. As soon as one expectation fails, no other expectations will be asserted.

Java

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

mockMvc.perform(get("/accounts/1")).andExpect(status().isOk());
```

Kotlin

```
import org.springframework.test.web.servlet.get

mockMvc.get("/accounts/1").andExpect {
    status { isOk() }
}
```

You can define multiple expectations by appending `andExpectAll(..)` after performing a request, as the following example shows. In contrast to `andExpect(..)`, `andExpectAll(..)` guarantees that all supplied expectations will be asserted and that all failures will be tracked and reported.

Java

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

mockMvc.perform(get("/accounts/1")).andExpectAll(
    status().isOk(),
    content().contentType("application/json;charset=UTF-8"));
```

`MockMvcResultMatchers.*` provides a number of expectations, some of which are further nested with more detailed expectations.

Expectations fall in two general categories. The first category of assertions verifies properties of the response (for example, the response status, headers, and content). These are the most important results to assert.

The second category of assertions goes beyond the response. These assertions let you inspect Spring MVC specific aspects, such as which controller method processed the request, whether an exception was raised and handled, what the content of the model is, what view was selected, what flash attributes were added, and so on. They also let you inspect Servlet specific aspects, such as request and session attributes.

The following test asserts that binding or validation failed:

Java

```
mockMvc.perform(post("/persons"))
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

Kotlin

```
import org.springframework.test.web.servlet.post

mockMvc.post("/persons").andExpect {
    status { isOk() }
    model {
        attributeHasErrors("person")
    }
}
```

Many times, when writing tests, it is useful to dump the results of the performed request. You can do so as follows, where `print()` is a static import from `MockMvcResultHandlers`:

Java

```
mockMvc.perform(post("/persons"))
    .andDo(print())
    .andExpect(status().isOk())
    .andExpect(model().attributeHasErrors("person"));
```

Kotlin

```
import org.springframework.test.web.servlet.post

mockMvc.post("/persons").andDo {
        print()
    }.andExpect {
        status { isOk() }
        model {
            attributeHasErrors("person")
        }
    }
```

As long as request processing does not cause an unhandled exception, the `print()` method prints all the available result data to `System.out`. There is also a `log()` method and two additional variants of the `print()` method, one that accepts an `OutputStream` and one that accepts a `Writer`. For example, invoking `print(System.err)` prints the result data to `System.err`, while invoking `print(myWriter)` prints the result data to a custom writer. If you want to have the result data logged instead of printed, you can invoke the `log()` method, which logs the result data as a single `DEBUG` message under the `org.springframework.test.web.servlet.result` logging category.

In some cases, you may want to get direct access to the result and verify something that cannot be verified otherwise. This can be achieved by appending `.andReturn()` after all other expectations, as the following example shows:

Java

```
MvcResult mvcResult =
mockMvc.perform(post("/persons")).andExpect(status().isOk()).andReturn();
// ...
```

Kotlin

```
var mvcResult = mockMvc.post("/persons").andExpect { status { isOk() } }.andReturn()
// ...
```

If all tests repeat the same expectations, you can set up common expectations once when building the `MockMvc` instance, as the following example shows:

Java

```
standaloneSetup(new SimpleController())
    .alwaysExpect(status().isOk())
    .alwaysExpect(content().contentType("application/json;charset=UTF-8"))
    .build()
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

Note that common expectations are always applied and cannot be overridden without creating a separate `MockMvc` instance.

When a JSON response content contains hypermedia links created with Spring HATEOAS, you can verify the resulting links by using JsonPath expressions, as the following example shows:

Java

```
mockMvc.perform(get("/people").accept(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath("$.links[?(@.rel ==
'self')].href").value("http://localhost:8080/people"));
```

Kotlin

```
mockMvc.get("/people") {
    accept(MediaType.APPLICATION_JSON)
}.andExpect {
    jsonPath("$.links[?(@.rel == 'self')].href") {
        value("http://localhost:8080/people")
    }
}
```

When XML response content contains hypermedia links created with Spring HATEOAS, you can verify the resulting links by using XPath expressions:

Java

```
Map<String, String> ns = Collections.singletonMap("ns",
"http://www.w3.org/2005/Atom");
mockMvc.perform(get("/handle").accept(MediaType.APPLICATION_XML))
    .andExpect(xpath("/person/ns:link[@rel='self']/@href",
ns).string("http://localhost:8080/people"));
```

Kotlin

```
val ns = mapOf("ns" to "http://www.w3.org/2005/Atom")
mockMvc.get("/handle") {
    accept(MediaType.APPLICATION_XML)
}.andExpect {
    xpath("/person/ns:link[@rel='self']/@href", ns) {
        string("http://localhost:8080/people")
    }
}
```

3.7.7. Async Requests

This section shows how to use MockMvc on its own to test asynchronous request handling. If using MockMvc through the WebTestClient, there is nothing special to do to make asynchronous requests work as the `WebTestClient` automatically does what is described in this section.

Servlet asynchronous requests, supported in Spring MVC, work by exiting the Servlet container thread and allowing the application to compute the response asynchronously, after which an async dispatch is made to complete processing on a Servlet container thread.

In Spring MVC Test, async requests can be tested by asserting the produced async value first, then manually performing the async dispatch, and finally verifying the response. Below is an example test for controller methods that return `DeferredResult`, `Callable`, or reactive type such as Reactor `Mono`:

Java

```
// static import of MockMvcRequestBuilders.* and MockMvcResultMatchers.*

@Test
void test() throws Exception {
    MvcResult mvcResult = this.mockMvc.perform(get("/path"))
            .andExpect(status().isOk()) ①
            .andExpect(request().asyncStarted()) ②
            .andExpect(request().asyncResult("body")) ③
            .andReturn();

    this.mockMvc.perform(asyncDispatch(mvcResult)) ④
            .andExpect(status().isOk()) ⑤
            .andExpect(content().string("body"));
}
```

① Check response status is still unchanged

② Async processing must have started

③ Wait and assert the async result

④ Manually perform an ASYNC dispatch (as there is no running container)

⑤ Verify the final response

Kotlin

```
@Test
fun test() {
    var mvcResult = mockMvc.get("/path").andExpect {
        status { isOk() } ①
        request { asyncStarted() } ②
        // TODO Remove unused generic parameter
        request { asyncResult<Nothing>("body") } ③
    }.andReturn()


    mockMvc.perform(asyncDispatch(mvcResult)) ④
            .andExpect {
                status { isOk() } ⑤
                content().string("body")
            }
}
```

① Check response status is still unchanged

② Async processing must have started

③ Wait and assert the async result

④ Manually perform an ASYNC dispatch (as there is no running container)

⑤ Verify the final response

3.7.8. Streaming Responses

The best way to test streaming responses such as Server-Sent Events is through the WebTestClient which can be used as a test client to connect to a `MockMvc` instance to perform tests on Spring MVC controllers without a running server. For example:

Java

```
WebTestClient client = MockMvcWebTestClient.bindToController(new
SseController()).build();

FluxExchangeResult<Person> exchangeResult = client.get()
        .uri("/persons")
        .exchange()
        .expectStatus().isOk()
        .expectHeader().contentType("text/event-stream")
        .returnResult(Person.class);

// Use StepVerifier from Project Reactor to test the streaming response

StepVerifier.create(exchangeResult.getResponseBody())
        .expectNext(new Person("N0"), new Person("N1"), new Person("N2"))
        .expectNextCount(4)
        .consumeNextWith(person -> assertThat(person.getName()).endsWith("7"))
        .thenCancel()
        .verify();
```

`WebTestClient` can also connect to a live server and perform full end-to-end integration tests. This is also supported in Spring Boot where you can test a running server.

3.7.9. Filter Registrations

When setting up a `MockMvc` instance, you can register one or more Servlet `Filter` instances, as the following example shows:

Java

```
mockMvc = standaloneSetup(new PersonController()).addFilters(new
CharacterEncodingFilter()).build();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

Registered filters are invoked through the `MockFilterChain` from `spring-test`, and the last filter delegates to the `DispatcherServlet`.

3.7.10. MockMvc vs End-to-End Tests

MockMVc is built on Servlet API mock implementations from the `spring-test` module and does not rely on a running container. Therefore, there are some differences when compared to full end-to-end integration tests with an actual client and a live server running.

The easiest way to think about this is by starting with a blank `MockHttpServletRequest`. Whatever you add to it is what the request becomes. Things that may catch you by surprise are that there is

no context path by default; no `jsessionid` cookie; no forwarding, error, or async dispatches; and, therefore, no actual JSP rendering. Instead, "forwarded" and "redirected" URLs are saved in the `MockHttpServletResponse` and can be asserted with expectations.

This means that, if you use JSPs, you can verify the JSP page to which the request was forwarded, but no HTML is rendered. In other words, the JSP is not invoked. Note, however, that all other rendering technologies that do not rely on forwarding, such as Thymeleaf and Freemarker, render HTML to the response body as expected. The same is true for rendering JSON, XML, and other formats through `@ResponseBody` methods.

Alternatively, you may consider the full end-to-end integration testing support from Spring Boot with `@SpringBootTest`. See the Spring Boot Reference Guide.

There are pros and cons for each approach. The options provided in Spring MVC Test are different stops on the scale from classic unit testing to full integration testing. To be certain, none of the options in Spring MVC Test fall under the category of classic unit testing, but they are a little closer to it. For example, you can isolate the web layer by injecting mocked services into controllers, in which case you are testing the web layer only through the `DispatcherServlet` but with actual Spring configuration, as you might test the data access layer in isolation from the layers above it. Also, you can use the stand-alone setup, focusing on one controller at a time and manually providing the configuration required to make it work.

Another important distinction when using Spring MVC Test is that, conceptually, such tests are the server-side, so you can check what handler was used, if an exception was handled with a HandlerExceptionResolver, what the content of the model is, what binding errors there were, and other details. That means that it is easier to write expectations, since the server is not an opaque box, as it is when testing it through an actual HTTP client. This is generally an advantage of classic unit testing: It is easier to write, reason about, and debug but does not replace the need for full integration tests. At the same time, it is important not to lose sight of the fact that the response is the most important thing to check. In short, there is room here for multiple styles and strategies of testing even within the same project.

3.7.11. Further Examples

The framework's own tests include many sample tests intended to show how to use MockMvc on its own or through the WebTestClient. Browse these examples for further ideas.

3.7.12. HtmlUnit Integration

Spring provides integration between MockMvc and HtmlUnit. This simplifies performing end-to-end testing when using HTML-based views. This integration lets you:

- Easily test HTML pages by using tools such as HtmlUnit, WebDriver, and Geb without the need to deploy to a Servlet container.
- Test JavaScript within pages.
- Optionally, test using mock services to speed up testing.
- Share logic between in-container end-to-end tests and out-of-container integration tests.

MockMvc works with templating technologies that do not rely on a Servlet Container (for example, Thymeleaf, FreeMarker, and others), but it does not work with JSPs, since they rely on the Servlet container.

Why HtmlUnit Integration?

The most obvious question that comes to mind is "Why do I need this?" The answer is best found by exploring a very basic sample application. Assume you have a Spring MVC web application that supports CRUD operations on a `Message` object. The application also supports paging through all messages. How would you go about testing it?

With Spring MVC Test, we can easily test if we are able to create a `Message`, as follows:

Java

```
MockHttpServletRequestBuilder createMessage = post("/messages/")
        .param("summary", "Spring Rocks")
        .param("text", "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl("/messages/123"));
```

Kotlin

```
@Test
fun test() {
    mockMvc.post("/messages/") {
        param("summary", "Spring Rocks")
        param("text", "In case you didn't know, Spring Rocks!")
    }.andExpect {
        status().is3xxRedirection()
        redirectedUrl("/messages/123")
    }
}
```

What if we want to test the form view that lets us create the message? For example, assume our form looks like the following snippet:

```
<form id="messageForm" action="/messages/" method="post">
    <div class="pull-right"><a href="/messages/">Messages</a></div>

    <label for="summary">Summary</label>
    <input type="text" class="required" id="summary" name="summary" value="" />

    <label for="text">Message</label>
    <textarea id="text" name="text"></textarea>

    <div class="form-actions">
        <input type="submit" value="Create" />
    </div>
</form>
```

How do we ensure that our form produce the correct request to create a new message? A naive attempt might resemble the following:

Java

```
mockMvc.perform(get("/messages/form"))
        .andExpect(xpath("//input[@name='summary']").exists())
        .andExpect(xpath("//textarea[@name='text']").exists());
```

Kotlin

```
mockMvc.get("/messages/form").andExpect {
    xpath("//input[@name='summary']") { exists() }
    xpath("//textarea[@name='text']") { exists() }
}
```

This test has some obvious drawbacks. If we update our controller to use the parameter `message` instead of `text`, our form test continues to pass, even though the HTML form is out of synch with the controller. To resolve this we can combine our two tests, as follows:

Java

```
String summaryParamName = "summary";
String textParamName = "text";
mockMvc.perform(get("/messages/form"))
        .andExpect(xpath("//input[@name='" + summaryParamName + "']").exists())
        .andExpect(xpath("//textarea[@name='" + textParamName + "']").exists());

MockHttpServletRequestBuilder createMessage = post("/messages/")
        .param(summaryParamName, "Spring Rocks")
        .param(textParamName, "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)
        .andExpect(status().is3xxRedirection())
        .andExpect(redirectedUrl("/messages/123"));
```

Kotlin

```
val summaryParamName = "summary";
val textParamName = "text";
mockMvc.get("/messages/form").andExpect {
    xpath("//input[@name='$summaryParamName']") { exists() }
    xpath("//textarea[@name='$textParamName']") { exists() }
}
mockMvc.post("/messages/") {
    param(summaryParamName, "Spring Rocks")
    param(textParamName, "In case you didn't know, Spring Rocks!")
}.andExpect {
    status().is3xxRedirection()
    redirectedUrl("/messages/123")
}
```

This would reduce the risk of our test incorrectly passing, but there are still some problems:

- What if we have multiple forms on our page? Admittedly, we could update our XPath expressions, but they get more complicated as we take more factors into account: Are the fields the correct type? Are the fields enabled? And so on.
- Another issue is that we are doing double the work we would expect. We must first verify the view, and then we submit the view with the same parameters we just verified. Ideally, this could be done all at once.
- Finally, we still cannot account for some things. For example, what if the form has JavaScript validation that we wish to test as well?

The overall problem is that testing a web page does not involve a single interaction. Instead, it is a combination of how the user interacts with a web page and how that web page interacts with other resources. For example, the result of a form view is used as the input to a user for creating a message. In addition, our form view can potentially use additional resources that impact the behavior of the page, such as JavaScript validation.

Integration Testing to the Rescue?

To resolve the issues mentioned earlier, we could perform end-to-end integration testing, but this has some drawbacks. Consider testing the view that lets us page through the messages. We might need the following tests:

- Does our page display a notification to the user to indicate that no results are available when the messages are empty?
- Does our page properly display a single message?
- Does our page properly support paging?

To set up these tests, we need to ensure our database contains the proper messages. This leads to a number of additional challenges:

- Ensuring the proper messages are in the database can be tedious. (Consider foreign key constraints.)
- Testing can become slow, since each test would need to ensure that the database is in the correct state.
- Since our database needs to be in a specific state, we cannot run tests in parallel.
- Performing assertions on such items as auto-generated ids, timestamps, and others can be difficult.

These challenges do not mean that we should abandon end-to-end integration testing altogether. Instead, we can reduce the number of end-to-end integration tests by refactoring our detailed tests to use mock services that run much faster, more reliably, and without side effects. We can then implement a small number of true end-to-end integration tests that validate simple workflows to ensure that everything works together properly.

Enter HtmlUnit Integration

So how can we achieve a balance between testing the interactions of our pages and still retain good performance within our test suite? The answer is: "By integrating MockMvc with HtmlUnit."

HtmlUnit Integration Options

You have a number of options when you want to integrate MockMvc with HtmlUnit:

- MockMvc and HtmlUnit: Use this option if you want to use the raw HtmlUnit libraries.
- MockMvc and WebDriver: Use this option to ease development and reuse code between integration and end-to-end testing.
- MockMvc and Geb: Use this option if you want to use Groovy for testing, ease development, and reuse code between integration and end-to-end testing.

MockMvc and HtmlUnit

This section describes how to integrate MockMvc and HtmlUnit. Use this option if you want to use the raw HtmlUnit libraries.

MockMvc and HtmlUnit Setup

First, make sure that you have included a test dependency on `net.sourceforge.htmlunit:htmlunit`. In order to use HtmlUnit with Apache HttpComponents 4.5+, you need to use HtmlUnit 2.18 or higher.

We can easily create an HtmlUnit `WebClient` that integrates with MockMvc by using the `MockMvcWebClientBuilder`, as follows:

Java

```
WebClient webClient;

@BeforeEach
void setup(WebApplicationContext context) {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build();
}
```

Kotlin

```
lateinit var webClient: WebClient

@BeforeEach
fun setup(context: WebApplicationContext) {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build()
}
```

This is a simple example of using `MockMvcWebClientBuilder`. For advanced usage, see Advanced `MockMvcWebClientBuilder`.

This ensures that any URL that references `localhost` as the server is directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL is requested by using a network connection, as normal. This lets us easily test the use of CDNs.

MockMvc and HtmlUnit Usage

Now we can use HtmlUnit as we normally would but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following:

Java

```
HtmlPage createMsgFormPage = webClient.getPage("http://localhost/messages/form");
```

Kotlin

```
val createMsgFormPage = webClient.getPage("http://localhost/messages/form")
```

The default context path is "". Alternatively, we can specify the context path, as described in Advanced `MockMvcWebClientBuilder`.

Once we have a reference to the `HtmlPage`, we can then fill out the form and submit it to create a message, as the following example shows:

Java

```
HtmlForm form = createMsgFormPage.getHtmlElementById("messageForm");
HtmlTextInput summaryInput = createMsgFormPage.getHtmlElementById("summary");
summaryInput.setValueAttribute("Spring Rocks");
HtmlTextArea textInput = createMsgFormPage.getHtmlElementById("text");
textInput.setText("In case you didn't know, Spring Rocks!");
HtmlSubmitInput submit = form.getOneHtmlElementByAttribute("input", "type", "submit");
HtmlPage newMessagePage = submit.click();
```

Kotlin

```
val form = createMsgFormPage.getHtmlElementById("messageForm")
val summaryInput = createMsgFormPage.getHtmlElementById("summary")
summaryInput.setValueAttribute("Spring Rocks")
val textInput = createMsgFormPage.getHtmlElementById("text")
textInput.setText("In case you didn't know, Spring Rocks!")
val submit = form.getOneHtmlElementByAttribute("input", "type", "submit")
val newMessagePage = submit.click()
```

Finally, we can verify that a new message was created successfully. The following assertions use the AssertJ library:

Java

```
assertThat(newMessagePage.getUrl().toString()).endsWith("/messages/123");
String id = newMessagePage.getHtmlElementById("id").getTextContent();
assertThat(id).isEqualTo("123");
String summary = newMessagePage.getHtmlElementById("summary").getTextContent();
assertThat(summary).isEqualTo("Spring Rocks");
String text = newMessagePage.getHtmlElementById("text").getTextContent();
assertThat(text).isEqualTo("In case you didn't know, Spring Rocks!");
```

Kotlin

```
assertThat(newMessagePage.getUrl().toString()).endsWith("/messages/123")
val id = newMessagePage.getHtmlElementById("id").getTextContent()
assertThat(id).isEqualTo("123")
val summary = newMessagePage.getHtmlElementById("summary").getTextContent()
assertThat(summary).isEqualTo("Spring Rocks")
val text = newMessagePage.getHtmlElementById("text").getTextContent()
assertThat(text).isEqualTo("In case you didn't know, Spring Rocks!")
```

The preceding code improves on our MockMvc test in a number of ways. First, we no longer have to explicitly verify our form and then create a request that looks like the form. Instead, we request the form, fill it out, and submit it, thereby significantly reducing the overhead.

Another important factor is that HtmlUnit uses the Mozilla Rhino engine to evaluate JavaScript. This means that we can also test the behavior of JavaScript within our pages.

See the HtmlUnit documentation for additional information about using HtmlUnit.

Advanced `MockMvcWebClientBuilder`

In the examples so far, we have used `MockMvcWebClientBuilder` in the simplest way possible, by building a `WebClient` based on the `WebApplicationContext` loaded for us by the Spring TestContext Framework. This approach is repeated in the following example:

Java

```
WebClient webClient;

@BeforeEach
void setup(WebApplicationContext context) {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build();
}
```

Kotlin

```
lateinit var webClient: WebClient

@BeforeEach
fun setup(context: WebApplicationContext) {
    webClient = MockMvcWebClientBuilder
            .webAppContextSetup(context)
            .build()
}
```

We can also specify additional configuration options, as the following example shows:

Java

```
WebClient webClient;

@BeforeEach
void setup() {
    webClient = MockMvcWebClientBuilder
        // demonstrates applying a MockMvcConfigurer (Spring Security)
        .webAppContextSetup(context, springSecurity())
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build();
}
```

Kotlin

```
lateinit var webClient: WebClient

@BeforeEach
fun setup() {
    webClient = MockMvcWebClientBuilder
        // demonstrates applying a MockMvcConfigurer (Spring Security)
        .webAppContextSetup(context, springSecurity())
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build()
}
```

As an alternative, we can perform the exact same setup by configuring the `MockMvc` instance separately and supplying it to the `MockMvcWebClientBuilder`, as follows:

Java

```
MockMvc mockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

webClient = MockMvcWebClientBuilder
        .mockMvcSetup(mockMvc)
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

This is more verbose, but, by building the `WebClient` with a `MockMvc` instance, we have the full power of MockMvc at our fingertips.

For additional information on creating a `MockMvc` instance, see Setup Choices.

MockMvc and WebDriver

In the previous sections, we have seen how to use MockMvc in conjunction with the raw HtmlUnit APIs. In this section, we use additional abstractions within the Selenium WebDriver to make things even easier.

Why WebDriver and MockMvc?

We can already use HtmlUnit and MockMvc, so why would we want to use WebDriver? The Selenium WebDriver provides a very elegant API that lets us easily organize our code. To better show how it works, we explore an example in this section.

Despite being a part of Selenium, WebDriver does not require a Selenium Server to run your tests.

Suppose we need to ensure that a message is created properly. The tests involve finding the HTML form input elements, filling them out, and making various assertions.

This approach results in numerous separate tests because we want to test error conditions as well. For example, we want to ensure that we get an error if we fill out only part of the form. If we fill out the entire form, the newly created message should be displayed afterwards.

If one of the fields were named “summary”, we might have something that resembles the following repeated in multiple places within our tests:

Java

```
HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
summaryInput.setValueAttribute(summary);
```

Kotlin

```
val summaryInput = currentPage.getHtmlElementById("summary")
summaryInput.setValueAttribute(summary)
```

So what happens if we change the `id` to `smmry`? Doing so would force us to update all of our tests to incorporate this change. This violates the DRY principle, so we should ideally extract this code into its own method, as follows:

Java

```
public HtmlPage createMessage(HtmlPage currentPage, String summary, String text) {
    setSummary(currentPage, summary);
    // ...
}

public void setSummary(HtmlPage currentPage, String summary) {
    HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");
    summaryInput.setValueAttribute(summary);
}
```

Kotlin

```
fun createMessage(currentPage: HtmlPage, summary:String, text:String) :HtmlPage{
    setSummary(currentPage, summary);
    // ...
}

fun setSummary(currentPage:HtmlPage , summary: String) {
    val summaryInput = currentPage.getHtmlElementById("summary")
    summaryInput.setValueAttribute(summary)
}
```

Doing so ensures that we do not have to update all of our tests if we change the UI.

We might even take this a step further and place this logic within an `Object` that represents the `HtmlPage` we are currently on, as the following example shows:

Java

```
public class CreateMessagePage {

    final HtmlPage currentPage;

    final HtmlTextInput summaryInput;

    final HtmlSubmitInput submit;

    public CreateMessagePage(HtmlPage currentPage) {
        this.currentPage = currentPage;
        this.summaryInput = currentPage.getHtmlElementById("summary");
        this.submit = currentPage.getHtmlElementById("submit");
    }

    public <T> T createMessage(String summary, String text) throws Exception {
        setSummary(summary);

        HtmlPage result = submit.click();
        boolean error = CreateMessagePage.at(result);

        return (T) (error ? new CreateMessagePage(result) : new
ViewMessagePage(result));
    }

    public void setSummary(String summary) throws Exception {
        summaryInput.setValueAttribute(summary);
    }

    public static boolean at(HtmlPage page) {
        return "Create Message".equals(page.getTitleText());
    }
}
```

Kotlin

```
    class CreateMessagePage(private val currentPage: HtmlPage) {

        val summaryInput: HtmlTextInput = currentPage.getHtmlElementById("summary")

        val submit: HtmlSubmitInput = currentPage.getHtmlElementById("submit")

        fun <T> createMessage(summary: String, text: String): T {
            setSummary(summary)

            val result = submit.click()
            val error = at(result)

            return (if (error) CreateMessagePage(result) else ViewMessagePage(result))
as T
        }

        fun setSummary(summary: String) {
            summaryInput.setValueAttribute(summary)
        }

        fun at(page: HtmlPage): Boolean {
            return "Create Message" == page.getTitleText()
        }
    }
}
```

Formerly, this pattern was known as the Page Object Pattern. While we can certainly do this with HtmlUnit, WebDriver provides some tools that we explore in the following sections to make this pattern much easier to implement.

MockMvc and WebDriver Setup

To use Selenium WebDriver with the Spring MVC Test framework, make sure that your project includes a test dependency on `org.seleniumhq.selenium:selenium-htmlunit-driver`.

We can easily create a Selenium WebDriver that integrates with MockMvc by using the `MockMvcHtmlUnitDriverBuilder` as the following example shows:

Java

```
WebDriver driver;

@BeforeEach
void setup(WebApplicationContext context) {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build();
}
```

Kotlin

```
lateinit var driver: WebDriver

@BeforeEach
fun setup(context: WebApplicationContext) {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build()
}
```

This is a simple example of using `MockMvcHtmlUnitDriverBuilder`. For more advanced usage, see Advanced `MockMvcHtmlUnitDriverBuilder`.

The preceding example ensures that any URL that references `localhost` as the server is directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL is requested by using a network connection, as normal. This lets us easily test the use of CDNs.

MockMvc and WebDriver Usage

Now we can use WebDriver as we normally would but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following:

Java

```
CreateMessagePage page = CreateMessagePage.to(driver);
```

Kotlin

```
val page = CreateMessagePage.to(driver)
```

We can then fill out the form and submit it to create a message, as follows:

Java

```
ViewMessagePage viewMessagePage =
        page.createMessage(ViewMessagePage.class, expectedSummary, expectedText);
```

Kotlin

```
val viewMessagePage =
    page.createMessage(ViewMessagePage::class, expectedSummary, expectedText)
```

This improves on the design of our HtmlUnit test by leveraging the Page Object Pattern. As we mentioned in Why WebDriver and MockMvc?, we can use the Page Object Pattern with HtmlUnit, but it is much easier with WebDriver. Consider the following `CreateMessagePage` implementation:

Java

```
public class CreateMessagePage
        extends AbstractPage { ①

    ②
    private WebElement summary;
    private WebElement text;

    ③
    @FindBy(css = "input[type=submit]")
    private WebElement submit;

    public CreateMessagePage(WebDriver driver) {
        super(driver);
    }

    public <T> T createMessage(Class<T> resultPage, String summary, String details) {
        this.summary.sendKeys(summary);
        this.text.sendKeys(details);
        this.submit.click();
        return PageFactory.initElements(driver, resultPage);
    }

    public static CreateMessagePage to(WebDriver driver) {
        driver.get("http://localhost:9990/mail/messages/form");
        return PageFactory.initElements(driver, CreateMessagePage.class);
    }
}
```

① `CreateMessagePage` extends the `AbstractPage`. We do not go over the details of `AbstractPage`, but, in summary, it contains common functionality for all of our pages. For example, if our application has a navigational bar, global error messages, and other features, we can place this logic in a shared location.

② We have a member variable for each of the parts of the HTML page in which we are interested. These are of type `WebElement`. WebDriver's `PageFactory` lets us remove a lot of code from the HtmlUnit version of `CreateMessagePage` by automatically resolving each `WebElement`. The `PageFactory#initElements(WebDriver,Class<T>)` method automatically resolves each `WebElement` by using the field name and looking it up by the `id` or `name` of the element within the HTML page.

③ We can use the `@FindBy` annotation to override the default lookup behavior. Our example shows how to use the `@FindBy` annotation to look up our submit button with a `css` selector (**input[type=submit]**).

Kotlin

```
class CreateMessagePage(private val driver: WebDriver) : AbstractPage(driver) { ①

    ②
    private lateinit var summary: WebElement
    private lateinit var text: WebElement

    ③
    @FindBy(css = "input[type=submit]")
    private lateinit var submit: WebElement

    fun <T> createMessage(resultPage: Class<T>, summary: String, details: String): T {
        this.summary.sendKeys(summary)
        text.sendKeys(details)
        submit.click()
        return PageFactory.initElements(driver, resultPage)
    }
    companion object {
        fun to(driver: WebDriver): CreateMessagePage {
            driver.get("http://localhost:9990/mail/messages/form")
            return PageFactory.initElements(driver, CreateMessagePage::class.java)
        }
    }
}
```

① `CreateMessagePage` extends the `AbstractPage`. We do not go over the details of `AbstractPage`, but, in summary, it contains common functionality for all of our pages. For example, if our application has a navigational bar, global error messages, and other features, we can place this logic in a shared location.

② We have a member variable for each of the parts of the HTML page in which we are interested. These are of type `WebElement`. WebDriver's `PageFactory` lets us remove a lot of code from the HtmlUnit version of `CreateMessagePage` by automatically resolving each `WebElement`. The `PageFactory#initElements(WebDriver,Class<T>)` method automatically resolves each `WebElement` by using the field name and looking it up by the `id` or `name` of the element within the HTML page.

③ We can use the `@FindBy` annotation to override the default lookup behavior. Our example shows how to use the `@FindBy` annotation to look up our submit button with a `css` selector (**input[type=submit]**).

Finally, we can verify that a new message was created successfully. The following assertions use the AssertJ assertion library:

Java

```
assertThat(viewMessagePage.getMessage()).isEqualTo(expectedMessage);
assertThat(viewMessagePage.getSuccess()).isEqualTo("Successfully created a new
message");
```

Kotlin

```
assertThat(viewMessagePage.message).isEqualTo(expectedMessage)
assertThat(viewMessagePage.success).isEqualTo("Successfully created a new message")
```

We can see that our `ViewMessagePage` lets us interact with our custom domain model. For example, it exposes a method that returns a `Message` object:

Java

```
public Message getMessage() throws ParseException {
    Message message = new Message();
    message.setId(getId());
    message.setCreated(getCreated());
    message.setSummary(getSummary());
    message.setText(getText());
    return message;
}
```

Kotlin

```
fun getMessage() = Message(getId(), getCreated(), getSummary(), getText())
```

We can then use the rich domain objects in our assertions.

Lastly, we must not forget to close the `WebDriver` instance when the test is complete, as follows:

Java

```
@AfterEach
void destroy() {
    if (driver != null) {
        driver.close();
    }
}
```

Kotlin

```
@AfterEach
fun destroy() {
    if (driver != null) {
        driver.close()
    }
}
```

For additional information on using WebDriver, see the Selenium WebDriver documentation.

Advanced `MockMvcHtmlUnitDriverBuilder`

In the examples so far, we have used `MockMvcHtmlUnitDriverBuilder` in the simplest way possible, by building a `WebDriver` based on the `WebApplicationContext` loaded for us by the Spring TestContext Framework. This approach is repeated here, as follows:

Java

```
WebDriver driver;

@BeforeEach
void setup(WebApplicationContext context) {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build();
}
```

Kotlin

```
lateinit var driver: WebDriver

@BeforeEach
fun setup(context: WebApplicationContext) {
    driver = MockMvcHtmlUnitDriverBuilder
            .webAppContextSetup(context)
            .build()
}
```

We can also specify additional configuration options, as follows:

Java

```
WebDriver driver;

@BeforeEach
void setup() {
    driver = MockMvcHtmlUnitDriverBuilder
            // demonstrates applying a MockMvcConfigurer (Spring Security)
            .webAppContextSetup(context, springSecurity())
            // for illustration only - defaults to ""
            .contextPath("")
            // By default MockMvc is used for localhost only;
            // the following will use MockMvc for example.com and example.org as well
            .useMockMvcForHosts("example.com","example.org")
            .build();
}
```

Kotlin

```
lateinit var driver: WebDriver

@BeforeEach
fun setup() {
    driver = MockMvcHtmlUnitDriverBuilder
            // demonstrates applying a MockMvcConfigurer (Spring Security)
            .webAppContextSetup(context, springSecurity())
            // for illustration only - defaults to ""
            .contextPath("")
            // By default MockMvc is used for localhost only;
            // the following will use MockMvc for example.com and example.org as well
            .useMockMvcForHosts("example.com","example.org")
            .build()
}
```

As an alternative, we can perform the exact same setup by configuring the `MockMvc` instance separately and supplying it to the `MockMvcHtmlUnitDriverBuilder`, as follows:

Java

```
MockMvc mockMvc = MockMvcBuilders
        .webAppContextSetup(context)
        .apply(springSecurity())
        .build();

driver = MockMvcHtmlUnitDriverBuilder
        .mockMvcSetup(mockMvc)
        // for illustration only - defaults to ""
        .contextPath("")
        // By default MockMvc is used for localhost only;
        // the following will use MockMvc for example.com and example.org as well
        .useMockMvcForHosts("example.com","example.org")
        .build();
```

Kotlin

```
// Not possible in Kotlin until https://youtrack.jetbrains.com/issue/KT-22208 is fixed
```

This is more verbose, but, by building the `WebDriver` with a `MockMvc` instance, we have the full power of MockMvc at our fingertips.

For additional information on creating a `MockMvc` instance, see Setup Choices.

MockMvc and Geb

In the previous section, we saw how to use MockMvc with WebDriver. In this section, we use Geb to make our tests even Groovy-er.

Why Geb and MockMvc?

Geb is backed by WebDriver, so it offers many of the same benefits that we get from WebDriver. However, Geb makes things even easier by taking care of some of the boilerplate code for us.

MockMvc and Geb Setup

We can easily initialize a Geb `Browser` with a Selenium WebDriver that uses MockMvc, as follows:

```
def setup() {
    browser.driver = MockMvcHtmlUnitDriverBuilder
        .webAppContextSetup(context)
        .build()
}
```

This is a simple example of using `MockMvcHtmlUnitDriverBuilder`. For more advanced usage, see Advanced `MockMvcHtmlUnitDriverBuilder`.

This ensures that any URL referencing `localhost` as the server is directed to our `MockMvc` instance without the need for a real HTTP connection. Any other URL is requested by using a network connection as normal. This lets us easily test the use of CDNs.

MockMvc and Geb Usage

Now we can use Geb as we normally would but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following:

```
to CreateMessagePage
```

We can then fill out the form and submit it to create a message, as follows:

```
when:
form.summary = expectedSummary
form.text = expectedMessage
submit.click(ViewMessagePage)
```

Any unrecognized method calls or property accesses or references that are not found are forwarded to the current page object. This removes a lot of the boilerplate code we needed when using WebDriver directly.

As with direct WebDriver usage, this improves on the design of our HtmlUnit test by using the Page Object Pattern. As mentioned previously, we can use the Page Object Pattern with HtmlUnit and WebDriver, but it is even easier with Geb. Consider our new Groovy-based `CreateMessagePage` implementation:

```
class CreateMessagePage extends Page {
    static url = 'messages/form'
    static at = { assert title == 'Messages : Create'; true }
    static content =  {
        submit { $('input[type=submit]') }
        form { $('form') }
        errors(required:false) { $('label.error, .alert-error')?.text() }
    }
}
```

Our `CreateMessagePage` extends `Page`. We do not go over the details of `Page`, but, in summary, it contains common functionality for all of our pages. We define a URL in which this page can be found. This lets us navigate to the page, as follows:

```
to CreateMessagePage
```

We also have an `at` closure that determines if we are at the specified page. It should return `true` if we are on the correct page. This is why we can assert that we are on the correct page, as follows:

```
then:
at CreateMessagePage
errors.contains('This field is required.')
```

We use an assertion in the closure so that we can determine where things went wrong if we were at the wrong page.

Next, we create a `content` closure that specifies all the areas of interest within the page. We can use a jQuery-ish Navigator API to select the content in which we are interested.

Finally, we can verify that a new message was created successfully, as follows:

```
then:
at ViewMessagePage
success == 'Successfully created a new message'
id
date
summary == expectedSummary
message == expectedMessage
```

For further details on how to get the most out of Geb, see The Book of Geb user's manual.

3.8. Testing Client Applications

You can use client-side tests to test code that internally uses the `RestTemplate`. The idea is to declare expected requests and to provide "stub" responses so that you can focus on testing the code in

isolation (that is, without running a server). The following example shows how to do so:

Java

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(requestTo("/greeting")).andRespond(withSuccess());

// Test code that uses the above RestTemplate ...

mockServer.verify();
```

Kotlin

```
val restTemplate = RestTemplate()

val mockServer = MockRestServiceServer.bindTo(restTemplate).build()
mockServer.expect(requestTo("/greeting")).andRespond(withSuccess())

// Test code that uses the above RestTemplate ...

mockServer.verify()
```

In the preceding example, `MockRestServiceServer` (the central class for client-side REST tests) configures the `RestTemplate` with a custom `ClientHttpRequestFactory` that asserts actual requests against expectations and returns "stub" responses. In this case, we expect a request to `/greeting` and want to return a 200 response with `text/plain` content. We can define additional expected requests and stub responses as needed. When we define expected requests and stub responses, the `RestTemplate` can be used in client-side code as usual. At the end of testing, `mockServer.verify()` can be used to verify that all expectations have been satisfied.

By default, requests are expected in the order in which expectations were declared. You can set the `ignoreExpectOrder` option when building the server, in which case all expectations are checked (in order) to find a match for a given request. That means requests are allowed to come in any order. The following example uses `ignoreExpectOrder`:

Java

```
server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build();
```

Kotlin

```
server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build()
```

Even with unordered requests by default, each request is allowed to run once only. The `expect` method provides an overloaded variant that accepts an `ExpectedCount` argument that specifies a count range (for example, `once`, `manyTimes`, `max`, `min`, `between`, and so on). The following example uses

`times`:

Java

```
RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();
mockServer.expect(times(2), requestTo("/something")).andRespond(withSuccess());
mockServer.expect(times(3), requestTo("/somewhere")).andRespond(withSuccess());

// ...

mockServer.verify();
```

Kotlin

```
val restTemplate = RestTemplate()

val mockServer = MockRestServiceServer.bindTo(restTemplate).build()
mockServer.expect(times(2), requestTo("/something")).andRespond(withSuccess())
mockServer.expect(times(3), requestTo("/somewhere")).andRespond(withSuccess())

// ...

mockServer.verify()
```

Note that, when `ignoreExpectOrder` is not set (the default), and, therefore, requests are expected in order of declaration, then that order applies only to the first of any expected request. For example if "/something" is expected two times followed by "/somewhere" three times, then there should be a request to "/something" before there is a request to "/somewhere", but, aside from that subsequent "/something" and "/somewhere", requests can come at any time.

As an alternative to all of the above, the client-side test support also provides a `ClientHttpRequestFactory` implementation that you can configure into a `RestTemplate` to bind it to a `MockMvc` instance. That allows processing requests using actual server-side logic but without running a server. The following example shows how to do so:

Java

```
MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();
this.restTemplate = new RestTemplate(new MockMvcClientHttpRequestFactory(mockMvc));

// Test code that uses the above RestTemplate ...
```

Kotlin

```
val mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build()
restTemplate = RestTemplate(MockMvcClientHttpRequestFactory(mockMvc))

// Test code that uses the above RestTemplate ...
```

3.8.1. Static Imports

As with server-side tests, the fluent API for client-side tests requires a few static imports. Those are easy to find by searching for `MockRest*`. Eclipse users should add `MockRestRequestMatchers.*` and `MockRestResponseCreators.*` as "favorite static members" in the Eclipse preferences under Java → Editor → Content Assist → Favorites. That allows using content assist after typing the first character of the static method name. Other IDEs (such IntelliJ) may not require any additional configuration. Check for the support for code completion on static members.

3.8.2. Further Examples of Client-side REST Tests

Spring MVC Test's own tests include example tests of client-side REST tests.

3.9. Appendix

3.9.1. Annotations

This section covers annotations that you can use when you test Spring applications. It includes the following topics:

- Standard Annotation Support
- Spring Testing Annotations
- Spring JUnit 4 Testing Annotations
- Spring JUnit Jupiter Testing Annotations
- Meta-Annotation Support for Testing

Standard Annotation Support

The following annotations are supported with standard semantics for all configurations of the Spring TestContext Framework. Note that these annotations are not specific to tests and can be used anywhere in the Spring Framework.

- `@Autowired`
- `@Qualifier`
- `@Value`
- `@Resource` (jakarta.annotation) if JSR-250 is present
- `@ManagedBean` (jakarta.annotation) if JSR-250 is present

- `@Inject` (jakarta.inject) if JSR-330 is present
- `@Named` (jakarta.inject) if JSR-330 is present
- `@PersistenceContext` (jakarta.persistence) if JPA is present
- `@PersistenceUnit` (jakarta.persistence) if JPA is present
- `@Transactional` (org.springframework.transaction.annotation) *with limited attribute support*

JSR-250 Lifecycle Annotations

In the Spring TestContext Framework, you can use `@PostConstruct` and `@PreDestroy` with standard semantics on any application components configured in the `ApplicationContext`. However, these lifecycle annotations have limited usage within an actual test class.

If a method within a test class is annotated with `@PostConstruct`, that method runs before any before methods of the underlying test framework (for example, methods annotated with JUnit Jupiter's `@BeforeEach`), and that applies for every test method in the test class. On the other hand, if a method within a test class is annotated with `@PreDestroy`, that method never runs. Therefore, within a test class, we recommend that you use test lifecycle callbacks from the underlying test framework instead of `@PostConstruct` and `@PreDestroy`.

Spring Testing Annotations

The Spring Framework provides the following set of Spring-specific annotations that you can use in your unit and integration tests in conjunction with the TestContext framework. See the corresponding javadoc for further information, including default attribute values, attribute aliases, and other details.

Spring's testing annotations include the following:

- `@BootstrapWith`
- `@ContextConfiguration`
- `@WebAppConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DynamicPropertySource`
- `@DirtiesContext`
- `@TestExecutionListeners`
- `@RecordApplicationEvents`
- `@Commit`
- `@Rollback`
- `@BeforeTransaction`

- `@AfterTransaction`
- `@Sql`
- `@SqlConfig`
- `@SqlMergeMode`
- `@SqlGroup`

`@BootstrapWith`

`@BootstrapWith` is a class-level annotation that you can use to configure how the Spring TestContext Framework is bootstrapped. Specifically, you can use `@BootstrapWith` to specify a custom `TestContextBootstrapper`. See the section on bootstrapping the TestContext framework for further details.

`@ContextConfiguration`

`@ContextConfiguration` defines class-level metadata that is used to determine how to load and configure an `ApplicationContext` for integration tests. Specifically, `@ContextConfiguration` declares the application context resource `locations` or the component `classes` used to load the context.

Resource locations are typically XML configuration files or Groovy scripts located in the classpath, while component classes are typically `@Configuration` classes. However, resource locations can also refer to files and scripts in the file system, and component classes can be `@Component` classes, `@Service` classes, and so on. See Component Classes for further details.

The following example shows a `@ContextConfiguration` annotation that refers to an XML file:

Java

```
@ContextConfiguration("/test-config.xml") ①
class XmlApplicationContextTests {
    // class body...
}
```

① Referring to an XML file.

Kotlin

```
@ContextConfiguration("/test-config.xml") ①
class XmlApplicationContextTests {
    // class body...
}
```

① Referring to an XML file.

The following example shows a `@ContextConfiguration` annotation that refers to a class:

Java

```
@ContextConfiguration(classes = TestConfig.class) ①
class ConfigClassApplicationContextTests {
    // class body...
}
```

① Referring to a class.

Kotlin

```
@ContextConfiguration(classes = [TestConfig::class]) ①
class ConfigClassApplicationContextTests {
    // class body...
}
```

① Referring to a class.

As an alternative or in addition to declaring resource locations or component classes, you can use `@ContextConfiguration` to declare `ApplicationContextInitializer` classes. The following example shows such a case:

Java

```
@ContextConfiguration(initializers = CustomContextInitializer.class) ①
class ContextInitializerTests {
    // class body...
}
```

① Declaring an initializer class.

Kotlin

```
@ContextConfiguration(initializers = [CustomContextInitializer::class]) ①
class ContextInitializerTests {
    // class body...
}
```

① Declaring an initializer class.

You can optionally use `@ContextConfiguration` to declare the `ContextLoader` strategy as well. Note, however, that you typically do not need to explicitly configure the loader, since the default loader supports `initializers` and either resource `locations` or component `classes`.

The following example uses both a location and a loader:

Java

```
@ContextConfiguration(locations = "/test-context.xml", loader =
CustomContextLoader.class) ①
class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

① Configuring both a location and a custom loader.

Kotlin

```
@ContextConfiguration("/test-context.xml", loader = CustomContextLoader::class) ①
class CustomLoaderXmlApplicationContextTests {
    // class body...
}
```

① Configuring both a location and a custom loader.

`@ContextConfiguration` provides support for inheriting resource locations or configuration classes as well as context initializers that are declared by superclasses or enclosing classes.

See Context Management, `@Nested` test class configuration, and the `@ContextConfiguration` javadocs for further details.

`@WebAppConfiguration`

`@WebAppConfiguration` is a class-level annotation that you can use to declare that the `ApplicationContext` loaded for an integration test should be a `WebApplicationContext`. The mere presence of `@WebAppConfiguration` on a test class ensures that a `WebApplicationContext` is loaded for the test, using the default value of `"file:src/main/webapp"` for the path to the root of the web application (that is, the resource base path). The resource base path is used behind the scenes to create a `MockServletContext`, which serves as the `ServletContext` for the test's `WebApplicationContext`.

The following example shows how to use the `@WebAppConfiguration` annotation:

Java

```
@ContextConfiguration
@WebAppConfiguration ①
class WebAppTests {
    // class body...
}
```

Kotlin

```
@ContextConfiguration
@WebAppConfiguration ①
class WebAppTests {
    // class body...
}
```

① The `@WebAppConfiguration` annotation.

To override the default, you can specify a different base resource path by using the implicit `value` attribute. Both `classpath:` and `file:` resource prefixes are supported. If no resource prefix is supplied, the path is assumed to be a file system resource. The following example shows how to specify a classpath resource:

Java

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources") ①
class WebAppTests {
    // class body...
}
```

① Specifying a classpath resource.

Kotlin

```
@ContextConfiguration
@WebAppConfiguration("classpath:test-web-resources") ①
class WebAppTests {
    // class body...
}
```

① Specifying a classpath resource.

Note that `@WebAppConfiguration` must be used in conjunction with `@ContextConfiguration`, either within a single test class or within a test class hierarchy. See the `@WebAppConfiguration` javadoc for further details.

`@ContextHierarchy`

`@ContextHierarchy` is a class-level annotation that is used to define a hierarchy of `ApplicationContext` instances for integration tests. `@ContextHierarchy` should be declared with a list of one or more `@ContextConfiguration` instances, each of which defines a level in the context hierarchy. The following examples demonstrate the use of `@ContextHierarchy` within a single test class (`@ContextHierarchy` can also be used within a test class hierarchy):

Java

```
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")
})
class ContextHierarchyTests {
    // class body...
}
```

Kotlin

```
@ContextHierarchy(
    ContextConfiguration("/parent-config.xml"),
    ContextConfiguration("/child-config.xml"))
class ContextHierarchyTests {
    // class body...
}
```

Java

```
@WebAppConfiguration
@ContextHierarchy({
    @ContextConfiguration(classes = AppConfig.class),
    @ContextConfiguration(classes = WebConfig.class)
})
class WebIntegrationTests {
    // class body...
}
```

Kotlin

```
@WebAppConfiguration
@ContextHierarchy(
        ContextConfiguration(classes = [AppConfig::class]),
        ContextConfiguration(classes = [WebConfig::class]))
class WebIntegrationTests {
    // class body...
}
```

If you need to merge or override the configuration for a given level of the context hierarchy within a test class hierarchy, you must explicitly name that level by supplying the same value to the `name` attribute in `@ContextConfiguration` at each corresponding level in the class hierarchy. See Context Hierarchies and the `@ContextHierarchy` javadoc for further examples.

`@ActiveProfiles`

`@ActiveProfiles` is a class-level annotation that is used to declare which bean definition profiles should be active when loading an `ApplicationContext` for an integration test.

The following example indicates that the `dev` profile should be active:

Java

```
@ContextConfiguration
@ActiveProfiles("dev") ①
class DeveloperTests {
    // class body...
}
```

① Indicate that the `dev` profile should be active.

Kotlin

```
@ContextConfiguration
@ActiveProfiles("dev") ①
class DeveloperTests {
    // class body...
}
```

① Indicate that the `dev` profile should be active.

The following example indicates that both the `dev` and the `integration` profiles should be active:

Java

```
@ContextConfiguration
@ActiveProfiles({"dev", "integration"}) ①
class DeveloperIntegrationTests {
    // class body...
}
```

① Indicate that the `dev` and `integration` profiles should be active.

Kotlin

```
@ContextConfiguration
@ActiveProfiles(["dev", "integration"]) ①
class DeveloperIntegrationTests {
    // class body...
}
```

① Indicate that the `dev` and `integration` profiles should be active.

`@ActiveProfiles` provides support for inheriting active bean definition profiles declared by superclasses and enclosing classes by default. You can also resolve active bean definition profiles programmatically by implementing a custom `ActiveProfilesResolver` and registering it by using the `resolver` attribute of `@ActiveProfiles`.

See Context Configuration with Environment Profiles, `@Nested` test class configuration, and the `@ActiveProfiles` javadoc for examples and further details.

`@TestPropertySource`

`@TestPropertySource` is a class-level annotation that you can use to configure the locations of properties files and inlined properties to be added to the set of `PropertySources` in the `Environment` for an `ApplicationContext` loaded for an integration test.

The following example demonstrates how to declare a properties file from the classpath:

Java

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Get properties from `test.properties` in the root of the classpath.

Kotlin

```
@ContextConfiguration
@TestPropertySource("/test.properties") ①
class MyIntegrationTests {
    // class body...
}
```

① Get properties from `test.properties` in the root of the classpath.

The following example demonstrates how to declare inlined properties:

Java

```
@ContextConfiguration
@TestPropertySource(properties = { "timezone = GMT", "port: 4242" }) ①
class MyIntegrationTests {
    // class body...
}
```

① Declare `timezone` and `port` properties.

Kotlin

```
@ContextConfiguration
@TestPropertySource(properties = ["timezone = GMT", "port: 4242"]) ①
class MyIntegrationTests {
    // class body...
}
```

① Declare `timezone` and `port` properties.

See Context Configuration with Test Property Sources for examples and further details.

`@DynamicPropertySource`

`@DynamicPropertySource` is a method-level annotation that you can use to register *dynamic* properties to be added to the set of `PropertySources` in the `Environment` for an `ApplicationContext` loaded for an integration test. Dynamic properties are useful when you do not know the value of the properties upfront – for example, if the properties are managed by an external resource such as for a container managed by the Testcontainers project.

The following example demonstrates how to register a dynamic property:

Java

```
@ContextConfiguration
class MyIntegrationTests {

    static MyExternalServer server = // ...

    @DynamicPropertySource ①
    static void dynamicProperties(DynamicPropertyRegistry registry) { ②
        registry.add("server.port", server::getPort); ③
    }

    // tests ...
}
```

① Annotate a `static` method with `@DynamicPropertySource`.

② Accept a `DynamicPropertyRegistry` as an argument.

③ Register a dynamic `server.port` property to be retrieved lazily from the server.

Kotlin

```
@ContextConfiguration
class MyIntegrationTests {

    companion object {

        @JvmStatic
        val server: MyExternalServer = // ...

        @DynamicPropertySource ①
        @JvmStatic
        fun dynamicProperties(registry: DynamicPropertyRegistry) { ②
            registry.add("server.port", server::getPort) ③
        }
    }

    // tests ...
}
```

① Annotate a `static` method with `@DynamicPropertySource`.

② Accept a `DynamicPropertyRegistry` as an argument.

③ Register a dynamic `server.port` property to be retrieved lazily from the server.

See Context Configuration with Dynamic Property Sources for further details.

`@DirtiesContext`

`@DirtiesContext` indicates that the underlying Spring `ApplicationContext` has been dirtied during the execution of a test (that is, the test modified or corrupted it in some manner — for example, by changing the state of a singleton bean) and should be closed. When an application context is marked as dirty, it is removed from the testing framework's cache and closed. As a consequence, the underlying Spring container is rebuilt for any subsequent test that requires a context with the same configuration metadata.

You can use `@DirtiesContext` as both a class-level and a method-level annotation within the same class or class hierarchy. In such scenarios, the `ApplicationContext` is marked as dirty before or after any such annotated method as well as before or after the current test class, depending on the configured `methodMode` and `classMode`.

The following examples explain when the context would be dirtied for various configuration scenarios:

- Before the current test class, when declared on a class with class mode set to `BEFORE_CLASS`.

Java

```
@DirtiesContext(classMode = BEFORE_CLASS) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before the current test class.

Kotlin

```
@DirtiesContext(classMode = BEFORE_CLASS) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before the current test class.

- After the current test class, when declared on a class with class mode set to `AFTER_CLASS` (i.e., the default class mode).

Java

```
@DirtiesContext ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after the current test class.

Kotlin

```
@DirtiesContext ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after the current test class.

- Before each test method in the current test class, when declared on a class with class mode set to `BEFORE_EACH_TEST_METHOD.`

Java

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before each test method.

Kotlin

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD) ①
class FreshContextTests {
    // some tests that require a new Spring container
}
```

① Dirty the context before each test method.

- After each test method in the current test class, when declared on a class with class mode set to `AFTER_EACH_TEST_METHOD`.

Java

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD) ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after each test method.

Kotlin

```
@DirtiesContext(classMode = AFTER_EACH_TEST_METHOD) ①
class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

① Dirty the context after each test method.

- Before the current test, when declared on a method with the method mode set to `BEFORE_METHOD`.

Java

```
@DirtiesContext(methodMode = BEFORE_METHOD) ①
@Test
void testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

① Dirty the context before the current test method.

Kotlin

```
@DirtiesContext(methodMode = BEFORE_METHOD) ①
@Test
fun testProcessWhichRequiresFreshAppCtx() {
    // some logic that requires a new Spring container
}
```

① Dirty the context before the current test method.

- After the current test, when declared on a method with the method mode set to `AFTER_METHOD` (i.e., the default method mode).

  Java

  ```
  @DirtiesContext ①
  @Test
  void testProcessWhichDirtiesAppCtx() {
      // some logic that results in the Spring container being dirtied
  }
  ```

  ① Dirty the context after the current test method.

  Kotlin

  ```
  @DirtiesContext ①
  @Test
  fun testProcessWhichDirtiesAppCtx() {
      // some logic that results in the Spring container being dirtied
  }
  ```

  ① Dirty the context after the current test method.

If you use `@DirtiesContext` in a test whose context is configured as part of a context hierarchy with `@ContextHierarchy`, you can use the `hierarchyMode` flag to control how the context cache is cleared. By default, an exhaustive algorithm is used to clear the context cache, including not only the current level but also all other context hierarchies that share an ancestor context common to the current test. All `ApplicationContext` instances that reside in a sub-hierarchy of the common ancestor context are removed from the context cache and closed. If the exhaustive algorithm is overkill for a particular use case, you can specify the simpler current level algorithm, as the following example shows.

Java

```
@ContextHierarchy({
    @ContextConfiguration("/parent-config.xml"),
    @ContextConfiguration("/child-config.xml")
})
class BaseTests {
    // class body...
}

class ExtendedTests extends BaseTests {

    @Test
    @DirtiesContext(hierarchyMode = CURRENT_LEVEL) ①
    void test() {
        // some logic that results in the child context being dirtied
    }
}
```

① Use the current-level algorithm.

Kotlin

```
@ContextHierarchy(
    ContextConfiguration("/parent-config.xml"),
    ContextConfiguration("/child-config.xml"))
open class BaseTests {
    // class body...
}

class ExtendedTests : BaseTests() {

    @Test
    @DirtiesContext(hierarchyMode = CURRENT_LEVEL) ①
    fun test() {
        // some logic that results in the child context being dirtied
    }
}
```

① Use the current-level algorithm.

For further details regarding the `EXHAUSTIVE` and `CURRENT_LEVEL` algorithms, see the `DirtiesContext.HierarchyMode` javadoc.

`@TestExecutionListeners`

`@TestExecutionListeners` is used to register listeners for a particular test class, its subclasses, and its nested classes. If you wish to register a listener globally, you should register it via the automatic discovery mechanism described in `TestExecutionListener` Configuration.

The following example shows how to register two `TestExecutionListener` implementations:

Java

```
@ContextConfiguration
@TestExecutionListeners({CustomTestExecutionListener.class,
AnotherTestExecutionListener.class}) ①
class CustomTestExecutionListenerTests {
    // class body...
}
```

① Register two `TestExecutionListener` implementations.

Kotlin

```
@ContextConfiguration
@TestExecutionListeners(CustomTestExecutionListener::class,
AnotherTestExecutionListener::class) ①
class CustomTestExecutionListenerTests {
    // class body...
}
```

① Register two `TestExecutionListener` implementations.

By default, `@TestExecutionListeners` provides support for inheriting listeners from superclasses or enclosing classes. See `@Nested` test class configuration and the `@TestExecutionListeners` javadoc for an example and further details. If you discover that you need to switch back to using the default `TestExecutionListener` implementations, see the note in Registering `TestExecutionListener` Implementations.

`@RecordApplicationEvents`

`@RecordApplicationEvents` is a class-level annotation that is used to instruct the *Spring TestContext Framework* to record all application events that are published in the `ApplicationContext` during the execution of a single test.

The recorded events can be accessed via the `ApplicationEvents` API within tests.

See Application Events and the `@RecordApplicationEvents` javadoc for an example and further details.

`@Commit`

`@Commit` indicates that the transaction for a transactional test method should be committed after the test method has completed. You can use `@Commit` as a direct replacement for `@Rollback(false)` to more explicitly convey the intent of the code. Analogous to `@Rollback`, `@Commit` can also be declared as a class-level or method-level annotation.

The following example shows how to use the `@Commit` annotation:

Java

```
@Commit ①
@Test
void testProcessWithoutRollback() {
    // ...
}
```

① Commit the result of the test to the database.

Kotlin

```
@Commit ①
@Test
fun testProcessWithoutRollback() {
    // ...
}
```

① Commit the result of the test to the database.

`@Rollback`

`@Rollback` indicates whether the transaction for a transactional test method should be rolled back after the test method has completed. If `true`, the transaction is rolled back. Otherwise, the transaction is committed (see also `@Commit`). Rollback for integration tests in the Spring TestContext Framework defaults to `true` even if `@Rollback` is not explicitly declared.

When declared as a class-level annotation, `@Rollback` defines the default rollback semantics for all test methods within the test class hierarchy. When declared as a method-level annotation, `@Rollback` defines rollback semantics for the specific test method, potentially overriding class-level `@Rollback` or `@Commit` semantics.

The following example causes a test method's result to not be rolled back (that is, the result is committed to the database):

Java

```
@Rollback(false) ①
@Test
void testProcessWithoutRollback() {
    // ...
}
```

① Do not roll back the result.

Kotlin

```
@Rollback(false) ①
@Test
fun testProcessWithoutRollback() {
    // ...
}
```

① Do not roll back the result.

`@BeforeTransaction`

`@BeforeTransaction` indicates that the annotated `void` method should be run before a transaction is started, for test methods that have been configured to run within a transaction by using Spring's `@Transactional` annotation. `@BeforeTransaction` methods are not required to be `public` and may be declared on Java 8-based interface default methods.

The following example shows how to use the `@BeforeTransaction` annotation:

Java

```
@BeforeTransaction ①
void beforeTransaction() {
    // logic to be run before a transaction is started
}
```

① Run this method before a transaction.

Kotlin

```
@BeforeTransaction ①
fun beforeTransaction() {
    // logic to be run before a transaction is started
}
```

① Run this method before a transaction.

`@AfterTransaction`

`@AfterTransaction` indicates that the annotated `void` method should be run after a transaction is ended, for test methods that have been configured to run within a transaction by using Spring's `@Transactional` annotation. `@AfterTransaction` methods are not required to be `public` and may be declared on Java 8-based interface default methods.

Java

```
@AfterTransaction ①
void afterTransaction() {
    // logic to be run after a transaction has ended
}
```

① Run this method after a transaction.

Kotlin

```
@AfterTransaction ①
fun afterTransaction() {
    // logic to be run after a transaction has ended
}
```

① Run this method after a transaction.

`@Sql`

`@Sql` is used to annotate a test class or test method to configure SQL scripts to be run against a given database during integration tests. The following example shows how to use it:

Java

```
@Test
@Sql({"/test-schema.sql", "/test-user-data.sql"}) ①
void userTest() {
    // run code that relies on the test schema and test data
}
```

① Run two scripts for this test.

Kotlin

```
@Test
@Sql("/test-schema.sql", "/test-user-data.sql") ①
fun userTest() {
    // run code that relies on the test schema and test data
}
```

① Run two scripts for this test.

See Executing SQL scripts declaratively with @Sql for further details.

`@SqlConfig`

`@SqlConfig` defines metadata that is used to determine how to parse and run SQL scripts configured with the `@Sql` annotation. The following example shows how to use it:

Java

```
@Test
@Sql(
    scripts = "/test-user-data.sql",
    config = @SqlConfig(commentPrefix = "`", separator = "@@") ①
)
void userTest() {
    // run code that relies on the test data
}
```

① Set the comment prefix and the separator in SQL scripts.

Kotlin

```
@Test
@Sql("/test-user-data.sql", config = SqlConfig(commentPrefix = "`", separator = "@@"))
①
fun userTest() {
    // run code that relies on the test data
}
```

① Set the comment prefix and the separator in SQL scripts.

`@SqlMergeMode`

`@SqlMergeMode` is used to annotate a test class or test method to configure whether method-level `@Sql` declarations are merged with class-level `@Sql` declarations. If `@SqlMergeMode` is not declared on a test class or test method, the `OVERRIDE` merge mode will be used by default. With the `OVERRIDE` mode, method-level `@Sql` declarations will effectively override class-level `@Sql` declarations.

Note that a method-level `@SqlMergeMode` declaration overrides a class-level declaration.

The following example shows how to use `@SqlMergeMode` at the class level.

Java

```
@SpringJUnitConfig(TestConfig.class)
@Sql("/test-schema.sql")
@SqlMergeMode(MERGE) ①
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    void standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for all test methods in the class.

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
@Sql("/test-schema.sql")
@SqlMergeMode(MERGE) ①
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    fun standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for all test methods in the class.

The following example shows how to use `@SqlMergeMode` at the method level.

Java

```
@SpringJUnitConfig(TestConfig.class)
@Sql("/test-schema.sql")
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    @SqlMergeMode(MERGE) ①
    void standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for a specific test method.

Kotlin

```
@SpringJUnitConfig(TestConfig::class)
@Sql("/test-schema.sql")
class UserTests {

    @Test
    @Sql("/user-test-data-001.sql")
    @SqlMergeMode(MERGE) ①
    fun standardUserProfile() {
        // run code that relies on test data set 001
    }
}
```

① Set the `@Sql` merge mode to `MERGE` for a specific test method.

`@SqlGroup`

`@SqlGroup` is a container annotation that aggregates several `@Sql` annotations. You can use `@SqlGroup` natively to declare several nested `@Sql` annotations, or you can use it in conjunction with Java 8's support for repeatable annotations, where `@Sql` can be declared several times on the same class or method, implicitly generating this container annotation. The following example shows how to declare an SQL group:

Java

```
@Test
@SqlGroup({ ①
    @Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),
    @Sql("/test-user-data.sql")
)}
void userTest() {
    // run code that uses the test schema and test data
}
```

① Declare a group of SQL scripts.

Kotlin

```
@Test
@SqlGroup( ①
    Sql("/test-schema.sql", config = SqlConfig(commentPrefix = "`")),
    Sql("/test-user-data.sql"))
fun userTest() {
    // run code that uses the test schema and test data
}
```

① Declare a group of SQL scripts.

Spring JUnit 4 Testing Annotations

The following annotations are supported only when used in conjunction with the SpringRunner, Spring's JUnit 4 rules, or Spring's JUnit 4 support classes:

- `@IfProfileValue`
- `@ProfileValueSourceConfiguration`
- `@Timed`
- `@Repeat`

`@IfProfileValue`

`@IfProfileValue` indicates that the annotated test is enabled for a specific testing environment. If the configured `ProfileValueSource` returns a matching `value` for the provided `name`, the test is enabled. Otherwise, the test is disabled and, effectively, ignored.

You can apply `@IfProfileValue` at the class level, the method level, or both. Class-level usage of

`@IfProfileValue` takes precedence over method-level usage for any methods within that class or its subclasses. Specifically, a test is enabled if it is enabled both at the class level and at the method level. The absence of `@IfProfileValue` means the test is implicitly enabled. This is analogous to the semantics of JUnit 4's `@Ignore` annotation, except that the presence of `@Ignore` always disables a test.

The following example shows a test that has an `@IfProfileValue` annotation:

Java

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation") ①
@Test
public void testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

① Run this test only when the Java vendor is "Oracle Corporation".

Kotlin

```
@IfProfileValue(name="java.vendor", value="Oracle Corporation") ①
@Test
fun testProcessWhichRunsOnlyOnOracleJvm() {
    // some logic that should run only on Java VMs from Oracle Corporation
}
```

① Run this test only when the Java vendor is "Oracle Corporation".

Alternatively, you can configure `@IfProfileValue` with a list of `values` (with `OR` semantics) to achieve TestNG-like support for test groups in a JUnit 4 environment. Consider the following example:

Java

```
@IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"}) ①
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

① Run this test for unit tests and integration tests.

Kotlin

```
@IfProfileValue(name="test-groups", values=["unit-tests", "integration-tests"]) ①
@Test
fun testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}
```

① Run this test for unit tests and integration tests.

`@ProfileValueSourceConfiguration`

`@ProfileValueSourceConfiguration` is a class-level annotation that specifies what type of `ProfileValueSource` to use when retrieving profile values configured through the `@IfProfileValue` annotation. If `@ProfileValueSourceConfiguration` is not declared for a test, `SystemProfileValueSource` is used by default. The following example shows how to use `@ProfileValueSourceConfiguration`:

Java

```
@ProfileValueSourceConfiguration(CustomProfileValueSource.class) ①
public class CustomProfileValueSourceTests {
    // class body...
}
```

① Use a custom profile value source.

Kotlin

```
@ProfileValueSourceConfiguration(CustomProfileValueSource::class) ①
class CustomProfileValueSourceTests {
    // class body...
}
```

① Use a custom profile value source.

`@Timed`

`@Timed` indicates that the annotated test method must finish execution in a specified time period (in milliseconds). If the text execution time exceeds the specified time period, the test fails.

The time period includes running the test method itself, any repetitions of the test (see `@Repeat`), as well as any setting up or tearing down of the test fixture. The following example shows how to use it:

Java

```
@Timed(millis = 1000) ①
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to run
}
```

① Set the time period for the test to one second.

Kotlin

```
@Timed(millis = 1000) ①
fun testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to run
}
```

① Set the time period for the test to one second.

Spring's `@Timed` annotation has different semantics than JUnit 4's `@Test(timeout=…)` support. Specifically, due to the manner in which JUnit 4 handles test execution timeouts (that is, by executing the test method in a separate `Thread`), `@Test(timeout=…)` preemptively fails the test if the test takes too long. Spring's `@Timed`, on the other hand, does not preemptively fail the test but rather waits for the test to complete before failing.

`@Repeat`

`@Repeat` indicates that the annotated test method must be run repeatedly. The number of times that the test method is to be run is specified in the annotation.

The scope of execution to be repeated includes execution of the test method itself as well as any setting up or tearing down of the test fixture. When used with the `SpringMethodRule`, the scope additionally includes preparation of the test instance by `TestExecutionListener` implementations. The following example shows how to use the `@Repeat` annotation:

Java

```
@Repeat(10) ①
@Test
public void testProcessRepeatedly() {
    // ...
}
```

① Repeat this test ten times.

Kotlin

```
@Repeat(10) ①
@Test
fun testProcessRepeatedly() {
    // ...
}
```

① Repeat this test ten times.

Spring JUnit Jupiter Testing Annotations

The following annotations are supported when used in conjunction with the `SpringExtension` and JUnit Jupiter (that is, the programming model in JUnit 5):

- `@SpringJUnitConfig`
- `@SpringJUnitWebConfig`
- `@TestConstructor`
- `@NestedTestConfiguration`
- `@EnabledIf`
- `@DisabledIf`

`@SpringJUnitConfig`

`@SpringJUnitConfig` is a composed annotation that combines `@ExtendWith(SpringExtension.class)` from JUnit Jupiter with `@ContextConfiguration` from the Spring TestContext Framework. It can be used at the class level as a drop-in replacement for `@ContextConfiguration`. With regard to configuration options, the only difference between `@ContextConfiguration` and `@SpringJUnitConfig` is that component classes may be declared with the `value` attribute in `@SpringJUnitConfig`.

The following example shows how to use the `@SpringJUnitConfig` annotation to specify a configuration class:

Java

```
@SpringJUnitConfig(TestConfig.class) ①
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the configuration class.

Kotlin

```
@SpringJUnitConfig(TestConfig::class) ①
class ConfigurationClassJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the configuration class.

The following example shows how to use the `@SpringJUnitConfig` annotation to specify the location of a configuration file:

Java

```
@SpringJUnitConfig(locations = "/test-config.xml") ①
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the location of a configuration file.

Kotlin

```
@SpringJUnitConfig(locations = ["/test-config.xml"]) ①
class XmlJUnitJupiterSpringTests {
    // class body...
}
```

① Specify the location of a configuration file.

See Context Management as well as the javadoc for `@SpringJUnitConfig` and `@ContextConfiguration`

for further details.

`@SpringJUnitWebConfig`

`@SpringJUnitWebConfig` is a composed annotation that combines `@ExtendWith(SpringExtension.class)` from JUnit Jupiter with `@ContextConfiguration` and `@WebAppConfiguration` from the Spring TestContext Framework. You can use it at the class level as a drop-in replacement for `@ContextConfiguration` and `@WebAppConfiguration`. With regard to configuration options, the only difference between `@ContextConfiguration` and `@SpringJUnitWebConfig` is that you can declare component classes by using the `value` attribute in `@SpringJUnitWebConfig`. In addition, you can override the `value` attribute from `@WebAppConfiguration` only by using the `resourcePath` attribute in `@SpringJUnitWebConfig`.

The following example shows how to use the `@SpringJUnitWebConfig` annotation to specify a configuration class:

Java

```
@SpringJUnitWebConfig(TestConfig.class) ①
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the configuration class.

Kotlin

```
@SpringJUnitWebConfig(TestConfig::class) ①
class ConfigurationClassJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the configuration class.

The following example shows how to use the `@SpringJUnitWebConfig` annotation to specify the location of a configuration file:

Java

```
@SpringJUnitWebConfig(locations = "/test-config.xml") ①
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the location of a configuration file.

Kotlin

```
@SpringJUnitWebConfig(locations = ["/test-config.xml"]) ①
class XmlJUnitJupiterSpringWebTests {
    // class body...
}
```

① Specify the location of a configuration file.

See Context Management as well as the javadoc for `@SpringJUnitWebConfig`, `@ContextConfiguration`, and `@WebAppConfiguration` for further details.

`@TestConstructor`

`@TestConstructor` is a type-level annotation that is used to configure how the parameters of a test class constructor are autowired from components in the test's `ApplicationContext`.

If `@TestConstructor` is not present or meta-present on a test class, the default *test constructor autowire mode* will be used. See the tip below for details on how to change the default mode. Note, however, that a local declaration of `@Autowired` on a constructor takes precedence over both `@TestConstructor` and the default mode.

Changing the default test constructor autowire mode

The default *test constructor autowire mode* can be changed by setting the `spring.test.constructor.autowire.mode` JVM system property to `all`. Alternatively, the default mode may be set via the `SpringProperties` mechanism.

As of Spring Framework 5.3, the default mode may also be configured as a JUnit Platform configuration parameter.

If the `spring.test.constructor.autowire.mode` property is not set, test class constructors will not be automatically autowired.

As of Spring Framework 5.2, `@TestConstructor` is only supported in conjunction with the `SpringExtension` for use with JUnit Jupiter. Note that the `SpringExtension` is often automatically registered for you – for example, when using annotations such as `@SpringJUnitConfig` and `@SpringJUnitWebConfig` or various test-related annotations from Spring Boot Test.

`@NestedTestConfiguration`

`@NestedTestConfiguration` is a type-level annotation that is used to configure how Spring test configuration annotations are processed within enclosing class hierarchies for inner test classes.

If `@NestedTestConfiguration` is not present or meta-present on a test class, in its supertype hierarchy, or in its enclosing class hierarchy, the default *enclosing configuration inheritance mode* will be used. See the tip below for details on how to change the default mode.

Changing the default enclosing configuration inheritance mode

The default *enclosing configuration inheritance mode* is `INHERIT`, but it can be changed by setting the `spring.test.enclosing.configuration` JVM system property to `OVERRIDE`. Alternatively, the default mode may be set via the `SpringProperties` mechanism.

The Spring TestContext Framework honors `@NestedTestConfiguration` semantics for the following annotations.

- `@BootstrapWith`
- `@ContextConfiguration`
- `@WebAppConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DynamicPropertySource`
- `@DirtiesContext`
- `@TestExecutionListeners`
- `@RecordApplicationEvents`
- `@Transactional`
- `@Commit`
- `@Rollback`
- `@Sql`
- `@SqlConfig`
- `@SqlMergeMode`
- `@TestConstructor`

The use of `@NestedTestConfiguration` typically only makes sense in conjunction with `@Nested` test classes in JUnit Jupiter; however, there may be other testing frameworks with support for Spring and nested test classes that make use of this annotation.

See `@Nested` test class configuration for an example and further details.

`@EnabledIf`

`@EnabledIf` is used to signal that the annotated JUnit Jupiter test class or test method is enabled and should be run if the supplied `expression` evaluates to `true`. Specifically, if the expression evaluates to `Boolean.TRUE` or a `String` equal to `true` (ignoring case), the test is enabled. When applied at the class level, all test methods within that class are automatically enabled by default as well.

Expressions can be any of the following:

- Spring Expression Language (SpEL) expression. For example: `@EnabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")`
- Placeholder for a property available in the Spring `Environment`. For example: `@EnabledIf("${smoke.tests.enabled}")`
- Text literal. For example: `@EnabledIf("true")`

Note, however, that a text literal that is not the result of dynamic resolution of a property placeholder is of zero practical value, since `@EnabledIf("false")` is equivalent to `@Disabled` and `@EnabledIf("true")` is logically meaningless.

You can use `@EnabledIf` as a meta-annotation to create custom composed annotations. For example, you can create a custom `@EnabledOnMac` annotation as follows:

Java

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@EnabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Enabled on Mac OS"
)
public @interface EnabledOnMac {}
```

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@EnabledIf(
        expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
        reason = "Enabled on Mac OS"
)
annotation class EnabledOnMac {}
```

`@EnabledOnMac` is meant only as an example of what is possible. If you have that exact use case, please use the built-in `@EnabledOnOs(MAC)` support in JUnit Jupiter.

Since JUnit 5.7, JUnit Jupiter also has a condition annotation named `@EnabledIf`. Thus, if you wish to use Spring's `@EnabledIf` support make sure you import the annotation type from the correct package.

`@DisabledIf`

`@DisabledIf` is used to signal that the annotated JUnit Jupiter test class or test method is disabled and should not be run if the supplied `expression` evaluates to `true`. Specifically, if the expression evaluates to `Boolean.TRUE` or a `String` equal to `true` (ignoring case), the test is disabled. When applied at the class level, all test methods within that class are automatically disabled as well.

Expressions can be any of the following:

- Spring Expression Language (SpEL) expression. For example: `@DisabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")`
- Placeholder for a property available in the Spring `Environment`. For example: `@DisabledIf("${smoke.tests.disabled}")`
- Text literal. For example: `@DisabledIf("true")`

Note, however, that a text literal that is not the result of dynamic resolution of a property placeholder is of zero practical value, since `@DisabledIf("true")` is equivalent to `@Disabled` and `@DisabledIf("false")` is logically meaningless.

You can use `@DisabledIf` as a meta-annotation to create custom composed annotations. For example, you can create a custom `@DisabledOnMac` annotation as follows:

Java

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@DisabledIf(
    expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
    reason = "Disabled on Mac OS"
)
public @interface DisabledOnMac {}
```

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@DisabledIf(
        expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",
        reason = "Disabled on Mac OS"
)
annotation class DisabledOnMac {}
```

`@DisabledOnMac` is meant only as an example of what is possible. If you have that exact use case, please use the built-in `@DisabledOnOs(MAC)` support in JUnit Jupiter.

Since JUnit 5.7, JUnit Jupiter also has a condition annotation named `@DisabledIf`. Thus, if you wish to use Spring's `@DisabledIf` support make sure you import the annotation type from the correct package.

Meta-Annotation Support for Testing

You can use most test-related annotations as meta-annotations to create custom composed annotations and reduce configuration duplication across a test suite.

You can use each of the following as a meta-annotation in conjunction with the TestContext framework.

- `@BootstrapWith`
- `@ContextConfiguration`
- `@ContextHierarchy`
- `@ActiveProfiles`
- `@TestPropertySource`
- `@DirtiesContext`
- `@WebAppConfiguration`
- `@TestExecutionListeners`
- `@Transactional`
- `@BeforeTransaction`
- `@AfterTransaction`
- `@Commit`
- `@Rollback`
- `@Sql`
- `@SqlConfig`
- `@SqlMergeMode`
- `@SqlGroup`
- `@Repeat` *(only supported on JUnit 4)*
- `@Timed` *(only supported on JUnit 4)*
- `@IfProfileValue` *(only supported on JUnit 4)*
- `@ProfileValueSourceConfiguration` *(only supported on JUnit 4)*
- `@SpringJUnitConfig` *(only supported on JUnit Jupiter)*
- `@SpringJUnitWebConfig` *(only supported on JUnit Jupiter)*
- `@TestConstructor` *(only supported on JUnit Jupiter)*
- `@NestedTestConfiguration` *(only supported on JUnit Jupiter)*
- `@EnabledIf` *(only supported on JUnit Jupiter)*
- `@DisabledIf` *(only supported on JUnit Jupiter)*

Consider the following example:

Java

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class UserRepositoryTests { }
```

Kotlin

```
@RunWith(SpringRunner::class)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
class OrderRepositoryTests { }

@RunWith(SpringRunner::class)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
class UserRepositoryTests { }
```

If we discover that we are repeating the preceding configuration across our JUnit 4-based test suite, we can reduce the duplication by introducing a custom composed annotation that centralizes the common test configuration for Spring, as follows:

Java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
annotation class TransactionalDevTestConfig { }
```

Then we can use our custom `@TransactionalDevTestConfig` annotation to simplify the configuration of individual JUnit 4 based test classes, as follows:

Java

```
@RunWith(SpringRunner.class)
@TransactionalDevTestConfig
public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)
@TransactionalDevTestConfig
public class UserRepositoryTests { }
```

Kotlin

```
@RunWith(SpringRunner::class)
@TransactionalDevTestConfig
class OrderRepositoryTests

@RunWith(SpringRunner::class)
@TransactionalDevTestConfig
class UserRepositoryTests
```

If we write tests that use JUnit Jupiter, we can reduce code duplication even further, since annotations in JUnit 5 can also be used as meta-annotations. Consider the following example:

Java

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
class OrderRepositoryTests { }

@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
class UserRepositoryTests { }
```

Kotlin

```
@ExtendWith(SpringExtension::class)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
class OrderRepositoryTests { }

@ExtendWith(SpringExtension::class)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
class UserRepositoryTests { }
```

If we discover that we are repeating the preceding configuration across our JUnit Jupiter-based test suite, we can reduce the duplication by introducing a custom composed annotation that centralizes the common test configuration for Spring and JUnit Jupiter, as follows:

Java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(SpringExtension.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTestConfig { }
```

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@ExtendWith(SpringExtension::class)
@ContextConfiguration("/app-config.xml", "/test-data-access-config.xml")
@ActiveProfiles("dev")
@Transactional
annotation class TransactionalDevTestConfig { }
```

Then we can use our custom `@TransactionalDevTestConfig` annotation to simplify the configuration of individual JUnit Jupiter based test classes, as follows:

Java

```
@TransactionalDevTestConfig
class OrderRepositoryTests { }

@TransactionalDevTestConfig
class UserRepositoryTests { }
```

Kotlin

```
@TransactionalDevTestConfig
class OrderRepositoryTests { }

@TransactionalDevTestConfig
class UserRepositoryTests { }
```

Since JUnit Jupiter supports the use of `@Test`, `@RepeatedTest`, `ParameterizedTest`, and others as meta-annotations, you can also create custom composed annotations at the test method level. For example, if we wish to create a composed annotation that combines the `@Test` and `@Tag` annotations from JUnit Jupiter with the `@Transactional` annotation from Spring, we could create an `@TransactionalIntegrationTest` annotation, as follows:

Java

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Transactional
@Tag("integration-test") // org.junit.jupiter.api.Tag
@Test // org.junit.jupiter.api.Test
public @interface TransactionalIntegrationTest { }
```

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@Transactional
@Tag("integration-test") // org.junit.jupiter.api.Tag
@Test // org.junit.jupiter.api.Test
annotation class TransactionalIntegrationTest { }
```

Then we can use our custom `@TransactionalIntegrationTest` annotation to simplify the configuration of individual JUnit Jupiter based test methods, as follows:

Java

```
@TransactionalIntegrationTest
void saveOrder() { }

@TransactionalIntegrationTest
void deleteOrder() { }
```

Kotlin

```
@TransactionalIntegrationTest
fun saveOrder() { }

@TransactionalIntegrationTest
fun deleteOrder() { }
```

For further details, see the Spring Annotation Programming Model wiki page.

3.9.2. Further Resources

See the following resources for more information about testing:

- JUnit: "A programmer-friendly testing framework for Java and the JVM". Used by the Spring Framework in its test suite and supported in the Spring TestContext Framework.
- TestNG: A testing framework inspired by JUnit with added support for test groups, data-driven testing, distributed testing, and other features. Supported in the Spring TestContext Framework
- AssertJ: "Fluent assertions for Java", including support for Java 8 lambdas, streams, and numerous other features.
- Mock Objects: Article in Wikipedia.
- MockObjects.com: Web site dedicated to mock objects, a technique for improving the design of code within test-driven development.
- Mockito: Java mock library based on the Test Spy pattern. Used by the Spring Framework in its test suite.
- EasyMock: Java library "that provides Mock Objects for interfaces (and objects through the class extension) by generating them on the fly using Java's proxy mechanism."
- JMock: Library that supports test-driven development of Java code with mock objects.
- DbUnit: JUnit extension (also usable with Ant and Maven) that is targeted at database-driven projects and, among other things, puts your database into a known state between test runs.
- Testcontainers: Java library that supports JUnit tests, providing lightweight, throwaway instances of common databases, Selenium web browsers, or anything else that can run in a Docker container.
- The Grinder: Java load testing framework.
- SpringMockK: Support for Spring Boot integration tests written in Kotlin using MockK instead of Mockito.

Chapter 4. Data Access

This part of the reference documentation is concerned with data access and the interaction between the data access layer and the business or service layer.

Spring's comprehensive transaction management support is covered in some detail, followed by thorough coverage of the various data access frameworks and technologies with which the Spring Framework integrates.

4.1. Transaction Management

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- A consistent programming model across different transaction APIs, such as Java Transaction API (JTA), JDBC, Hibernate, and the Java Persistence API (JPA).
- Support for declarative transaction management.
- A simpler API for programmatic transaction management than complex transaction APIs, such as JTA.
- Excellent integration with Spring's data access abstractions.

The following sections describe the Spring Framework's transaction features and technologies:

- Advantages of the Spring Framework's transaction support model describes why you would use the Spring Framework's transaction abstraction instead of EJB Container-Managed Transactions (CMT) or choosing to drive local transactions through a proprietary API, such as Hibernate.
- Understanding the Spring Framework transaction abstraction outlines the core classes and describes how to configure and obtain `DataSource` instances from a variety of sources.
- Synchronizing resources with transactions describes how the application code ensures that resources are created, reused, and cleaned up properly.
- Declarative transaction management describes support for declarative transaction management.
- Programmatic transaction management covers support for programmatic (that is, explicitly coded) transaction management.
- Transaction bound event describes how you could use application events within a transaction.

The chapter also includes discussions of best practices, application server integration, and solutions to common problems.

4.1.1. Advantages of the Spring Framework's Transaction Support Model

Traditionally, EE application developers have had two choices for transaction management: global or local transactions, both of which have profound limitations. Global and local transaction management is reviewed in the next two sections, followed by a discussion of how the Spring

Framework's transaction management support addresses the limitations of the global and local transaction models.

Global Transactions

Global transactions let you work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which is a cumbersome API (partly due to its exception model). Furthermore, a JTA `UserTransaction` normally needs to be sourced from JNDI, meaning that you also need to use JNDI in order to use JTA. The use of global transactions limits any potential reuse of application code, as JTA is normally only available in an application server environment.

Previously, the preferred way to use global transactions was through EJB CMT (Container Managed Transaction). CMT is a form of declarative transaction management (as distinguished from programmatic transaction management). EJB CMT removes the need for transaction-related JNDI lookups, although the use of EJB itself necessitates the use of JNDI. It removes most but not all of the need to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs (or at least behind a transactional EJB facade). The negatives of EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

Local Transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use but have a significant disadvantage: They cannot work across multiple transactional resources. For example, code that manages transactions by using a JDBC connection cannot run within a global JTA transaction. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

Spring Framework's Consistent Programming Model

Spring resolves the disadvantages of global and local transactions. It lets application developers use a consistent programming model in any environment. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Most users prefer declarative transaction management, which we recommend in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management and, hence, do not depend on the Spring Framework transaction API or any other transaction API.

Do you need an application server for transaction management?

The Spring Framework's transaction management support changes traditional rules as to when an enterprise Java application requires an application server.

In particular, you do not need an application server purely for declarative transactions through EJBs. In fact, even if your application server has powerful JTA capabilities, you may decide that the Spring Framework's declarative transactions offer more power and a more productive programming model than EJB CMT.

Typically, you need an application server's JTA capability only if your application needs to handle transactions across multiple resources, which is not a requirement for many applications. Many high-end applications use a single, highly scalable database (such as Oracle RAC) instead. Stand-alone transaction managers (such as Atomikos Transactions and JOTM) are other options. Of course, you may need other application server capabilities, such as Java Message Service (JMS) and Jakarta EE Connector Architecture (JCA).

The Spring Framework gives you the choice of when to scale your application to a fully loaded application server. Gone are the days when the only alternative to using EJB CMT or JTA was to write code with local transactions (such as those on JDBC connections) and face a hefty rework if you need that code to run within global, container-managed transactions. With the Spring Framework, only some of the bean definitions in your configuration file need to change (rather than your code).

4.1.2. Understanding the Spring Framework Transaction Abstraction

The key to the Spring transaction abstraction is the notion of a transaction strategy. A transaction strategy is defined by a `TransactionManager`, specifically the `org.springframework.transaction.PlatformTransactionManager` interface for imperative transaction management and the `org.springframework.transaction.ReactiveTransactionManager` interface for reactive transaction management. The following listing shows the definition of the `PlatformTransactionManager` API:

```
public interface PlatformTransactionManager extends TransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws
TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

This is primarily a service provider interface (SPI), although you can use it programmatically from your application code. Because `PlatformTransactionManager` is an interface, it can be easily mocked or stubbed as necessary. It is not tied to a lookup strategy, such as JNDI. `PlatformTransactionManager` implementations are defined like any other object (or bean) in the Spring Framework IoC

container. This benefit alone makes Spring Framework transactions a worthwhile abstraction, even when you work with JTA. You can test transactional code much more easily than if it used JTA directly.

Again, in keeping with Spring's philosophy, the `TransactionException` that can be thrown by any of the `PlatformTransactionManager` interface's methods is unchecked (that is, it extends the `java.lang.RuntimeException` class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle `TransactionException`. The salient point is that developers are not *forced* to do so.

The `getTransaction(..)` method returns a `TransactionStatus` object, depending on a `TransactionDefinition` parameter. The returned `TransactionStatus` might represent a new transaction or can represent an existing transaction, if a matching transaction exists in the current call stack. The implication in this latter case is that, as with Jakarta EE transaction contexts, a `TransactionStatus` is associated with a thread of execution.

As of Spring Framework 5.2, Spring also provides a transaction management abstraction for reactive applications that make use of reactive types or Kotlin Coroutines. The following listing shows the transaction strategy defined by `org.springframework.transaction.ReactiveTransactionManager`:

```
public interface ReactiveTransactionManager extends TransactionManager {

    Mono<ReactiveTransaction> getReactiveTransaction(TransactionDefinition definition)
throws TransactionException;

    Mono<Void> commit(ReactiveTransaction status) throws TransactionException;

    Mono<Void> rollback(ReactiveTransaction status) throws TransactionException;
}
```

The reactive transaction manager is primarily a service provider interface (SPI), although you can use it programmatically from your application code. Because `ReactiveTransactionManager` is an interface, it can be easily mocked or stubbed as necessary.

The `TransactionDefinition` interface specifies:

- Propagation: Typically, all code within a transaction scope runs in that transaction. However, you can specify the behavior if a transactional method is run when a transaction context already exists. For example, code can continue running in the existing transaction (the common case), or the existing transaction can be suspended and a new transaction created. Spring offers all of the transaction propagation options familiar from EJB CMT. To read about the semantics of transaction propagation in Spring, see Transaction Propagation.
- Isolation: The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- Timeout: How long this transaction runs before timing out and being automatically rolled back by the underlying transaction infrastructure.

- Read-only status: You can use a read-only transaction when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you use Hibernate.

These settings reflect standard transactional concepts. If necessary, refer to resources that discuss transaction isolation levels and other core transaction concepts. Understanding these concepts is essential to using the Spring Framework or any transaction management solution.

The `TransactionStatus` interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs. The following listing shows the `TransactionStatus` interface:

```
public interface TransactionStatus extends TransactionExecution, SavepointManager,
Flushable {

    @Override
    boolean isNewTransaction();

    boolean hasSavepoint();

    @Override
    void setRollbackOnly();

    @Override
    boolean isRollbackOnly();

    void flush();

    @Override
    boolean isCompleted();
}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `TransactionManager` implementation is absolutely essential. You typically define this implementation through dependency injection.

`TransactionManager` implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, and so on. The following examples show how you can define a local `PlatformTransactionManager` implementation (in this case, with plain JDBC.)

You can define a JDBC `DataSource` by creating a bean similar to the following:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
```

The related `PlatformTransactionManager` bean definition then has a reference to the `DataSource` definition. It should resemble the following example:

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

If you use JTA in a Jakarta EE container, then you use a container `DataSource`, obtained through JNDI, in conjunction with Spring's `JtaTransactionManager`. The following example shows what the JTA and JNDI lookup version would look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jee
        https://www.springframework.org/schema/jee/spring-jee.xsd">

    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

    <bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />

    <!-- other <bean/> definitions here -->

</beans>
```

The `JtaTransactionManager` does not need to know about the `DataSource` (or any other specific resources) because it uses the container's global transaction management infrastructure.

The preceding definition of the `dataSource` bean uses the `<jndi-lookup/>` tag from the `jee` namespace. For more information see The JEE Schema.

If you use JTA, your transaction manager definition should look the same, regardless of what data access technology you use, be it JDBC, Hibernate JPA, or any other supported technology. This is due to the fact that JTA transactions are global transactions, which can enlist any transactional resource.

In all Spring transaction setups, application code does not need to change. You can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

Hibernate Transaction Setup

You can also easily use Hibernate local transactions, as shown in the following examples. In this case, you need to define a Hibernate `LocalSessionFactoryBean`, which your application code can use to obtain Hibernate `Session` instances.

The `DataSource` bean definition is similar to the local JDBC example shown previously and, thus, is not shown in the following example.

If the `DataSource` (used by any non-JTA transaction manager) is looked up through JNDI and managed by a Jakarta EE container, it should be non-transactional, because the Spring Framework (rather than the Jakarta EE container) manages the transactions.

The `txManager` bean in this case is of the `HibernateTransactionManager` type. In the same way as the `DataSourceTransactionManager` needs a reference to the `DataSource`, the `HibernateTransactionManager` needs a reference to the `SessionFactory`. The following example declares `sessionFactory` and `txManager` beans:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=${hibernate.dialect}
        </value>
    </property>
</bean>

<bean id="txManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

If you use Hibernate and Jakarta EE container-managed JTA transactions, you should use the same `JtaTransactionManager` as in the previous JTA example for JDBC, as the following example shows. Also, it is recommended to make Hibernate aware of JTA through its transaction coordinator and possibly also its connection release mode configuration:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=${hibernate.dialect}
            hibernate.transaction.coordinator_class=jta

hibernate.connection.handling_mode=DELAYED_ACQUISITION_AND_RELEASE_AFTER_STATEMENT
        </value>
    </property>
</bean>

<bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Or alternatively, you may pass the `JtaTransactionManager` into your `LocalSessionFactoryBean` for enforcing the same defaults:

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list>

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=${hibernate.dialect}
        </value>
    </property>
    <property name="jtaTransactionManager" ref="txManager"/>
</bean>

<bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

4.1.3. Synchronizing Resources with Transactions

How to create different transaction managers and how they are linked to related resources that need to be synchronized to transactions (for example `DataSourceTransactionManager` to a JDBC `DataSource`, `HibernateTransactionManager` to a Hibernate `SessionFactory`, and so forth) should now be clear. This section describes how the application code (directly or indirectly, by using a persistence API such as JDBC, Hibernate, or JPA) ensures that these resources are created, reused, and cleaned up properly. The section also discusses how transaction synchronization is (optionally) triggered through the relevant `TransactionManager`.

High-level Synchronization Approach

The preferred approach is to use Spring's highest-level template-based persistence integration APIs or to use native ORM APIs with transaction-aware factory beans or proxies for managing the native resource factories. These transaction-aware solutions internally handle resource creation and reuse, cleanup, optional transaction synchronization of the resources, and exception mapping. Thus, user data access code does not have to address these tasks but can focus purely on non-boilerplate persistence logic. Generally, you use the native ORM API or take a template approach for JDBC access by using the `JdbcTemplate`. These solutions are detailed in subsequent sections of this reference documentation.

Low-level Synchronization Approach

Classes such as `DataSourceUtils` (for JDBC), `EntityManagerFactoryUtils` (for JPA), `SessionFactoryUtils` (for Hibernate), and so on exist at a lower level. When you want the application code to deal directly with the resource types of the native persistence APIs, you use these classes to ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions that occur in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the `getConnection()` method on the `DataSource`, you can instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class, as follows:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

If an existing transaction already has a connection synchronized (linked) to it, that instance is returned. Otherwise, the method call triggers the creation of a new connection, which is (optionally) synchronized to any existing transaction and made available for subsequent reuse in that same transaction. As mentioned earlier, any `SQLException` is wrapped in a Spring Framework `CannotGetJdbcConnectionException`, one of the Spring Framework's hierarchy of unchecked `DataAccessException` types. This approach gives you more information than can be obtained easily from the `SQLException` and ensures portability across databases and even across different persistence technologies.

This approach also works without Spring transaction management (transaction synchronization is optional), so you can use it whether or not you use Spring for transaction management.

Of course, once you have used Spring's JDBC support, JPA support, or Hibernate support, you generally prefer not to use `DataSourceUtils` or the other helper classes, because you are much happier working through the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring `JdbcTemplate` or `jdbc.object` package to simplify your use of JDBC, correct connection retrieval occurs behind the scenes and you need not write any special code.

`TransactionAwareDataSourceProxy`

At the very lowest level exists the `TransactionAwareDataSourceProxy` class. This is a proxy for a target `DataSource`, which wraps the target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource`, as provided by a Jakarta EE server.

You should almost never need or want to use this class, except when existing code must be called and passed a standard JDBC `DataSource` interface implementation. In that case, it is possible that this code is usable but is participating in Spring-managed transactions. You can write your new code by using the higher-level abstractions mentioned earlier.

4.1.4. Declarative Transaction Management

Most Spring Framework users choose declarative transaction management. This option has the least impact on application code and, hence, is most consistent with the ideals of a non-invasive lightweight container.

The Spring Framework's declarative transaction management is made possible with Spring aspect-oriented programming (AOP). However, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

The Spring Framework's declarative transaction management is similar to EJB CMT, in that you can specify transaction behavior (or lack of it) down to the individual method level. You can make a

`setRollbackOnly()` call within a transaction context, if necessary. The differences between the two types of transaction management are:

- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JTA transactions or local transactions by using JDBC, JPA, or Hibernate by adjusting the configuration files.
- You can apply the Spring Framework declarative transaction management to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative rollback rules, a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.
- The Spring Framework lets you customize transactional behavior by using AOP. For example, you can insert custom behavior in the case of transaction rollback. You can also add arbitrary advice, along with transactional advice. With EJB CMT, you cannot influence the container's transaction management, except with `setRollbackOnly()`.
- The Spring Framework does not support propagation of transaction contexts across remote calls, as high-end application servers do. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature, because, normally, one does not want transactions to span remote calls.

The concept of rollback rules is important. They let you specify which exceptions (and throwables) should cause automatic rollback. You can specify this declaratively, in configuration, not in Java code. So, although you can still call `setRollbackOnly()` on the `TransactionStatus` object to roll back the current transaction back, most often you can specify a rule that `MyApplicationException` must always result in rollback. The significant advantage to this option is that business objects do not depend on the transaction infrastructure. For example, they typically do not need to import Spring transaction APIs or other Spring APIs.

Although EJB container default behavior automatically rolls back the transaction on a system exception (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an application exception (that is, a checked exception other than `java.rmi.RemoteException`). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this behavior.

Understanding the Spring Framework's Declarative Transaction Implementation

It is not sufficient merely to tell you to annotate your classes with the `@Transactional` annotation, add `@EnableTransactionManagement` to your configuration, and expect you to understand how it all works. To provide a deeper understanding, this section explains the inner workings of the Spring Framework's declarative transaction infrastructure in the context of transaction-related issues.

The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled via AOP proxies and that the transactional advice is driven by metadata (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a `TransactionInterceptor` in conjunction with an appropriate `TransactionManager` implementation to drive transactions around method invocations.

Spring AOP is covered in the AOP section.

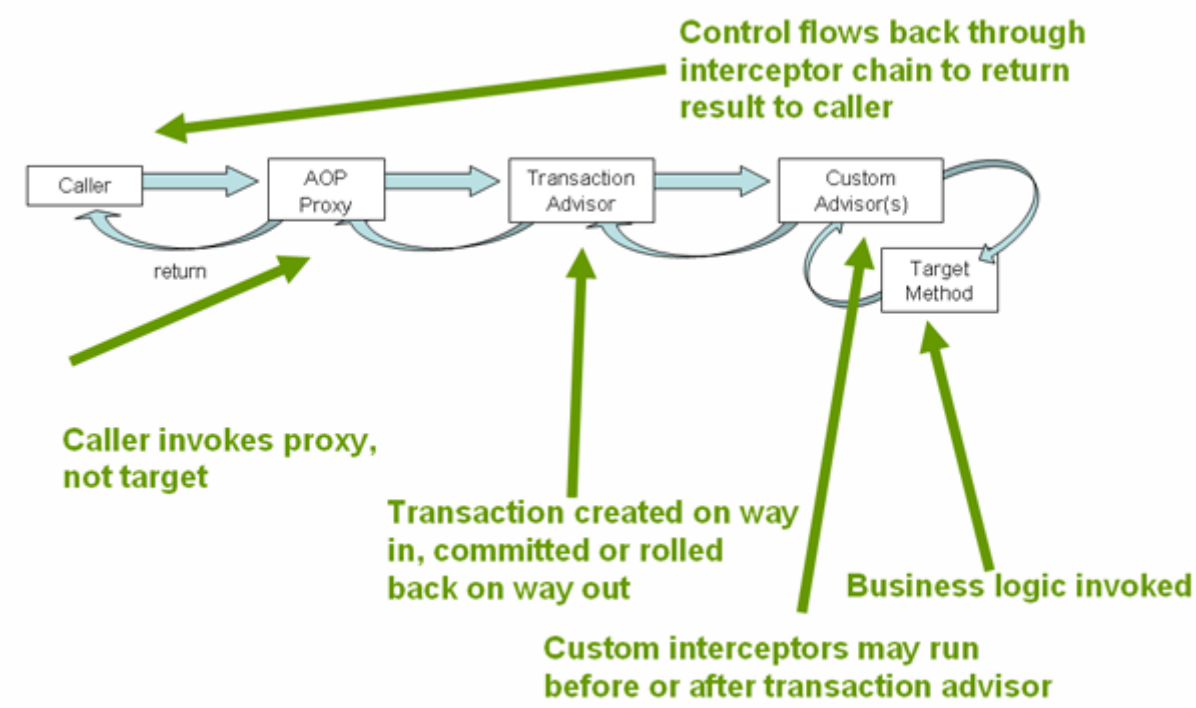
Spring Framework's `TransactionInterceptor` provides transaction management for imperative and reactive programming models. The interceptor detects the desired flavor of transaction management by inspecting the method return type. Methods returning a reactive type such as `Publisher` or Kotlin `Flow` (or a subtype of those) qualify for reactive transaction management. All other return types including `void` use the code path for imperative transaction management.

Transaction management flavors impact which transaction manager is required. Imperative transactions require a `PlatformTransactionManager`, while reactive transactions use `ReactiveTransactionManager` implementations.

`@Transactional` commonly works with thread-bound transactions managed by `PlatformTransactionManager`, exposing a transaction to all data access operations within the current execution thread. Note: This does *not* propagate to newly started threads within the method.

A reactive transaction managed by `ReactiveTransactionManager` uses the Reactor context instead of thread-local attributes. As a consequence, all participating data access operations need to execute within the same Reactor context in the same reactive pipeline.

The following image shows a conceptual view of calling a method on a transactional proxy:



Example of Declarative Transaction Implementation

Consider the following interface and its attendant implementation. This example uses `Foo` and `Bar` classes as placeholders so that you can concentrate on the transaction usage without focusing on a particular domain model. For the purposes of this example, the fact that the `DefaultFooService` class throws `UnsupportedOperationException` instances in the body of each implemented method is good.

That behavior lets you see transactions being created and then rolled back in response to the `UnsupportedOperationException` instance. The following listing shows the `FooService` interface:

Java

```
// the service interface that we want to make transactional

package x.y.service;

public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);

}
```

Kotlin

```
// the service interface that we want to make transactional

package x.y.service

interface FooService {

    fun getFoo(fooName: String): Foo

    fun getFoo(fooName: String, barName: String): Foo

    fun insertFoo(foo: Foo)

    fun updateFoo(foo: Foo)
}
```

The following example shows an implementation of the preceding interface:

Java

```
package x.y.service;

public class DefaultFooService implements FooService {

    @Override
    public Foo getFoo(String fooName) {
        // ...
    }

    @Override
    public Foo getFoo(String fooName, String barName) {
        // ...
    }

    @Override
    public void insertFoo(Foo foo) {
        // ...
    }

    @Override
    public void updateFoo(Foo foo) {
        // ...
    }
}
```

Kotlin

```
package x.y.service

class DefaultFooService : FooService {

    override fun getFoo(fooName: String): Foo {
        // ...
    }

    override fun getFoo(fooName: String, barName: String): Foo {
        // ...
    }

    override fun insertFoo(foo: Foo) {
        // ...
    }

    override fun updateFoo(foo: Foo) {
        // ...
    }
}
```

Assume that the first two methods of the `FooService` interface, `getFoo(String)` and `getFoo(String, String)`, must run in the context of a transaction with read-only semantics and that the other methods, `insertFoo(Foo)` and `updateFoo(Foo)`, must run in the context of a transaction with read-write semantics. The following configuration is explained in detail in the next few paragraphs:

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- this is the service object that we want to make transactional -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below)
-->
    <tx:advice id="txAdvice" transaction-manager="txManager">
        <!-- the transactional semantics... -->
        <tx:attributes>
            <!-- all methods starting with 'get' are read-only -->
            <tx:method name="get*" read-only="true"/>
            <!-- other methods use the default transaction settings (see below) -->
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- ensure that the above transactional advice runs for any execution
        of an operation defined by the FooService interface -->
    <aop:config>
        <aop:pointcut id="fooServiceOperation" expression="execution(*
x.y.service.FooService.*(..))"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
    </aop:config>

    <!-- don't forget the DataSource -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>
```

```
    <!-- similarly, don't forget the TransactionManager -->
    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- other <bean/> definitions here -->

</beans>
```

Examine the preceding configuration. It assumes that you want to make a service object, the `fooService` bean, transactional. The transaction semantics to apply are encapsulated in the `<tx:advice/>` definition. The `<tx:advice/>` definition reads as "all methods starting with `get` are to run in the context of a read-only transaction, and all other methods are to run with the default transaction semantics". The `transaction-manager` attribute of the `<tx:advice/>` tag is set to the name of the `TransactionManager` bean that is going to drive the transactions (in this case, the `txManager` bean).

You can omit the `transaction-manager` attribute in the transactional advice (`<tx:advice/>`) if the bean name of the `TransactionManager` that you want to wire in has the name `transactionManager`. If the `TransactionManager` bean that you want to wire in has any other name, you must use the `transaction-manager` attribute explicitly, as in the preceding example.

The `<aop:config/>` definition ensures that the transactional advice defined by the `txAdvice` bean runs at the appropriate points in the program. First, you define a pointcut that matches the execution of any operation defined in the `FooService` interface (`fooServiceOperation`). Then you associate the pointcut with the `txAdvice` by using an advisor. The result indicates that, at the execution of a `fooServiceOperation`, the advice defined by `txAdvice` is run.

The expression defined within the `<aop:pointcut/>` element is an AspectJ pointcut expression. See the AOP section for more details on pointcut expressions in Spring.

A common requirement is to make an entire service layer transactional. The best way to do this is to change the pointcut expression to match any operation in your service layer. The following example shows how to do so:

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(*
x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
</aop:config>
```

In the preceding example, it is assumed that all your service interfaces are defined in the `x.y.service` package. See the AOP section for more details.

Now that we have analyzed the configuration, you may be asking yourself, "What does all this

configuration actually do?"

The configuration shown earlier is used to create a transactional proxy around the object that is created from the `fooService` bean definition. The proxy is configured with the transactional advice so that, when an appropriate method is invoked on the proxy, a transaction is started, suspended, marked as read-only, and so on, depending on the transaction configuration associated with that method. Consider the following program that test drives the configuration shown earlier:

Java

```
public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml");
        FooService fooService = ctx.getBean(FooService.class);
        fooService.insertFoo(new Foo());
    }
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = ClassPathXmlApplicationContext("context.xml")
    val fooService = ctx.getBean<FooService>("fooService")
    fooService.insertFoo(Foo())
}
```

The output from running the preceding program should resemble the following (the Log4J output and the stack trace from the `UnsupportedOperationException` thrown by the `insertFoo(..)` method of the `DefaultFooService` class have been truncated for clarity):

```
<!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
for bean 'fooService' with 0 common interceptors and 1 specific interceptors

<!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

<!-- ... the insertFoo(..) method is now being invoked on the proxy -->
[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo

<!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name
[x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
[org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

<!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction
should rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on
x.y.service.FooService.insertFoo due to throwable
[java.lang.UnsupportedOperationException]

<!-- and the transaction is rolled back (by default, RuntimeException instances cause
rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
[org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException at
x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
<!-- AOP infrastructure stack trace elements removed for clarity -->
at $Proxy0.insertFoo(Unknown Source)
at Boot.main(Boot.java:11)
```

To use reactive transaction management the code has to use reactive types.

Spring Framework uses the `ReactiveAdapterRegistry` to determine whether a method return type is reactive.

The following listing shows a modified version of the previously used `FooService`, but this time the code uses reactive types:

Java

```
// the reactive service interface that we want to make transactional

package x.y.service;

public interface FooService {

    Flux<Foo> getFoo(String fooName);

    Publisher<Foo> getFoo(String fooName, String barName);

    Mono<Void> insertFoo(Foo foo);

    Mono<Void> updateFoo(Foo foo);

}
```

Kotlin

```
// the reactive service interface that we want to make transactional

package x.y.service

interface FooService {

    fun getFoo(fooName: String): Flow<Foo>

    fun getFoo(fooName: String, barName: String): Publisher<Foo>

    fun insertFoo(foo: Foo) : Mono<Void>

    fun updateFoo(foo: Foo) : Mono<Void>
}
```

The following example shows an implementation of the preceding interface:

Java

```
package x.y.service;

public class DefaultFooService implements FooService {

    @Override
    public Flux<Foo> getFoo(String fooName) {
        // ...
    }

    @Override
    public Publisher<Foo> getFoo(String fooName, String barName) {
        // ...
    }

    @Override
    public Mono<Void> insertFoo(Foo foo) {
        // ...
    }

    @Override
    public Mono<Void> updateFoo(Foo foo) {
        // ...
    }
}
```

Kotlin

```
package x.y.service

class DefaultFooService : FooService {

    override fun getFoo(fooName: String): Flow<Foo> {
        // ...
    }

    override fun getFoo(fooName: String, barName: String): Publisher<Foo> {
        // ...
    }

    override fun insertFoo(foo: Foo): Mono<Void> {
        // ...
    }

    override fun updateFoo(foo: Foo): Mono<Void> {
        // ...
    }
}
```

Imperative and reactive transaction management share the same semantics for transaction boundary and transaction attribute definitions. The main difference between imperative and reactive transactions is the deferred nature of the latter. `TransactionInterceptor` decorates the returned reactive type with a transactional operator to begin and clean up the transaction. Therefore, calling a transactional reactive method defers the actual transaction management to a subscription type that activates processing of the reactive type.

Another aspect of reactive transaction management relates to data escaping which is a natural consequence of the programming model.

Method return values of imperative transactions are returned from transactional methods upon successful termination of a method so that partially computed results do not escape the method closure.

Reactive transaction methods return a reactive wrapper type which represents a computation sequence along with a promise to begin and complete the computation.

A `Publisher` can emit data while a transaction is ongoing but not necessarily completed. Therefore, methods that depend upon successful completion of an entire transaction need to ensure completion and buffer results in the calling code.

Rolling Back a Declarative Transaction

The previous section outlined the basics of how to specify transactional settings for classes, typically service layer classes, declaratively in your application. This section describes how you can control the rollback of transactions in a simple, declarative fashion in XML configuration. For details on controlling rollback semantics declaratively with the `@Transactional` annotation, see `@Transactional` Settings.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an `Exception` from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code catches any unhandled `Exception` as it bubbles up the call stack and makes a determination whether to mark the transaction for rollback.

In its default configuration, the Spring Framework's transaction infrastructure code marks a transaction for rollback only in the case of runtime, unchecked exceptions. That is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Error` instances also, by default, result in a rollback). Checked exceptions that are thrown from a transactional method do not result in rollback in the default configuration.

You can configure exactly which `Exception` types mark a transaction for rollback, including checked exceptions by specifying *rollback rules*.

Rollback rules

Rollback rules determine if a transaction should be rolled back when a given exception is thrown, and the rules are based on exception types or exception patterns.

Rollback rules may be configured in XML via the `rollback-for` and `no-rollback-for` attributes, which allow rules to be defined as patterns. When using `@Transactional`, rollback rules may be configured via the `rollbackFor`/`noRollbackFor` and `rollbackForClassName`/`noRollbackForClassName` attributes, which allow rules to be defined based on exception types or patterns, respectively.

When a rollback rule is defined with an exception type, that type will be used to match against the type of a thrown exception and its super types, providing type safety and avoiding any unintentional matches that may occur when using a pattern. For example, a value of `jakarta.servlet.ServletException.class` will only match thrown exceptions of type `jakarta.servlet.ServletException` and its subclasses.

When a rollback rule is defined with an exception pattern, the pattern can be a fully qualified class name or a substring of a fully qualified class name for an exception type (which must be a subclass of `Throwable`), with no wildcard support at present. For example, a value of `"jakarta.servlet.ServletException"` or `"ServletException"` will match `jakarta.servlet.ServletException` and its subclasses.

You must carefully consider how specific a pattern is and whether to include package information (which isn't mandatory). For example, `"Exception"` will match nearly anything and will probably hide other rules. `"java.lang.Exception"` would be correct if `"Exception"` were meant to define a rule for all checked exceptions. With more unique exception names such as `"BaseBusinessException"` there is likely no need to use the fully qualified class name for the exception pattern.

Furthermore, pattern-based rollback rules may result in unintentional matches for similarly named exceptions and nested classes. This is due to the fact that a thrown exception is considered to be a match for a given pattern-based rollback rule if the name of the thrown exception contains the exception pattern configured for the rollback rule. For example, given a rule configured to match on `"com.example.CustomException"`, that rule will match against an exception named `com.example.CustomExceptionV2` (an exception in the same package as `CustomException` but with an additional suffix) or an exception named `com.example.CustomException$AnotherException` (an exception declared as a nested class in `CustomException`).

The following XML snippet demonstrates how to configure rollback for a checked, application-specific `Exception` type by supplying an *exception pattern* via the `rollback-for` attribute:

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
    <tx:method name="get*" read-only="true" rollback-for="NoProductInStockException"/>
    <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

If you do not want a transaction rolled back when an exception is thrown, you can also specify 'no rollback' rules. The following example tells the Spring Framework's transaction infrastructure to commit the attendant transaction even in the face of an unhandled `InstrumentNotFoundException`:

```
<tx:advice id="txAdvice">
    <tx:attributes>
    <tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>
    <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```

When the Spring Framework's transaction infrastructure catches an exception and consults the configured rollback rules to determine whether to mark the transaction for rollback, the strongest matching rule wins. So, in the case of the following configuration, any exception other than an `InstrumentNotFoundException` results in a rollback of the attendant transaction:

```
<tx:advice id="txAdvice">
    <tx:attributes>
    <tx:method name="*" rollback-for="Throwable" no-rollback-
for="InstrumentNotFoundException"/>
    </tx:attributes>
</tx:advice>
```

You can also indicate a required rollback programmatically. Although simple, this process is quite invasive and tightly couples your code to the Spring Framework's transaction infrastructure. The following example shows how to programmatically indicate a required rollback:

Java

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

Kotlin

```
fun resolvePosition() {
    try {
        // some business logic...
    } catch (ex: NoProductInStockException) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

You are strongly encouraged to use the declarative approach to rollback, if at all possible. Programmatic rollback is available should you absolutely need it, but its usage flies in the face of achieving a clean POJO-based architecture.

Configuring Different Transactional Semantics for Different Beans

Consider the scenario where you have a number of service layer objects, and you want to apply a totally different transactional configuration to each of them. You can do so by defining distinct `<aop:advisor/>` elements with differing `pointcut` and `advice-ref` attribute values.

As a point of comparison, first assume that all of your service layer classes are defined in a root `x.y.service` package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in `Service` have the default transactional configuration, you could write the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <aop:config>

        <aop:pointcut id="serviceOperation"
                expression="execution(* x.y.service..*Service.*(..))"/>

        <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

    </aop:config>

    <!-- these two beans will be transactional... -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>
    <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

    <!-- ... and these two beans won't -->
    <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right
package) -->
    <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in
'Service') -->

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a TransactionManager
omitted... -->

</beans>
```

The following example shows how to configure two distinct beans with totally different transactional settings:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <aop:config>

        <aop:pointcut id="defaultServiceOperation"
                expression="execution(* x.y.service.*Service.*(..))"/>

        <aop:pointcut id="noTxServiceOperation"
                expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>

        <aop:advisor pointcut-ref="defaultServiceOperation" advice-
ref="defaultTxAdvice"/>

        <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

    </aop:config>

    <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut)
-->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this bean will also be transactional, but with totally different
transactional settings -->
    <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

    <tx:advice id="defaultTxAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <tx:advice id="noTxAdvice">
        <tx:attributes>
            <tx:method name="*" propagation="NEVER"/>
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a TransactionManager
omitted... -->

</beans>
```

<tx:advice/> Settings

This section summarizes the various transactional settings that you can specify by using the `<tx:advice/>` tag. The default `<tx:advice/>` settings are:

- The propagation setting is `REQUIRED`.
- The isolation level is `DEFAULT`.
- The transaction is read-write.
- The transaction timeout defaults to the default timeout of the underlying transaction system or none if timeouts are not supported.
- Any `RuntimeException` triggers rollback, and any checked `Exception` does not.

You can change these default settings. The following table summarizes the various attributes of the `<tx:method/>` tags that are nested within `<tx:advice/>` and `<tx:attributes/>` tags:

Table 17. <tx:method/> settings

Attribute	Required?	Default	Description
`name`	Yes		Method names with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods (for example, `get*`, `handle*`, `on*Event`, and so forth).
`propagation`	No	`REQUIRED`	Transaction propagation behavior.
`isolation`	No	`DEFAULT`	Transaction isolation level. Only applicable to propagation settings of `REQUIRED` or `REQUIRES_NEW`.
`timeout`	No	-1	Transaction timeout (seconds). Only applicable to propagation `REQUIRED` or `REQUIRES_NEW`.
`read-only`	No	false	Read-write versus read-only transaction. Applies only to `REQUIRED` or `REQUIRES_NEW`.

Attribute	Required?	Default	Description
`rollback-for`	No		Comma-delimited list of `Exception` instances that trigger rollback. For example, `com.foo.MyBusinessException,ServletException`.
`no-rollback-for`	No		Comma-delimited list of `Exception` instances that do not trigger rollback. For example, `com.foo.MyBusinessException,ServletException`.

Using `@Transactional`

In addition to the XML-based declarative approach to transaction configuration, you can use an annotation-based approach. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code. There is not much danger of undue coupling, because code that is meant to be used transactionally is almost always deployed that way anyway.

The standard `jakarta.transaction.Transactional` annotation is also supported as a drop-in replacement to Spring's own annotation. Please refer to the JTA documentation for more details.

The ease-of-use afforded by the use of the `@Transactional` annotation is best illustrated with an example, which is explained in the text that follows. Consider the following class definition:

Java

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

    @Override
    public Foo getFoo(String fooName) {
        // ...
    }

    @Override
    public Foo getFoo(String fooName, String barName) {
        // ...
    }

    @Override
    public void insertFoo(Foo foo) {
        // ...
    }

    @Override
    public void updateFoo(Foo foo) {
        // ...
    }
}
```

Kotlin

```
// the service class that we want to make transactional
@Transactional
class DefaultFooService : FooService {

    override fun getFoo(fooName: String): Foo {
        // ...
    }

    override fun getFoo(fooName: String, barName: String): Foo {
        // ...
    }

    override fun insertFoo(foo: Foo) {
        // ...
    }

    override fun updateFoo(foo: Foo) {
        // ...
    }
}
```

Used at the class level as above, the annotation indicates a default for all methods of the declaring class (as well as its subclasses). Alternatively, each method can be annotated individually. See Method visibility and `@Transactional` for further details on which methods Spring considers transactional. Note that a class-level annotation does not apply to ancestor classes up the class hierarchy; in such a scenario, inherited methods need to be locally redeclared in order to participate in a subclass-level annotation.

When a POJO class such as the one above is defined as a bean in a Spring context, you can make the bean instance transactional through an `@EnableTransactionManagement` annotation in a `@Configuration` class. See the javadoc for full details.

In XML configuration, the `<tx:annotation-driven/>` tag provides similar convenience:

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- this is the service object that we want to make transactional -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- enable the configuration of transactional behavior based on annotations -->
    <!-- a TransactionManager is still required -->
    <tx:annotation-driven transaction-manager="txManager"/> ①

    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <!-- (this dependency is defined somewhere else) -->
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- other <bean/> definitions here -->

</beans>
```

① The line that makes the bean instance transactional.

You can omit the `transaction-manager` attribute in the `<tx:annotation-driven/>` tag if the bean name of the `TransactionManager` that you want to wire in has the name `transactionManager`. If the `TransactionManager` bean that you want to dependency-inject has any other name, you have to use the `transaction-manager` attribute, as in the preceding example.

Reactive transactional methods use reactive return types in contrast to imperative programming arrangements as the following listing shows:

Java

```
// the reactive service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

    @Override
    public Publisher<Foo> getFoo(String fooName) {
        // ...
    }

    @Override
    public Mono<Foo> getFoo(String fooName, String barName) {
        // ...
    }

    @Override
    public Mono<Void> insertFoo(Foo foo) {
        // ...
    }

    @Override
    public Mono<Void> updateFoo(Foo foo) {
        // ...
    }
}
```

Kotlin

```
// the reactive service class that we want to make transactional
@Transactional
class DefaultFooService : FooService {

    override fun getFoo(fooName: String): Flow<Foo> {
        // ...
    }

    override fun getFoo(fooName: String, barName: String): Mono<Foo> {
        // ...
    }

    override fun insertFoo(foo: Foo): Mono<Void> {
        // ...
    }

    override fun updateFoo(foo: Foo): Mono<Void> {
        // ...
    }
}
```

Note that there are special considerations for the returned `Publisher` with regards to Reactive Streams cancellation signals. See the Cancel Signals section under "Using the TransactionalOperator" for more details.

Method visibility and `@Transactional`

When you use transactional proxies with Spring's standard configuration, you should apply the `@Transactional` annotation only to methods with `public` visibility. If you do annotate `protected`, `private`, or package-visible methods with the `@Transactional` annotation, no error is raised, but the annotated method does not exhibit the configured transactional settings. If you need to annotate non-public methods, consider the tip in the following paragraph for class-based proxies or consider using AspectJ compile-time or load-time weaving (described later).

When using `@EnableTransactionManagement` in a `@Configuration` class, `protected` or package-visible methods can also be made transactional for class-based proxies by registering a custom `transactionAttributeSource` bean like in the following example. Note, however, that transactional methods in interface-based proxies must always be `public` and defined in the proxied interface.

```
/**
 * Register a custom AnnotationTransactionAttributeSource with the
 * publicMethodsOnly flag set to false to enable support for
 * protected and package-private @Transactional methods in
 * class-based proxies.
 *
 * @see
ProxyTransactionManagementConfiguration#transactionAttributeSource()
 */
@Bean
TransactionAttributeSource transactionAttributeSource() {
    return new AnnotationTransactionAttributeSource(false);
}
```

The *Spring TestContext Framework* supports non-private `@Transactional` test methods by default. See Transaction Management in the testing chapter for examples.

You can apply the `@Transactional` annotation to an interface definition, a method on an interface, a class definition, or a method on a class. However, the mere presence of the `@Transactional` annotation is not enough to activate the transactional behavior. The `@Transactional` annotation is merely metadata that can be consumed by some runtime infrastructure that is `@Transactional` -aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the preceding example, the `<tx:annotation-driven/>` element switches on the transactional behavior.

The Spring team recommends that you annotate only concrete classes (and methods of concrete classes) with the `@Transactional` annotation, as opposed to annotating interfaces. You certainly can place the `@Transactional` annotation on an interface (or an interface method), but this works only as you would expect it to if you use interface-based proxies. The fact that Java annotations are not inherited from interfaces means that, if you use class-based proxies (`proxy-target-class="true"`) or the weaving-based aspect (`mode="aspectj"`), the transaction settings are not recognized by the proxying and weaving infrastructure, and the object is not wrapped in a transactional proxy.

In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation (in effect, a method within the target object calling another method of the target object) does not lead to an actual transaction at runtime even if the invoked method is marked with `@Transactional`. Also, the proxy must be fully initialized to provide the expected behavior, so you should not rely on this feature in your initialization code — for example, in a `@PostConstruct` method.

Consider using AspectJ mode (see the `mode` attribute in the following table) if you expect self-invocations to be wrapped with transactions as well. In this case, there is no proxy in the first place. Instead, the target class is woven (that is, its byte code is modified) to support `@Transactional` runtime behavior on any kind of method.

Table 18. Annotation driven transaction settings

XML Attribute	Annotation Attribute	Default	Description
`transaction-manager`	N/A (see `TransactionManagementConfigurer` javadoc)	`transactionManager`	Name of the transaction manager to use. Required only if the name of the transaction manager is not `transactionManager`, as in the preceding example.

XML Attribute	Annotation Attribute	Default	Description
`mode`	`mode`	`proxy`	The default mode (`proxy`) processes annotated beans to be proxied by using Spring's AOP framework (following proxy semantics, as discussed earlier, applying to method calls coming in through the proxy only). The alternative mode (`aspectj`) instead weaves the affected classes with Spring's AspectJ transaction aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires `spring-aspects.jar` in the classpath as well as having load-time weaving (or compile-time weaving) enabled. (See Spring configuration for details on how to set up load-time weaving.)

XML Attribute	Annotation Attribute	Default	Description
`proxy-target-class`	`proxyTargetClass`	`false`	Applies to `proxy` mode only. Controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If the `proxy-target-class` attribute is set to `true`, class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, then standard JDK interface-based proxies are created. (See Proxying Mechanisms for a detailed examination of the different proxy types.)
`order`	`order`	`Ordered.LOWEST_PRECEDENCE`	Defines the order of the transaction advice that is applied to beans annotated with `@Transactional`. (For more information about the rules related to ordering of AOP advice, see Advice Ordering.) No specified ordering means that the AOP subsystem determines the order of the advice.

The default advice mode for processing `@Transactional` annotations is `proxy`, which allows for interception of calls through the proxy only. Local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to `aspectj` mode in combination with compile-time or load-time weaving.

The `proxy-target-class` attribute controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If `proxy-target-class` is set to `true`, class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, standard JDK interface-based proxies are created. (See Proxying Mechanisms for a discussion of the different proxy types.)

`@EnableTransactionManagement` and `<tx:annotation-driven/>` look for `@Transactional` only on beans in the same application context in which they are defined. This means that, if you put annotation-driven configuration in a `WebApplicationContext` for a `DispatcherServlet`, it checks for `@Transactional` beans only in your controllers and not in your services. See MVC for more information.

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the `DefaultFooService` class is annotated at the class level with the settings for a read-only transaction, but the `@Transactional` annotation on the `updateFoo(Foo)` method in the same class takes precedence over the transactional settings defined at the class level.

Java

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // ...
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // ...
    }
}
```

Kotlin

```
@Transactional(readOnly = true)
class DefaultFooService : FooService {

    override fun getFoo(fooName: String): Foo {
        // ...
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    override fun updateFoo(foo: Foo) {
        // ...
    }
}
```

`@Transactional` Settings

The `@Transactional` annotation is metadata that specifies that an interface, class, or method must have transactional semantics (for example, "start a brand new read-only transaction when this method is invoked, suspending any existing transaction"). The default `@Transactional` settings are as follows:

- The propagation setting is `PROPAGATION_REQUIRED`.
- The isolation level is `ISOLATION_DEFAULT`.
- The transaction is read-write.
- The transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.
- Any `RuntimeException` or `Error` triggers rollback, and any checked `Exception` does not.

You can change these default settings. The following table summarizes the various properties of the `@Transactional` annotation:

Table 19. @Transactional Settings

Property	Type	Description
value	`String`	Optional qualifier that specifies the transaction manager to be used.
`transactionManager`	`String`	Alias for `value`.
`label`	Array of `String` labels to add an expressive description to the transaction.	Labels may be evaluated by transaction managers to associate implementation-specific behavior with the actual transaction.
propagation	`enum`: `Propagation`	Optional propagation setting.

Property	Type	Description
`isolation`	`enum`: `Isolation`	Optional isolation level. Applies only to propagation values of `REQUIRED` or `REQUIRES_NEW`.
`timeout`	`int` (in seconds of granularity)	Optional transaction timeout. Applies only to propagation values of `REQUIRED` or `REQUIRES_NEW`.
`timeoutString`	`String` (in seconds of granularity)	Alternative for specifying the `timeout` in seconds as a `String` value — for example, as a placeholder.
`readOnly`	`boolean`	Read-write versus read-only transaction. Only applicable to values of `REQUIRED` or `REQUIRES_NEW`.
`rollbackFor`	Array of `Class` objects, which must be derived from `Throwable`.	Optional array of exception types that must cause rollback.
`rollbackForClassName`	Array of exception name patterns.	Optional array of exception name patterns that must cause rollback.
`noRollbackFor`	Array of `Class` objects, which must be derived from `Throwable`.	Optional array of exception types that must not cause rollback.
`noRollbackForClassName`	Array of exception name patterns.	Optional array of exception name patterns that must not cause rollback.

See Rollback rules for further details on rollback rule semantics, patterns, and warnings regarding possible unintentional matches for pattern-based rollback rules.

Currently, you cannot have explicit control over the name of a transaction, where 'name' means the transaction name that appears in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + `.` + the method name of the transactionally advised class. For example, if the `handlePayment(..)` method of the `BusinessService` class started a transaction, the name of the transaction would be: `com.example.BusinessService.handlePayment`.

Multiple Transaction Managers with `@Transactional`

Most Spring applications need only a single transaction manager, but there may be situations where you want multiple independent transaction managers in a single application. You can use the `value` or `transactionManager` attribute of the `@Transactional` annotation to optionally specify the

identity of the `TransactionManager` to be used. This can either be the bean name or the qualifier value of the transaction manager bean. For example, using the qualifier notation, you can combine the following Java code with the following transaction manager bean declarations in the application context:

Java

```
public class TransactionalService {

    @Transactional("order")
    public void setSomething(String name) { ... }

    @Transactional("account")
    public void doSomething() { ... }

    @Transactional("reactive-account")
    public Mono<Void> doSomethingReactive() { ... }
}
```

Kotlin

```
class TransactionalService {

    @Transactional("order")
    fun setSomething(name: String) {
        // ...
    }

    @Transactional("account")
    fun doSomething() {
        // ...
    }

    @Transactional("reactive-account")
    fun doSomethingReactive(): Mono<Void> {
        // ...
    }
}
```

The following listing shows the bean declarations:

```
<tx:annotation-driven/>

    <bean id="transactionManager1"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        ...
        <qualifier value="order"/>
    </bean>

    <bean id="transactionManager2"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        ...
        <qualifier value="account"/>
    </bean>

    <bean id="transactionManager3"
class="org.springframework.data.r2dbc.connectionfactory.R2dbcTransactionManager">
        ...
        <qualifier value="reactive-account"/>
    </bean>
```

In this case, the individual methods on `TransactionalService` run under separate transaction managers, differentiated by the `order`, `account`, and `reactive-account` qualifiers. The default `<tx:annotation-driven>` target bean name, `transactionManager`, is still used if no specifically qualified `TransactionManager` bean is found.

Custom Composed Annotations

If you find you repeatedly use the same attributes with `@Transactional` on many different methods, Spring's meta-annotation support lets you define custom composed annotations for your specific use cases. For example, consider the following annotation definitions:

Java

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional(transactionManager = "order", label = "causal-consistency")
public @interface OrderTx {
}

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional(transactionManager = "account", label = "retryable")
public @interface AccountTx {
}
```

Kotlin

```
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@Transactional(transactionManager = "order", label = ["causal-consistency"])
annotation class OrderTx

@Target(AnnotationTarget.FUNCTION, AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@Transactional(transactionManager = "account", label = ["retryable"])
annotation class AccountTx
```

The preceding annotations let us write the example from the previous section as follows:

Java

```
public class TransactionalService {

    @OrderTx
    public void setSomething(String name) {
        // ...
    }

    @AccountTx
    public void doSomething() {
        // ...
    }
}
```

Kotlin

```
class TransactionalService {

    @OrderTx
    fun setSomething(name: String) {
        // ...
    }

    @AccountTx
    fun doSomething() {
        // ...
    }
}
```

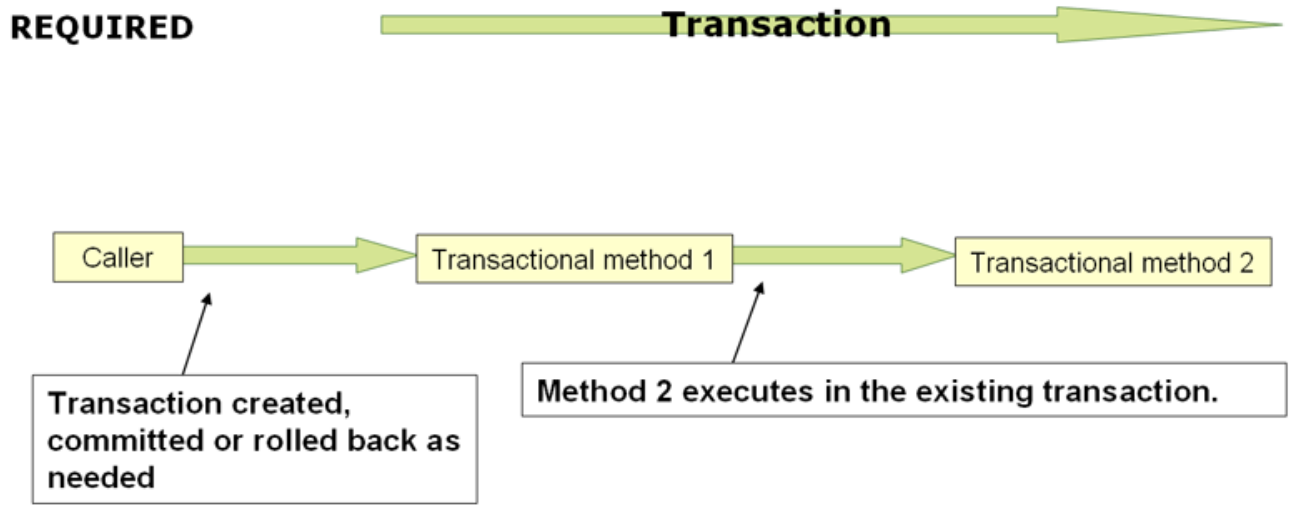
In the preceding example, we used the syntax to define the transaction manager qualifier and transactional labels, but we could also have included propagation behavior, rollback rules, timeouts, and other features.

Transaction Propagation

This section describes some semantics of transaction propagation in Spring. Note that this section is not a proper introduction to transaction propagation. Rather, it details some of the semantics regarding transaction propagation in Spring.

In Spring-managed transactions, be aware of the difference between physical and logical transactions, and how the propagation setting applies to this difference.

Understanding `PROPAGATION_REQUIRED`



`PROPAGATION_REQUIRED` enforces a physical transaction, either locally for the current scope if no transaction exists yet or participating in an existing 'outer' transaction defined for a larger scope. This is a fine default in common call stack arrangements within the same thread (for example, a service facade that delegates to several repository methods where all the underlying resources have to participate in the service-level transaction).

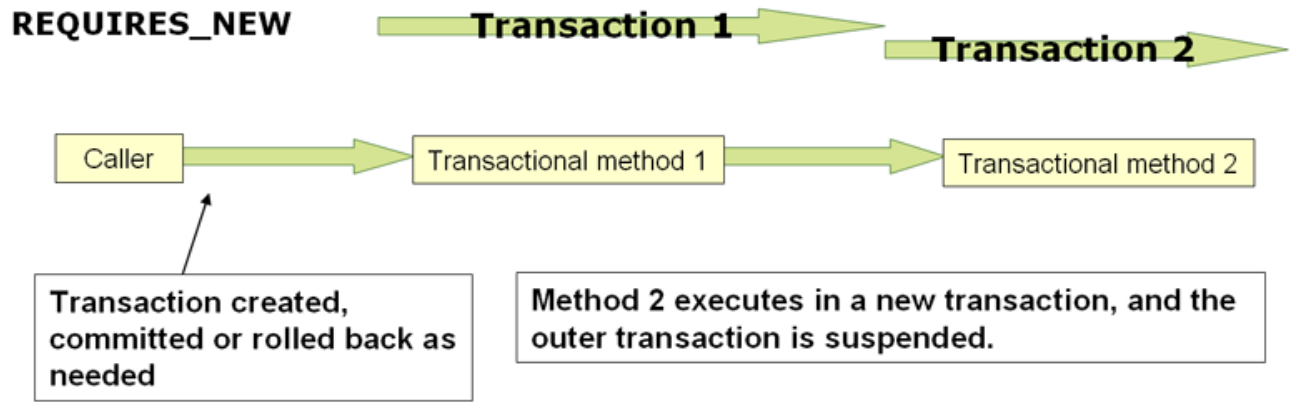
By default, a participating transaction joins the characteristics of the outer scope, silently ignoring the local isolation level, timeout value, or read-only flag (if any). Consider switching the `validateExistingTransactions` flag to `true` on your transaction manager if you want isolation level declarations to be rejected when participating in an existing transaction with a different isolation level. This non-lenient mode also rejects read-only mismatches (that is, an inner read-write transaction that tries to participate in a read-only outer scope).

When the propagation setting is `PROPAGATION_REQUIRED`, a logical transaction scope is created for each method upon which the setting is applied. Each such logical transaction scope can determine rollback-only status individually, with an outer transaction scope being logically independent from the inner transaction scope. In the case of standard `PROPAGATION_REQUIRED` behavior, all these scopes are mapped to the same physical transaction. So a rollback-only marker set in the inner transaction scope does affect the outer transaction's chance to actually commit.

However, in the case where an inner transaction scope sets the rollback-only marker, the outer transaction has not decided on the rollback itself, so the rollback (silently triggered by the inner transaction scope) is unexpected. A corresponding `UnexpectedRollbackException` is thrown at that

point. This is expected behavior so that the caller of a transaction can never be misled to assume that a commit was performed when it really was not. So, if an inner transaction (of which the outer caller is not aware) silently marks a transaction as rollback-only, the outer caller still calls commit. The outer caller needs to receive an `UnexpectedRollbackException` to indicate clearly that a rollback was performed instead.

Understanding `PROPAGATION_REQUIRES_NEW`



`PROPAGATION_REQUIRES_NEW`, in contrast to `PROPAGATION_REQUIRED`, always uses an independent physical transaction for each affected transaction scope, never participating in an existing transaction for an outer scope. In such an arrangement, the underlying resource transactions are different and, hence, can commit or roll back independently, with an outer transaction not affected by an inner transaction's rollback status and with an inner transaction's locks released immediately after its completion. Such an independent inner transaction can also declare its own isolation level, timeout, and read-only settings and not inherit an outer transaction's characteristics.

Understanding `PROPAGATION_NESTED`

`PROPAGATION_NESTED` uses a single physical transaction with multiple savepoints that it can roll back to. Such partial rollbacks let an inner transaction scope trigger a rollback for its scope, with the outer transaction being able to continue the physical transaction despite some operations having been rolled back. This setting is typically mapped onto JDBC savepoints, so it works only with JDBC resource transactions. See Spring's `DataSourceTransactionManager`.

Advising Transactional Operations

Suppose you want to run both transactional operations and some basic profiling advice. How do you effect this in the context of `<tx:annotation-driven/>`?

When you invoke the `updateFoo(Foo)` method, you want to see the following actions:

- The configured profiling aspect starts.
- The transactional advice runs.
- The method on the advised object runs.
- The transaction commits.
- The profiling aspect reports the exact duration of the whole transactional method invocation.

This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). See AOP for detailed coverage of the AOP configuration and AOP in general.

The following code shows the simple profiling aspect discussed earlier:

Java

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

    private int order;

    // allows us to control the ordering of advice
    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    // this method is the around advice
    public Object profile(ProceedingJoinPoint call) throws Throwable {
        Object returnValue;
        StopWatch clock = new StopWatch(getClass().getName());
        try {
            clock.start(call.toShortString());
            returnValue = call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
        return returnValue;
    }
}
```

Kotlin

```
class SimpleProfiler : Ordered {

    private var order: Int = 0

    // allows us to control the ordering of advice
    override fun getOrder(): Int {
        return this.order
    }

    fun setOrder(order: Int) {
        this.order = order
    }

    // this method is the around advice
    fun profile(call: ProceedingJoinPoint): Any {
        var returnValue: Any
        val clock = StopWatch(javaClass.name)
        try {
            clock.start(call.toShortString())
            returnValue = call.proceed()
        } finally {
            clock.stop()
            println(clock.prettyPrint())
        }
        return returnValue
    }
}
```

The ordering of advice is controlled through the `Ordered` interface. For full details on advice ordering, see Advice ordering.

The following configuration creates a `fooService` bean that has profiling and transactional aspects applied to it in the desired order:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this is the aspect -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- run before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <tx:annotation-driven transaction-manager="txManager" order="200"/>

    <aop:config>
            <!-- this advice runs around the transactional advice -->
            <aop:aspect id="profilingAspect" ref="profiler">
                <aop:pointcut id="serviceMethodWithReturnValue"
                        expression="execution(!void x.y..*Service.*(..))"/>
                <aop:around method="profile" pointcut-
ref="serviceMethodWithReturnValue"/>
            </aop:aspect>
    </aop:config>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>

    <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

You can configure any number of additional aspects in similar fashion.

The following example creates the same setup as the previous two examples but uses the purely XML declarative approach:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the profiling advice -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- run before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <aop:config>
        <aop:pointcut id="entryPointMethod" expression="execution(*
x.y..*Service.*(..))"/>
        <!-- runs after the profiling advice (cf. the order attribute) -->

        <aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod" order="2"/>
        <!-- order value is higher than the profiling aspect -->

        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                    expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>

    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other <bean/> definitions such as a DataSource and a TransactionManager here
-->

</beans>
```

The result of the preceding configuration is a `fooService` bean that has profiling and transactional

aspects applied to it in that order. If you want the profiling advice to run after the transactional advice on the way in and before the transactional advice on the way out, you can swap the value of the profiling aspect bean's `order` property so that it is higher than the transactional advice's order value.

You can configure additional aspects in similar fashion.

Using `@Transactional` with AspectJ

You can also use the Spring Framework's `@Transactional` support outside of a Spring container by means of an AspectJ aspect. To do so, first annotate your classes (and optionally your classes' methods) with the `@Transactional` annotation, and then link (weave) your application with the `org.springframework.transaction.aspectj.AnnotationTransactionAspect` defined in the `spring-aspects.jar` file. You must also configure the aspect with a transaction manager. You can use the Spring Framework's IoC container to take care of dependency-injecting the aspect. The simplest way to configure the transaction management aspect is to use the `<tx:annotation-driven/>` element and specify the `mode` attribute to `aspectj` as described in Using `@Transactional`. Because we focus here on applications that run outside of a Spring container, we show you how to do it programmatically.

Prior to continuing, you may want to read Using `@Transactional` and AOP respectively.

The following example shows how to create a transaction manager and configure the `AnnotationTransactionAspect` to use it:

Java

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new
DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before
executing any transactional methods
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

Kotlin

```
// construct an appropriate transaction manager
val txManager = DataSourceTransactionManager(getDataSource())

// configure the AnnotationTransactionAspect to use it; this must be done before
executing any transactional methods
AnnotationTransactionAspect.aspectOf().transactionManager = txManager
```

When you use this aspect, you must annotate the implementation class (or the methods within that class or both), not the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are not inherited.

The `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any public method in the class.

The `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). You can annotate any method, regardless of visibility.

To weave your applications with the `AnnotationTransactionAspect`, you must either build your application with AspectJ (see the AspectJ Development Guide) or use load-time weaving. See Load-time weaving with AspectJ in the Spring Framework for a discussion of load-time weaving with AspectJ.

4.1.5. Programmatic Transaction Management

The Spring Framework provides two means of programmatic transaction management, by using:

- The `TransactionTemplate` or `TransactionalOperator`.
- A `TransactionManager` implementation directly.

The Spring team generally recommends the `TransactionTemplate` for programmatic transaction management in imperative flows and `TransactionalOperator` for reactive code. The second approach is similar to using the JTA `UserTransaction` API, although exception handling is less cumbersome.

Using the `TransactionTemplate`

The `TransactionTemplate` adopts the same approach as other Spring templates, such as the `JdbcTemplate`. It uses a callback approach (to free application code from having to do the boilerplate acquisition and release transactional resources) and results in code that is intention driven, in that your code focuses solely on what you want to do.

As the examples that follow show, using the `TransactionTemplate` absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you have to make yourself.

Application code that must run in a transactional context and that explicitly uses the `TransactionTemplate` resembles the next example. You, as an application developer, can write a `TransactionCallback` implementation (typically expressed as an anonymous inner class) that contains the code that you need to run in the context of a transaction. You can then pass an instance of your custom `TransactionCallback` to the `execute(..)` method exposed on the `TransactionTemplate`. The following example shows how to do so:

Java

```
public class SimpleService implements Service {

    // single TransactionTemplate shared amongst all methods in this instance
    private final TransactionTemplate transactionTemplate;

    // use constructor-injection to supply the PlatformTransactionManager
    public SimpleService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {
        return transactionTemplate.execute(new TransactionCallback() {
            // the code in this method runs in a transactional context
            public Object doInTransaction(TransactionStatus status) {
                updateOperation1();
                return resultOfUpdateOperation2();
            }
        });
    }
}
```

Kotlin

```
// use constructor-injection to supply the PlatformTransactionManager
class SimpleService(transactionManager: PlatformTransactionManager) : Service {

    // single TransactionTemplate shared amongst all methods in this instance
    private val transactionTemplate = TransactionTemplate(transactionManager)

    fun someServiceMethod() = transactionTemplate.execute<Any?> {
        updateOperation1()
        resultOfUpdateOperation2()
    }
}
```

If there is no return value, you can use the convenient `TransactionCallbackWithoutResult` class with an anonymous class, as follows:

Java

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Kotlin

```
transactionTemplate.execute(object : TransactionCallbackWithoutResult() {
    override fun doInTransactionWithoutResult(status: TransactionStatus) {
        updateOperation1()
        updateOperation2()
    }
})
```

Code within the callback can roll the transaction back by calling the `setRollbackOnly()` method on the supplied `TransactionStatus` object, as follows:

Java

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
        } catch (SomeBusinessException ex) {
            status.setRollbackOnly();
        }
    }
});
```

Kotlin

```
transactionTemplate.execute(object : TransactionCallbackWithoutResult() {

    override fun doInTransactionWithoutResult(status: TransactionStatus) {
        try {
            updateOperation1()
            updateOperation2()
        } catch (ex: SomeBusinessException) {
            status.setRollbackOnly()
        }
    }
})
```

Specifying Transaction Settings

You can specify transaction settings (such as the propagation mode, the isolation level, the timeout, and so forth) on the `TransactionTemplate` either programmatically or in configuration. By default, `TransactionTemplate` instances have the default transactional settings. The following example shows the programmatic customization of the transactional settings for a specific `TransactionTemplate:`

Java

```
public class SimpleService implements Service {

    private final TransactionTemplate transactionTemplate;

    public SimpleService(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);

        // the transaction settings can be set here explicitly if so desired

this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_UNCOMM
ITTED);
        this.transactionTemplate.setTimeout(30); // 30 seconds
        // and so forth...
    }
}
```

Kotlin

```
class SimpleService(transactionManager: PlatformTransactionManager) : Service {

    private val transactionTemplate = TransactionTemplate(transactionManager).apply {
        // the transaction settings can be set here explicitly if so desired
        isolationLevel = TransactionDefinition.ISOLATION_READ_UNCOMMITTED
        timeout = 30 // 30 seconds
        // and so forth...
    }
}
```

The following example defines a `TransactionTemplate` with some custom transactional settings by using Spring XML configuration:

```
<bean id="sharedTransactionTemplate"
        class="org.springframework.transaction.support.TransactionTemplate">
    <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>
    <property name="timeout" value="30"/>
</bean>
```

You can then inject the `sharedTransactionTemplate` into as many services as are required.

Finally, instances of the `TransactionTemplate` class are thread-safe, in that instances do not maintain any conversational state. `TransactionTemplate` instances do, however, maintain configuration state. So, while a number of classes may share a single instance of a `TransactionTemplate`, if a class needs to use a `TransactionTemplate` with different settings (for example, a different isolation level), you need to create two distinct `TransactionTemplate` instances.

Using the `TransactionalOperator`

The `TransactionalOperator` follows an operator design that is similar to other reactive operators. It uses a callback approach (to free application code from having to do the boilerplate acquisition and release transactional resources) and results in code that is intention driven, in that your code focuses solely on what you want to do.

As the examples that follow show, using the `TransactionalOperator` absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you have to make yourself.

Application code that must run in a transactional context and that explicitly uses the `TransactionalOperator` resembles the next example:

Java

```
public class SimpleService implements Service {

    // single TransactionalOperator shared amongst all methods in this instance
    private final TransactionalOperator transactionalOperator;

    // use constructor-injection to supply the ReactiveTransactionManager
    public SimpleService(ReactiveTransactionManager transactionManager) {
        this.transactionalOperator = TransactionalOperator.create(transactionManager);
    }

    public Mono<Object> someServiceMethod() {

        // the code in this method runs in a transactional context

        Mono<Object> update = updateOperation1();

        return
update.then(resultOfUpdateOperation2).as(transactionalOperator::transactional);
    }
}
```

Kotlin

```
// use constructor-injection to supply the ReactiveTransactionManager
class SimpleService(transactionManager: ReactiveTransactionManager) : Service {

    // single TransactionalOperator shared amongst all methods in this instance
    private val transactionalOperator =
TransactionalOperator.create(transactionManager)

    suspend fun someServiceMethod() = transactionalOperator.executeAndAwait<Any?> {
        updateOperation1()
        resultOfUpdateOperation2()
    }
}
```

`TransactionalOperator` can be used in two ways:

- Operator-style using Project Reactor types (`mono.as(transactionalOperator::transactional)`)
- Callback-style for every other case (`transactionalOperator.execute(TransactionCallback<T>)`)

Code within the callback can roll the transaction back by calling the `setRollbackOnly()` method on the supplied `ReactiveTransaction` object, as follows:

Java

```
transactionalOperator.execute(new TransactionCallback<>() {

    public Mono<Object> doInTransaction(ReactiveTransaction status) {
        return updateOperation1().then(updateOperation2)
                    .doOnError(SomeBusinessException.class, e ->
status.setRollbackOnly());
        }
    }
});
```

Kotlin

```
transactionalOperator.execute(object : TransactionCallback() {

    override fun doInTransactionWithoutResult(status: ReactiveTransaction) {
        updateOperation1().then(updateOperation2)
                    .doOnError(SomeBusinessException.class, e ->
status.setRollbackOnly())
    }
})
```

Cancel Signals

In Reactive Streams, a `Subscriber` can cancel its `Subscription` and stop its `Publisher`. Operators in

Project Reactor, as well as in other libraries, such as `next()`, `take(long)`, `timeout(Duration)`, and others can issue cancellations. There is no way to know the reason for the cancellation, whether it is due to an error or a simply lack of interest to consume further. Since version 5.3 cancel signals lead to a roll back. As a result it is important to consider the operators used downstream from a transaction `Publisher`. In particular in the case of a `Flux` or other multi-value `Publisher`, the full output must be consumed to allow the transaction to complete.

Specifying Transaction Settings

You can specify transaction settings (such as the propagation mode, the isolation level, the timeout, and so forth) for the `TransactionalOperator`. By default, `TransactionalOperator` instances have default transactional settings. The following example shows customization of the transactional settings for a specific `TransactionalOperator:`

Java

```
public class SimpleService implements Service {

    private final TransactionalOperator transactionalOperator;

    public SimpleService(ReactiveTransactionManager transactionManager) {
        DefaultTransactionDefinition definition = new DefaultTransactionDefinition();

        // the transaction settings can be set here explicitly if so desired

definition.setIsolationLevel(TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
        definition.setTimeout(30); // 30 seconds
        // and so forth...

        this.transactionalOperator = TransactionalOperator.create(transactionManager,
definition);
    }
}
```

Kotlin

```
class SimpleService(transactionManager: ReactiveTransactionManager) : Service {

    private val definition = DefaultTransactionDefinition().apply {
        // the transaction settings can be set here explicitly if so desired
        isolationLevel = TransactionDefinition.ISOLATION_READ_UNCOMMITTED
        timeout = 30 // 30 seconds
        // and so forth...
    }
    private val transactionalOperator = TransactionalOperator(transactionManager,
definition)
}
```

Using the `TransactionManager`

The following sections explain programmatic usage of imperative and reactive transaction managers.

Using the `PlatformTransactionManager`

For imperative transactions, you can use a `org.springframework.transaction.PlatformTransactionManager` directly to manage your transaction. To do so, pass the implementation of the `PlatformTransactionManager` you use to your bean through a bean reference. Then, by using the `TransactionDefinition` and `TransactionStatus` objects, you can initiate transactions, roll back, and commit. The following example shows how to do so:

Java

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can be done only
programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // put your business logic here
} catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

Kotlin

```
val def = DefaultTransactionDefinition()
// explicitly setting the transaction name is something that can be done only
programmatically
def.setName("SomeTxName")
def.propagationBehavior = TransactionDefinition.PROPAGATION_REQUIRED

val status = txManager.getTransaction(def)
try {
    // put your business logic here
} catch (ex: MyException) {
    txManager.rollback(status)
    throw ex
}

txManager.commit(status)
```

Using the `ReactiveTransactionManager`

When working with reactive transactions, you can use a `org.springframework.transaction.ReactiveTransactionManager` directly to manage your transaction. To do so, pass the implementation of the `ReactiveTransactionManager` you use to your bean through a bean reference. Then, by using the `TransactionDefinition` and `ReactiveTransaction` objects, you can initiate transactions, roll back, and commit. The following example shows how to do so:

Java

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can be done only
programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

Mono<ReactiveTransaction> reactiveTx = txManager.getReactiveTransaction(def);

reactiveTx.flatMap(status -> {

    Mono<Object> tx = ...; // put your business logic here

    return tx.then(txManager.commit(status))
            .onErrorResume(ex -> txManager.rollback(status).then(Mono.error(ex)));
});
```

Kotlin

```
val def = DefaultTransactionDefinition()
// explicitly setting the transaction name is something that can be done only
programmatically
def.setName("SomeTxName")
def.propagationBehavior = TransactionDefinition.PROPAGATION_REQUIRED

val reactiveTx = txManager.getReactiveTransaction(def)
reactiveTx.flatMap { status ->

    val tx = ... // put your business logic here

    tx.then(txManager.commit(status))
            .onErrorResume { ex -> txManager.rollback(status).then(Mono.error(ex)) }
}
```

4.1.6. Choosing Between Programmatic and Declarative Transaction Management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that requires transactions only for certain update operations, you may not want to set up transactional proxies by using Spring or

any other technology. In this case, using the `TransactionTemplate` may be a good approach. Being able to set the transaction name explicitly is also something that can be done only by using the programmatic approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic and is not difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

4.1.7. Transaction-bound Events

As of Spring 4.2, the listener of an event can be bound to a phase of the transaction. The typical example is to handle the event when the transaction has completed successfully. Doing so lets events be used with more flexibility when the outcome of the current transaction actually matters to the listener.

You can register a regular event listener by using the `@EventListener` annotation. If you need to bind it to the transaction, use `@TransactionalEventListener`. When you do so, the listener is bound to the commit phase of the transaction by default.

The next example shows this concept. Assume that a component publishes an order-created event and that we want to define a listener that should only handle that event once the transaction in which it has been published has committed successfully. The following example sets up such an event listener:

Java

```
@Component
public class MyComponent {

    @TransactionalEventListener
    public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {
        // ...
    }
}
```

Kotlin

```
@Component
class MyComponent {

    @TransactionalEventListener
    fun handleOrderCreatedEvent(creationEvent: CreationEvent<Order>) {
        // ...
    }
}
```

The `@TransactionalEventListener` annotation exposes a `phase` attribute that lets you customize the phase of the transaction to which the listener should be bound. The valid phases are `BEFORE_COMMIT`,

`AFTER_COMMIT` (default), `AFTER_ROLLBACK`, as well as `AFTER_COMPLETION` which aggregates the transaction completion (be it a commit or a rollback).

If no transaction is running, the listener is not invoked at all, since we cannot honor the required semantics. You can, however, override that behavior by setting the `fallbackExecution` attribute of the annotation to `true`.

`@TransactionalEventListener` only works with thread-bound transactions managed by `PlatformTransactionManager`. A reactive transaction managed by `ReactiveTransactionManager` uses the Reactor context instead of thread-local attributes, so from the perspective of an event listener, there is no compatible active transaction that it can participate in.

4.1.8. Application server-specific integration

Spring's transaction abstraction is generally application server-agnostic. Additionally, Spring's `JtaTransactionManager` class (which can optionally perform a JNDI lookup for the JTA `UserTransaction` and `TransactionManager` objects) autodetects the location for the latter object, which varies by application server. Having access to the JTA `TransactionManager` allows for enhanced transaction semantics — in particular, supporting transaction suspension. See the `JtaTransactionManager` javadoc for details.

Spring's `JtaTransactionManager` is the standard choice to run on Jakarta EE application servers and is known to work on all common servers. Advanced functionality, such as transaction suspension, works on many servers as well (including GlassFish, JBoss and Geronimo) without any special configuration required. However, for fully supported transaction suspension and further advanced integration, Spring includes special adapters for WebLogic Server and WebSphere. These adapters are discussed in the following sections.

For standard scenarios, including WebLogic Server and WebSphere, consider using the convenient `<tx:jta-transaction-manager/>` configuration element. When configured, this element automatically detects the underlying server and chooses the best transaction manager available for the platform. This means that you need not explicitly configure server-specific adapter classes (as discussed in the following sections). Rather, they are chosen automatically, with the standard `JtaTransactionManager` as the default fallback.

IBM WebSphere

On WebSphere 6.1.0.9 and above, the recommended Spring JTA transaction manager to use is `WebSphereUowTransactionManager`. This special adapter uses IBM's `UOWManager` API, which is available in WebSphere Application Server 6.1.0.9 and later. With this adapter, Spring-driven transaction suspension (suspend and resume as initiated by `PROPAGATION_REQUIRES_NEW`) is officially supported by IBM.

Oracle WebLogic Server

On WebLogic Server 9.0 or above, you would typically use the `WebLogicJtaTransactionManager` instead of the stock `JtaTransactionManager` class. This special WebLogic-specific subclass of the normal `JtaTransactionManager` supports the full power of Spring's transaction definitions in a

WebLogic-managed transaction environment, beyond standard JTA semantics. Features include transaction names, per-transaction isolation levels, and proper resuming of transactions in all cases.

4.1.9. Solutions to Common Problems

This section describes solutions to some common problems.

Using the Wrong Transaction Manager for a Specific `DataSource`

Use the correct `PlatformTransactionManager` implementation based on your choice of transactional technologies and requirements. Used properly, the Spring Framework merely provides a straightforward and portable abstraction. If you use global transactions, you must use the `org.springframework.transaction.jta.JtaTransactionManager` class (or an application server-specific subclass of it) for all your transactional operations. Otherwise, the transaction infrastructure tries to perform local transactions on such resources as container `DataSource` instances. Such local transactions do not make sense, and a good application server treats them as errors.

4.1.10. Further Resources

For more information about the Spring Framework's transaction support, see:

- Distributed transactions in Spring, with and without XA is a JavaWorld presentation in which Spring's David Syer guides you through seven patterns for distributed transactions in Spring applications, three of them with XA and four without.
- *Java Transaction Design Strategies* is a book available from InfoQ that provides a well-paced introduction to transactions in Java. It also includes side-by-side examples of how to configure and use transactions with both the Spring Framework and EJB3.

4.2. DAO Support

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies (such as JDBC, Hibernate, or JPA) in a consistent way. This lets you switch between the aforementioned persistence technologies fairly easily, and it also lets you code without worrying about catching exceptions that are specific to each technology.

4.2.1. Consistent Exception Hierarchy

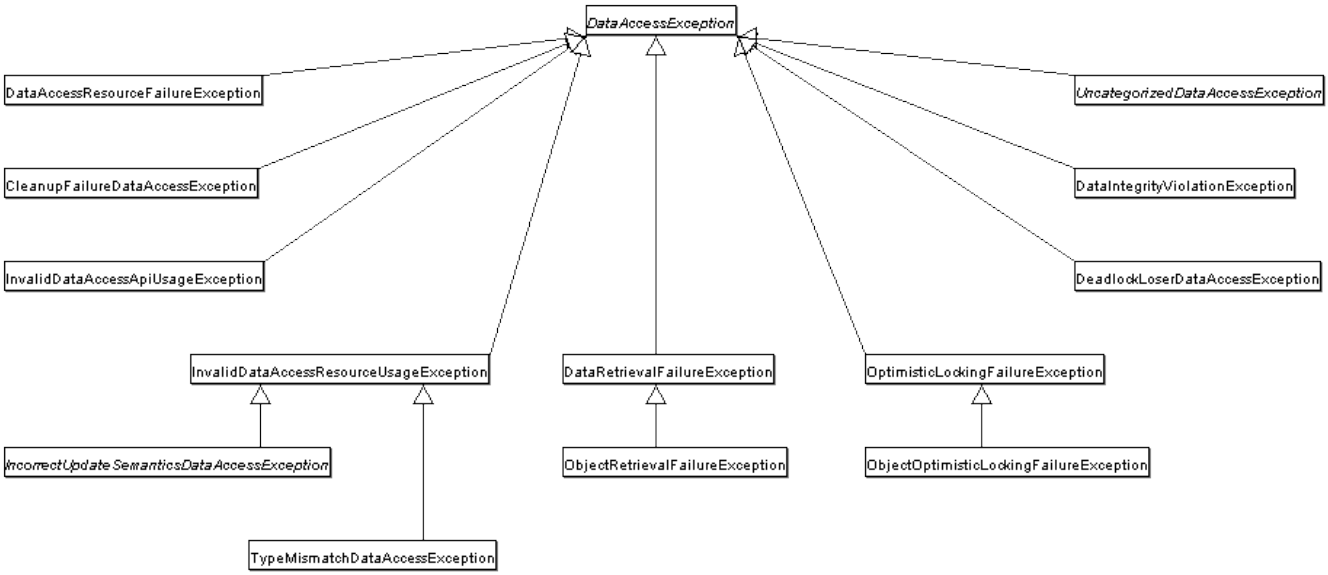
Spring provides a convenient translation from technology-specific exceptions, such as `SQLException` to its own exception class hierarchy, which has `DataAccessException` as the root exception. These exceptions wrap the original exception so that there is never any risk that you might lose any information about what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap JPA- and Hibernate-specific exceptions, converting them to a set of focused runtime exceptions. This lets you handle most non-recoverable persistence exceptions in only the appropriate layers, without having annoying boilerplate catch-and-throw blocks and exception declarations in your DAOs. (You can still trap and handle exceptions anywhere you need to though.) As mentioned above, JDBC exceptions (including

database-specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

The preceding discussion holds true for the various template classes in Spring's support for various ORM frameworks. If you use the interceptor-based classes, the application must care about handling `HibernateExceptions` and `PersistenceExceptions` itself, preferably by delegating to the `convertHibernateAccessException(..)` or `convertJpaAccessException(..)` methods, respectively, of `SessionFactoryUtils`. These methods convert the exceptions to exceptions that are compatible with the exceptions in the `org.springframework.dao` exception hierarchy. As `PersistenceExceptions` are unchecked, they can get thrown, too (sacrificing generic DAO abstraction in terms of exceptions, though).

The following image shows the exception hierarchy that Spring provides. (Note that the class hierarchy detailed in the image shows only a subset of the entire `DataAccessException` hierarchy.)



4.2.2. Annotations Used to Configure DAO or Repository Classes

The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception translation is to use the `@Repository` annotation. This annotation also lets the component scanning support find and configure your DAOs and repositories without having to provide XML configuration entries for them. The following example shows how to use the `@Repository` annotation:

Java

```
@Repository ①
public class SomeMovieFinder implements MovieFinder {
    // ...
}
```

① The `@Repository` annotation.

Kotlin

```
@Repository ①
class SomeMovieFinder : MovieFinder {
    // ...
}
```

① The `@Repository` annotation.

Any DAO or repository implementation needs access to a persistence resource, depending on the persistence technology used. For example, a JDBC-based repository needs access to a JDBC `DataSource`, and a JPA-based repository needs access to an `EntityManager`. The easiest way to accomplish this is to have this resource dependency injected by using one of the `@Autowired`, `@Inject`, `@Resource` or `@PersistenceContext` annotations. The following example works for a JPA repository:

Java

```
@Repository
public class JpaMovieFinder implements MovieFinder {

    @PersistenceContext
    private EntityManager entityManager;

    // ...
}
```

Kotlin

```
@Repository
class JpaMovieFinder : MovieFinder {

    @PersistenceContext
    private lateinit var entityManager: EntityManager

    // ...
}
```

If you use the classic Hibernate APIs, you can inject `SessionFactory`, as the following example shows:

Java

```
@Repository
public class HibernateMovieFinder implements MovieFinder {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    // ...
}
```

Kotlin

```
@Repository
class HibernateMovieFinder(private val sessionFactory: SessionFactory) : MovieFinder {
    // ...
}
```

The last example we show here is for typical JDBC support. You could have the `DataSource` injected into an initialization method or a constructor, where you would create a `JdbcTemplate` and other data access support classes (such as `SimpleJdbcCall` and others) by using this `DataSource`. The following example autowires a `DataSource`:

Java

```
@Repository
public class JdbcMovieFinder implements MovieFinder {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public void init(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // ...
}
```

Kotlin

```
@Repository
class JdbcMovieFinder(dataSource: DataSource) : MovieFinder {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    // ...
}
```

See the specific coverage of each persistence technology for details on how to configure the application context to take advantage of these annotations.

4.3. Data Access with JDBC

The value provided by the Spring Framework JDBC abstraction is perhaps best shown by the sequence of actions outlined in the following table below. The table shows which actions Spring takes care of and which actions are your responsibility.

Table 20. Spring JDBC - who does what?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and run the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, the statement, and the resultset.	X	

The Spring Framework takes care of all the low-level details that can make JDBC such a tedious API.

4.3.1. Choosing an Approach for JDBC Database Access

You can choose among several approaches to form the basis for your JDBC database access. In addition to three flavors of `JdbcTemplate`, a new `SimpleJdbcInsert` and `SimpleJdbcCall` approach optimizes database metadata, and the RDBMS Object style takes a more object-oriented approach similar to that of JDO Query design. Once you start using one of these approaches, you can still mix

and match to include a feature from a different approach. All approaches require a JDBC 2.0-compliant driver, and some advanced features require a JDBC 3.0 driver.

- `JdbcTemplate` is the classic and most popular Spring JDBC approach. This "lowest-level" approach and all others use a JdbcTemplate under the covers.
- `NamedParameterJdbcTemplate` wraps a `JdbcTemplate` to provide named parameters instead of the traditional JDBC `?` placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.
- `SimpleJdbcInsert` and `SimpleJdbcCall` optimize database metadata to limit the amount of necessary configuration. This approach simplifies coding so that you need to provide only the name of the table or procedure and provide a map of parameters matching the column names. This works only if the database provides adequate metadata. If the database does not provide this metadata, you have to provide explicit configuration of the parameters.
- RDBMS objects — including `MappingSqlQuery`, `SqlUpdate`, and `StoredProcedure` — require you to create reusable and thread-safe objects during initialization of your data-access layer. This approach is modeled after JDO Query, wherein you define your query string, declare parameters, and compile the query. Once you do that, `execute(…)`, `update(…)`, and `findObject(…)` methods can be called multiple times with various parameter values.

4.3.2. Package Hierarchy

The Spring Framework's JDBC abstraction framework consists of four different packages:

- `core`: The `org.springframework.jdbc.core` package contains the `JdbcTemplate` class and its various callback interfaces, plus a variety of related classes. A subpackage named `org.springframework.jdbc.core.simple` contains the `SimpleJdbcInsert` and `SimpleJdbcCall` classes. Another subpackage named `org.springframework.jdbc.core.namedparam` contains the `NamedParameterJdbcTemplate` class and the related support classes. See Using the JDBC Core Classes to Control Basic JDBC Processing and Error Handling, JDBC Batch Operations, and Simplifying JDBC Operations with the `SimpleJdbc` Classes.
- `datasource`: The `org.springframework.jdbc.datasource` package contains a utility class for easy `DataSource` access and various simple `DataSource` implementations that you can use for testing and running unmodified JDBC code outside of a Jakarta EE container. A subpackage named `org.springfamework.jdbc.datasource.embedded` provides support for creating embedded databases by using Java database engines, such as HSQL, H2, and Derby. See Controlling Database Connections and Embedded Database Support.
- `object`: The `org.springframework.jdbc.object` package contains classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects. See Modeling JDBC Operations as Java Objects. This approach is modeled by JDO, although objects returned by queries are naturally disconnected from the database. This higher-level of JDBC abstraction depends on the lower-level abstraction in the `org.springframework.jdbc.core` package.
- `support`: The `org.springframework.jdbc.support` package provides `SQLException` translation functionality and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the `org.springframework.dao` package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked, which gives you the option of catching the

exceptions from which you can recover while letting other exceptions be propagated to the caller. See Using `SQLExceptionTranslator`.

4.3.3. Using the JDBC Core Classes to Control Basic JDBC Processing and Error Handling

This section covers how to use the JDBC core classes to control basic JDBC processing, including error handling. It includes the following topics:

- Using `JdbcTemplate`
- Using `NamedParameterJdbcTemplate`
- Using `SQLExceptionTranslator`
- Running Statements
- Running Queries
- Updating the Database
- Retrieving Auto-generated Keys

Using `JdbcTemplate`

`JdbcTemplate` is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors, such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow (such as statement creation and execution), leaving application code to provide SQL and extract results. The `JdbcTemplate` class:

- Runs SQL queries
- Updates statements and stored procedure calls
- Performs iteration over `ResultSet` instances and extraction of returned parameter values.
- Catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package. (See Consistent Exception Hierarchy.)

When you use the `JdbcTemplate` for your code, you need only to implement callback interfaces, giving them a clearly defined contract. Given a `Connection` provided by the `JdbcTemplate` class, the `PreparedStatementCreator` callback interface creates a prepared statement, providing SQL and any necessary parameters. The same is true for the `CallableStatementCreator` interface, which creates callable statements. The `RowCallbackHandler` interface extracts values from each row of a `ResultSet`.

You can use `JdbcTemplate` within a DAO implementation through direct instantiation with a `DataSource` reference, or you can configure it in a Spring IoC container and give it to DAOs as a bean reference.

The `DataSource` should always be configured as a bean in the Spring IoC container. In the first case the bean is given to the service directly; in the second case it is given to the prepared template.

All SQL issued by this class is logged at the `DEBUG` level under the category corresponding to the fully

qualified class name of the template instance (typically `JdbcTemplate`, but it may be different if you use a custom subclass of the `JdbcTemplate` class).

The following sections provide some examples of `JdbcTemplate` usage. These examples are not an exhaustive list of all of the functionality exposed by the `JdbcTemplate`. See the attendant javadoc for that.

Querying (`SELECT`)

The following query gets the number of rows in a relation:

Java

```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor",
Integer.class);
```

Kotlin

```
val rowCount = jdbcTemplate.queryForObject<Int>("select count(*) from t_actor")!!
```

The following query uses a bind variable:

Java

```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
        "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```

Kotlin

```
val countOfActorsNamedJoe = jdbcTemplate.queryForObject<Int>(
        "select count(*) from t_actor where first_name = ?", arrayOf("Joe"))!!
```

The following query looks for a `String`:

Java

```
String lastName = this.jdbcTemplate.queryForObject(
        "select last_name from t_actor where id = ?",
        String.class, 1212L);
```

Kotlin

```
val lastName = this.jdbcTemplate.queryForObject<String>(
        "select last_name from t_actor where id = ?",
        arrayOf(1212L))!!
```

The following query finds and populates a single domain object:

Java

```
Actor actor = jdbcTemplate.queryForObject(
        "select first_name, last_name from t_actor where id = ?",
        (resultSet, rowNum) -> {
            Actor newActor = new Actor();
            newActor.setFirstName(resultSet.getString("first_name"));
            newActor.setLastName(resultSet.getString("last_name"));
            return newActor;
        },
        1212L);
```

Kotlin

```
val actor = jdbcTemplate.queryForObject(
            "select first_name, last_name from t_actor where id = ?",
            arrayOf(1212L)) { rs, _ ->
        Actor(rs.getString("first_name"), rs.getString("last_name"))
    }
```

The following query finds and populates a list of domain objects:

Java

```
List<Actor> actors = this.jdbcTemplate.query(
        "select first_name, last_name from t_actor",
        (resultSet, rowNum) -> {
            Actor actor = new Actor();
            actor.setFirstName(resultSet.getString("first_name"));
            actor.setLastName(resultSet.getString("last_name"));
            return actor;
        });
```

Kotlin

```
val actors = jdbcTemplate.query("select first_name, last_name from t_actor") { rs, _
->
        Actor(rs.getString("first_name"), rs.getString("last_name"))
```

If the last two snippets of code actually existed in the same application, it would make sense to remove the duplication present in the two `RowMapper` lambda expressions and extract them out into a single field that could then be referenced by DAO methods as needed. For example, it may be better to write the preceding code snippet as follows:

Java

```
private final RowMapper<Actor> actorRowMapper = (resultSet, rowNum) -> {
    Actor actor = new Actor();
    actor.setFirstName(resultSet.getString("first_name"));
    actor.setLastName(resultSet.getString("last_name"));
    return actor;
};

public List<Actor> findAllActors() {
    return this.jdbcTemplate.query("select first_name, last_name from t_actor",
actorRowMapper);
}
```

Kotlin

```
val actorMapper = RowMapper<Actor> { rs: ResultSet, rowNum: Int ->
    Actor(rs.getString("first_name"), rs.getString("last_name"))
}

fun findAllActors(): List<Actor> {
    return jdbcTemplate.query("select first_name, last_name from t_actor",
actorMapper)
}
```

Updating (`INSERT`, `UPDATE`, and `DELETE`) with `JdbcTemplate`

You can use the `update(..)` method to perform insert, update, and delete operations. Parameter values are usually provided as variable arguments or, alternatively, as an object array.

The following example inserts a new entry:

Java

```
this.jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling");
```

Kotlin

```
jdbcTemplate.update(
        "insert into t_actor (first_name, last_name) values (?, ?)",
        "Leonor", "Watling")
```

The following example updates an existing entry:

Java

```
this.jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L);
```

Kotlin

```
jdbcTemplate.update(
        "update t_actor set last_name = ? where id = ?",
        "Banjo", 5276L)
```

The following example deletes an entry:

Java

```
this.jdbcTemplate.update(
        "delete from t_actor where id = ?",
        Long.valueOf(actorId));
```

Kotlin

```
jdbcTemplate.update("delete from t_actor where id = ?", actorId.toLong())
```

Other `JdbcTemplate` Operations

You can use the `execute(..)` method to run any arbitrary SQL. Consequently, the method is often used for DDL statements. It is heavily overloaded with variants that take callback interfaces, binding variable arrays, and so on. The following example creates a table:

Java

```
this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

Kotlin

```
jdbcTemplate.execute("create table mytable (id integer, name varchar(100))")
```

The following example invokes a stored procedure:

Java

```
this.jdbcTemplate.update(
        "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
        Long.valueOf(unionId));
```

Kotlin

```
jdbcTemplate.update(
        "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
        unionId.toLong())
```

More sophisticated stored procedure support is covered later.

`JdbcTemplate` Best Practices

Instances of the `JdbcTemplate` class are thread-safe, once configured. This is important because it means that you can configure a single instance of a `JdbcTemplate` and then safely inject this shared reference into multiple DAOs (or repositories). The `JdbcTemplate` is stateful, in that it maintains a reference to a `DataSource`, but this state is not conversational state.

A common practice when using the `JdbcTemplate` class (and the associated `NamedParameterJdbcTemplate` class) is to configure a `DataSource` in your Spring configuration file and then dependency-inject that shared `DataSource` bean into your DAO classes. The `JdbcTemplate` is created in the setter for the `DataSource`. This leads to DAOs that resemble the following:

Java

```
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

Kotlin

```
class JdbcCorporateEventDao(dataSource: DataSource) : CorporateEventDao {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

The following example shows the corresponding XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

An alternative to explicit configuration is to use component-scanning and annotation support for dependency injection. In this case, you can annotate the class with `@Repository` (which makes it a candidate for component-scanning) and annotate the `DataSource` setter method with `@Autowired`. The following example shows how to do so:

Java

```
@Repository ①
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    @Autowired ②
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource); ③
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

① Annotate the class with `@Repository`.

② Annotate the `DataSource` setter method with `@Autowired`.

③ Create a new `JdbcTemplate` with the `DataSource`.

Kotlin

```
@Repository ①
class JdbcCorporateEventDao(dataSource: DataSource) : CorporateEventDao { ②

    private val jdbcTemplate = JdbcTemplate(dataSource) ③

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

① Annotate the class with `@Repository`.

② Constructor injection of the `DataSource`.

③ Create a new `JdbcTemplate` with the `DataSource`.

The following example shows the corresponding XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <!-- Scans within the base package of the application for @Component classes to
configure as beans -->
    <context:component-scan base-package="org.springframework.docs.test" />

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="jdbc.properties"/>

</beans>
```

If you use Spring's `JdbcDaoSupport` class and your various JDBC-backed DAO classes extend from it, your sub-class inherits a `setDataSource(..)` method from the `JdbcDaoSupport` class. You can choose whether to inherit from this class. The `JdbcDaoSupport` class is provided as a convenience only.

Regardless of which of the above template initialization styles you choose to use (or not), it is

seldom necessary to create a new instance of a `JdbcTemplate` class each time you want to run SQL. Once configured, a `JdbcTemplate` instance is thread-safe. If your application accesses multiple databases, you may want multiple `JdbcTemplate` instances, which requires multiple `DataSources` and, subsequently, multiple differently configured `JdbcTemplate` instances.

Using `NamedParameterJdbcTemplate`

The `NamedParameterJdbcTemplate` class adds support for programming JDBC statements by using named parameters, as opposed to programming JDBC statements using only classic placeholder (`'?'`) arguments. The `NamedParameterJdbcTemplate` class wraps a `JdbcTemplate` and delegates to the wrapped `JdbcTemplate` to do much of its work. This section describes only those areas of the `NamedParameterJdbcTemplate` class that differ from the `JdbcTemplate` itself — namely, programming JDBC statements by using named parameters. The following example shows how to use `NamedParameterJdbcTemplate`:

Java

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name",
firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Integer.class);
}
```

Kotlin

```
private val namedParameterJdbcTemplate = NamedParameterJdbcTemplate(dataSource)

fun countOfActorsByFirstName(firstName: String): Int {
    val sql = "select count(*) from T_ACTOR where first_name = :first_name"
    val namedParameters = MapSqlParameterSource("first_name", firstName)
    return namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Int::class.java)!!
}
```

Notice the use of the named parameter notation in the value assigned to the `sql` variable and the corresponding value that is plugged into the `namedParameters` variable (of type `MapSqlParameterSource`).

Alternatively, you can pass along named parameters and their corresponding values to a `NamedParameterJdbcTemplate` instance by using the `Map`-based style. The remaining methods exposed by the `NamedParameterJdbcOperations` and implemented by the `NamedParameterJdbcTemplate` class follow a similar pattern and are not covered here.

The following example shows the use of the `Map`-based style:

Java

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(*) from T_ACTOR where first_name = :first_name";

    Map<String, String> namedParameters = Collections.singletonMap("first_name",
firstName);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Integer.class);
}
```

Kotlin

```
// some JDBC-backed DAO class...
private val namedParameterJdbcTemplate = NamedParameterJdbcTemplate(dataSource)

fun countOfActorsByFirstName(firstName: String): Int {
    val sql = "select count(*) from T_ACTOR where first_name = :first_name"
    val namedParameters = mapOf("first_name" to firstName)
    return namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Int::class.java)!!
}
```

One nice feature related to the `NamedParameterJdbcTemplate` (and existing in the same Java package) is the `SqlParameterSource` interface. You have already seen an example of an implementation of this interface in one of the previous code snippets (the `MapSqlParameterSource` class). An `SqlParameterSource` is a source of named parameter values to a `NamedParameterJdbcTemplate`. The `MapSqlParameterSource` class is a simple implementation that is an adapter around a `java.util.Map`, where the keys are the parameter names and the values are the parameter values.

Another `SqlParameterSource` implementation is the `BeanPropertySqlParameterSource` class. This class wraps an arbitrary JavaBean (that is, an instance of a class that adheres to the JavaBean conventions) and uses the properties of the wrapped JavaBean as the source of named parameter

values.

The following example shows a typical JavaBean:

Java

```
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...

}
```

Kotlin

```
data class Actor(val id: Long, val firstName: String, val lastName: String)
```

The following example uses a `NamedParameterJdbcTemplate` to return the count of the members of the class shown in the preceding example:

Java

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of the above 'Actor' class
    String sql = "select count(*) from T_ACTOR where first_name = :firstName and
last_name = :lastName";

    SqlParameterSource namedParameters = new
BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Integer.class);
}
```

Kotlin

```
// some JDBC-backed DAO class...
private val namedParameterJdbcTemplate = NamedParameterJdbcTemplate(dataSource)

private val namedParameterJdbcTemplate = NamedParameterJdbcTemplate(dataSource)

fun countOfActors(exampleActor: Actor): Int {
    // notice how the named parameters match the properties of the above 'Actor' class
    val sql = "select count(*) from T_ACTOR where first_name = :firstName and
last_name = :lastName"
    val namedParameters = BeanPropertySqlParameterSource(exampleActor)
    return namedParameterJdbcTemplate.queryForObject(sql, namedParameters,
Int::class.java)!!
}
```

Remember that the `NamedParameterJdbcTemplate` class wraps a classic `JdbcTemplate` template. If you need access to the wrapped `JdbcTemplate` instance to access functionality that is present only in the `JdbcTemplate` class, you can use the `getJdbcOperations()` method to access the wrapped `JdbcTemplate` through the `JdbcOperations` interface.

See also `JdbcTemplate` Best Practices for guidelines on using the `NamedParameterJdbcTemplate` class in the context of an application.

Using `SQLExceptionTranslator`

`SQLExceptionTranslator` is an interface to be implemented by classes that can translate between `SQLException`s and Spring's own `org.springframework.dao.DataAccessException`, which is agnostic in

regard to data access strategy. Implementations can be generic (for example, using SQLState codes for JDBC) or proprietary (for example, using Oracle error codes) for greater precision.

`SQLErrorCodeSQLExceptionTranslator` is the implementation of `SQLExceptionTranslator` that is used by default. This implementation uses specific vendor codes. It is more precise than the `SQLState` implementation. The error code translations are based on codes held in a JavaBean type class called `SQLErrorCodes`. This class is created and populated by an `SQLErrorCodesFactory`, which (as the name suggests) is a factory for creating `SQLErrorCodes` based on the contents of a configuration file named `sql-error-codes.xml`. This file is populated with vendor codes and based on the `DatabaseProductName` taken from `DatabaseMetaData`. The codes for the actual database you are using are used.

The `SQLErrorCodeSQLExceptionTranslator` applies matching rules in the following sequence:

1. Any custom translation implemented by a subclass. Normally, the provided concrete `SQLErrorCodeSQLExceptionTranslator` is used, so this rule does not apply. It applies only if you have actually provided a subclass implementation.
2. Any custom implementation of the `SQLExceptionTranslator` interface that is provided as the `customSqlExceptionTranslator` property of the `SQLErrorCodes` class.
3. The list of instances of the `CustomSQLErrorCodesTranslation` class (provided for the `customTranslations` property of the `SQLErrorCodes` class) are searched for a match.
4. Error code matching is applied.
5. Use the fallback translator. `SQLExceptionSubclassTranslator` is the default fallback translator. If this translation is not available, the next fallback translator is the `SQLStateSQLExceptionTranslator`.

The `SQLErrorCodesFactory` is used by default to define `Error` codes and custom exception translations. They are looked up in a file named `sql-error-codes.xml` from the classpath, and the matching `SQLErrorCodes` instance is located based on the database name from the database metadata of the database in use.

You can extend `SQLErrorCodeSQLExceptionTranslator`, as the following example shows:

Java

```
public class CustomSQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator
{

    protected DataAccessException customTranslate(String task, String sql,
SQLException sqlEx) {
        if (sqlEx.getErrorCode() == -12345) {
            return new DeadlockLoserDataAccessException(task, sqlEx);
        }
        return null;
    }
}
```

Kotlin

```
class CustomSQLErrorCodesTranslator : SQLErrorCodeSQLExceptionTranslator() {

    override fun customTranslate(task: String, sql: String?, sqlEx: SQLException):
DataAccessException? {
        if (sqlEx.errorCode == -12345) {
            return DeadlockLoserDataAccessException(task, sqlEx)
        }
        return null
    }
}
```

In the preceding example, the specific error code (`-12345`) is translated, while other errors are left to be translated by the default translator implementation. To use this custom translator, you must pass it to the `JdbcTemplate` through the method `setExceptionTranslator`, and you must use this `JdbcTemplate` for all of the data access processing where this translator is needed. The following example shows how you can use this custom translator:

Java

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

    // create a JdbcTemplate and set data source
    this.jdbcTemplate = new JdbcTemplate();
    this.jdbcTemplate.setDataSource(dataSource);

    // create a custom translator and set the DataSource for the default translation
lookup
    CustomSQLErrorCodesTranslator tr = new CustomSQLErrorCodesTranslator();
    tr.setDataSource(dataSource);
    this.jdbcTemplate.setExceptionTranslator(tr);

}

public void updateShippingCharge(long orderId, long pct) {
    // use the prepared JdbcTemplate for this update
    this.jdbcTemplate.update("update orders" +
        " set shipping_charge = shipping_charge * ? / 100" +
        " where id = ?", pct, orderId);
}
```

Kotlin

```
// create a JdbcTemplate and set data source
private val jdbcTemplate = JdbcTemplate(dataSource).apply {
    // create a custom translator and set the DataSource for the default translation
lookup
    exceptionTranslator = CustomSQLErrorCodesTranslator().apply {
        this.dataSource = dataSource
    }
}

fun updateShippingCharge(orderId: Long, pct: Long) {
    // use the prepared JdbcTemplate for this update
    this.jdbcTemplate!!.update("update orders" +
            " set shipping_charge = shipping_charge * ? / 100" +
            " where id = ?", pct, orderId)
}
```

The custom translator is passed a data source in order to look up the error codes in `sql-error-codes.xml`.

Running Statements

Running an SQL statement requires very little code. You need a `DataSource` and a `JdbcTemplate`, including the convenience methods that are provided with the `JdbcTemplate`. The following example shows what you need to include for a minimal but fully functional class that creates a new table:

Java

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void doExecute() {
        this.jdbcTemplate.execute("create table mytable (id integer, name
varchar(100))");
    }
}
```

Kotlin

```
import javax.sql.DataSource
import org.springframework.jdbc.core.JdbcTemplate

class ExecuteAStatement(dataSource: DataSource) {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    fun doExecute() {
        jdbcTemplate.execute("create table mytable (id integer, name varchar(100))")
    }
}
```

Running Queries

Some query methods return a single value. To retrieve a count or a specific value from one row, use `queryForObject(..)`. The latter converts the returned JDBC `Type` to the Java class that is passed in as an argument. If the type conversion is invalid, an `InvalidDataAccessApiUsageException` is thrown. The following example contains two query methods, one for an `int` and one that queries for a `String`:

Java

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int getCount() {
        return this.jdbcTemplate.queryForObject("select count(*) from mytable",
Integer.class);
    }

    public String getName() {
        return this.jdbcTemplate.queryForObject("select name from mytable",
String.class);
    }
}
```

Kotlin

```
import javax.sql.DataSource
import org.springframework.jdbc.core.JdbcTemplate

class RunAQuery(dataSource: DataSource) {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    val count: Int
        get() = jdbcTemplate.queryForObject("select count(*) from mytable")!!

    val name: String?
        get() = jdbcTemplate.queryForObject("select name from mytable")
}
```

In addition to the single result query methods, several methods return a list with an entry for each row that the query returned. The most generic method is `queryForList(..)`, which returns a `List` where each element is a `Map` containing one entry for each column, using the column name as the key. If you add a method to the preceding example to retrieve a list of all the rows, it might be as follows:

Java

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public List<Map<String, Object>> getList() {
    return this.jdbcTemplate.queryForList("select * from mytable");
}
```

Kotlin

```
private val jdbcTemplate = JdbcTemplate(dataSource)

fun getList(): List<Map<String, Any>> {
    return jdbcTemplate.queryForList("select * from mytable")
}
```

The returned list would resemble the following:

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

Updating the Database

The following example updates a column for a certain primary key:

Java

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void setName(int id, String name) {
        this.jdbcTemplate.update("update mytable set name = ? where id = ?", name,
id);
    }
}
```

Kotlin

```
import javax.sql.DataSource
import org.springframework.jdbc.core.JdbcTemplate

class ExecuteAnUpdate(dataSource: DataSource) {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    fun setName(id: Int, name: String) {
        jdbcTemplate.update("update mytable set name = ? where id = ?", name, id)
    }
}
```

In the preceding example, an SQL statement has placeholders for row parameters. You can pass the parameter values in as varargs or, alternatively, as an array of objects. Thus, you should explicitly wrap primitives in the primitive wrapper classes, or you should use auto-boxing.

Retrieving Auto-generated Keys

An `update()` convenience method supports the retrieval of primary keys generated by the database. This support is part of the JDBC 3.0 standard. See Chapter 13.6 of the specification for details. The method takes a `PreparedStatementCreator` as its first argument, and this is the way the required insert statement is specified. The other argument is a `KeyHolder`, which contains the generated key on successful return from the update. There is no standard single way to create an appropriate `PreparedStatement` (which explains why the method signature is the way it is). The following example works on Oracle but may not work on other platforms:

Java

```
final String INSERT_SQL = "insert into my_test (name) values(?)";
final String name = "Rob";

KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update(connection -> {
    PreparedStatement ps = connection.prepareStatement(INSERT_SQL, new String[] { "id"
});
    ps.setString(1, name);
    return ps;
}, keyHolder);

// keyHolder.getKey() now contains the generated key
```

Kotlin

```
val INSERT_SQL = "insert into my_test (name) values(?)"
val name = "Rob"

val keyHolder = GeneratedKeyHolder()
jdbcTemplate.update({
    it.prepareStatement (INSERT_SQL, arrayOf("id")).apply { setString(1, name) }
}, keyHolder)

// keyHolder.getKey() now contains the generated key
```

4.3.4. Controlling Database Connections

This section covers:

- Using `DataSource`
- Using `DataSourceUtils`
- Implementing `SmartDataSource`
- Extending `AbstractDataSource`
- Using `SingleConnectionDataSource`
- Using `DriverManagerDataSource`
- Using `TransactionAwareDataSourceProxy`
- Using `DataSourceTransactionManager`

Using `DataSource`

Spring obtains a connection to the database through a `DataSource`. A `DataSource` is part of the JDBC specification and is a generalized connection factory. It lets a container or a framework hide connection pooling and transaction management issues from the application code. As a developer, you need not know details about how to connect to the database. That is the responsibility of the

administrator who sets up the datasource. You most likely fill both roles as you develop and test code, but you do not necessarily have to know how the production data source is configured.

When you use Spring's JDBC layer, you can obtain a data source from JNDI, or you can configure your own with a connection pool implementation provided by a third party. Traditional choices are Apache Commons DBCP and C3P0 with bean-style `DataSource` classes; for a modern JDBC connection pool, consider HikariCP with its builder-style API instead.

You should use the `DriverManagerDataSource` and `SimpleDriverDataSource` classes (as included in the Spring distribution) only for testing purposes! Those variants do not provide pooling and perform poorly when multiple requests for a connection are made.

The following section uses Spring's `DriverManagerDataSource` implementation. Several other `DataSource` variants are covered later.

To configure a `DriverManagerDataSource`:

1. Obtain a connection with `DriverManagerDataSource` as you typically obtain a JDBC connection.
2. Specify the fully qualified classname of the JDBC driver so that the `DriverManager` can load the driver class.
3. Provide a URL that varies between JDBC drivers. (See the documentation for your driver for the correct value.)
4. Provide a username and a password to connect to the database.

The following example shows how to configure a `DriverManagerDataSource` in Java:

Java

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername("sa");
dataSource.setPassword("");
```

Kotlin

```
val dataSource = DriverManagerDataSource().apply {
    setDriverClassName("org.hsqldb.jdbcDriver")
    url = "jdbc:hsqldb:hsql://localhost:"
    username = "sa"
    password = ""
}
```

The following example shows the corresponding XML configuration:

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

The next two examples show the basic connectivity and configuration for DBCP and C3P0. To learn about more options that help control the pooling features, see the product documentation for the respective connection pooling implementations.

The following example shows DBCP configuration:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

The following example shows C3P0 configuration:

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-
method="close">
    <property name="driverClass" value="${jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

Using `DataSourceUtils`

The `DataSourceUtils` class is a convenient and powerful helper class that provides `static` methods to obtain connections from JNDI and close connections if necessary. It supports thread-bound connections with, for example, `DataSourceTransactionManager`.

Implementing `SmartDataSource`

The `SmartDataSource` interface should be implemented by classes that can provide a connection to a

relational database. It extends the `DataSource` interface to let classes that use it query whether the connection should be closed after a given operation. This usage is efficient when you know that you need to reuse a connection.

Extending `AbstractDataSource`

`AbstractDataSource` is an `abstract` base class for Spring's `DataSource` implementations. It implements code that is common to all `DataSource` implementations. You should extend the `AbstractDataSource` class if you write your own `DataSource` implementation.

Using `SingleConnectionDataSource`

The `SingleConnectionDataSource` class is an implementation of the `SmartDataSource` interface that wraps a single `Connection` that is not closed after each use. This is not multi-threading capable.

If any client code calls `close` on the assumption of a pooled connection (as when using persistence tools), you should set the `suppressClose` property to `true`. This setting returns a close-suppressing proxy that wraps the physical connection. Note that you can no longer cast this to a native Oracle `Connection` or a similar object.

`SingleConnectionDataSource` is primarily a test class. It typically enables easy testing of code outside an application server, in conjunction with a simple JNDI environment. In contrast to `DriverManagerDataSource`, it reuses the same connection all the time, avoiding excessive creation of physical connections.

Using `DriverManagerDataSource`

The `DriverManagerDataSource` class is an implementation of the standard `DataSource` interface that configures a plain JDBC driver through bean properties and returns a new `Connection` every time.

This implementation is useful for test and stand-alone environments outside of a Jakarta EE container, either as a `DataSource` bean in a Spring IoC container or in conjunction with a simple JNDI environment. Pool-assuming `Connection.close()` calls close the connection, so any `DataSource` -aware persistence code should work. However, using JavaBean-style connection pools (such as `commons-dbcp`) is so easy, even in a test environment, that it is almost always preferable to use such a connection pool over `DriverManagerDataSource`.

Using `TransactionAwareDataSourceProxy`

`TransactionAwareDataSourceProxy` is a proxy for a target `DataSource`. The proxy wraps that target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource`, as provided by a Jakarta EE server.

It is rarely desirable to use this class, except when already existing code must be called and passed a standard JDBC `DataSource` interface implementation. In this case, you can still have this code be usable and, at the same time, have this code participating in Spring managed transactions. It is generally preferable to write your own new code by using the higher level abstractions for resource management, such as `JdbcTemplate` or `DataSourceUtils`.

See the `TransactionAwareDataSourceProxy` javadoc for more details.

Using `DataSourceTransactionManager`

The `DataSourceTransactionManager` class is a `PlatformTransactionManager` implementation for single JDBC datasources. It binds a JDBC connection from the specified data source to the currently executing thread, potentially allowing for one thread connection per data source.

Application code is required to retrieve the JDBC connection through `DataSourceUtils.getConnection(DataSource)` instead of Jakarta EE's standard `DataSource.getConnection`. It throws unchecked `org.springframework.dao` exceptions instead of checked `SQLExceptions`. All framework classes (such as `JdbcTemplate`) use this strategy implicitly. If not used with this transaction manager, the lookup strategy behaves exactly like the common one. Thus, it can be used in any case.

The `DataSourceTransactionManager` class supports custom isolation levels and timeouts that get applied as appropriate JDBC statement query timeouts. To support the latter, application code must either use `JdbcTemplate` or call the `DataSourceUtils.applyTransactionTimeout(..)` method for each created statement.

You can use this implementation instead of `JtaTransactionManager` in the single-resource case, as it does not require the container to support JTA. Switching between both is just a matter of configuration, provided you stick to the required connection lookup pattern. JTA does not support custom isolation levels.

4.3.5. JDBC Batch Operations

Most JDBC drivers provide improved performance if you batch multiple calls to the same prepared statement. By grouping updates into batches, you limit the number of round trips to the database.

Basic Batch Operations with `JdbcTemplate`

You accomplish `JdbcTemplate` batch processing by implementing two methods of a special interface, `BatchPreparedStatementSetter`, and passing that implementation in as the second parameter in your `batchUpdate` method call. You can use the `getBatchSize` method to provide the size of the current batch. You can use the `setValues` method to set the values for the parameters of the prepared statement. This method is called the number of times that you specified in the `getBatchSize` call. The following example updates the `t_actor` table based on entries in a list, and the entire list is used as the batch:

Java

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] batchUpdate(final List<Actor> actors) {
        return this.jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                new BatchPreparedStatementSetter() {
                    public void setValues(PreparedStatement ps, int i) throws
SQLException {
                        Actor actor = actors.get(i);
                        ps.setString(1, actor.getFirstName());
                        ps.setString(2, actor.getLastName());
                        ps.setLong(3, actor.getId().longValue());
                    }
                    public int getBatchSize() {
                        return actors.size();
                    }
                });
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    fun batchUpdate(actors: List<Actor>): IntArray {
        return jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                object: BatchPreparedStatementSetter {
                    override fun setValues(ps: PreparedStatement, i: Int) {
                        ps.setString(1, actors[i].firstName)
                        ps.setString(2, actors[i].lastName)
                        ps.setLong(3, actors[i].id)
                    }

                    override fun getBatchSize() = actors.size
                })
    }

    // ... additional methods
}
```

If you process a stream of updates or reading from a file, you might have a preferred batch size, but the last batch might not have that number of entries. In this case, you can use the `InterruptibleBatchPreparedStatementSetter` interface, which lets you interrupt a batch once the input source is exhausted. The `isBatchExhausted` method lets you signal the end of the batch.

Batch Operations with a List of Objects

Both the `JdbcTemplate` and the `NamedParameterJdbcTemplate` provides an alternate way of providing the batch update. Instead of implementing a special batch interface, you provide all parameter values in the call as a list. The framework loops over these values and uses an internal prepared statement setter. The API varies, depending on whether you use named parameters. For the named parameters, you provide an array of `SqlParameterSource`, one entry for each member of the batch. You can use the `SqlParameterSourceUtils.createBatch` convenience methods to create this array, passing in an array of bean-style objects (with getter methods corresponding to parameters), `String` -keyed `Map` instances (containing the corresponding parameters as values), or a mix of both.

The following example shows a batch update using named parameters:

Java

```
public class JdbcActorDao implements ActorDao {

    private NamedParameterTemplate namedParameterJdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
    }

    public int[] batchUpdate(List<Actor> actors) {
        return this.namedParameterJdbcTemplate.batchUpdate(
                "update t_actor set first_name = :firstName, last_name = :lastName
where id = :id",
                SqlParameterSourceUtils.createBatch(actors));
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val namedParameterJdbcTemplate = NamedParameterJdbcTemplate(dataSource)

    fun batchUpdate(actors: List<Actor>): IntArray {
        return this.namedParameterJdbcTemplate.batchUpdate(
                "update t_actor set first_name = :firstName, last_name = :lastName
where id = :id",
                SqlParameterSourceUtils.createBatch(actors));
    }

        // ... additional methods
}
```

For an SQL statement that uses the classic ? placeholders, you pass in a list containing an object array with the update values. This object array must have one entry for each placeholder in the SQL statement, and they must be in the same order as they are defined in the SQL statement.

The following example is the same as the preceding example, except that it uses classic JDBC ? placeholders:

Java

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[] batchUpdate(final List<Actor> actors) {
        List<Object[]> batch = new ArrayList<Object[]>();
        for (Actor actor : actors) {
            Object[] values = new Object[] {
                    actor.getFirstName(), actor.getLastName(), actor.getId()};
            batch.add(values);
        }
        return this.jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                batch);
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    fun batchUpdate(actors: List<Actor>): IntArray {
        val batch = mutableListOf<Array<Any>>()
        for (actor in actors) {
            batch.add(arrayOf(actor.firstName, actor.lastName, actor.id))
        }
        return jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
batch)
    }

    // ... additional methods
}
```

All of the batch update methods that we described earlier return an `int` array containing the number of affected rows for each batch entry. This count is reported by the JDBC driver. If the count is not available, the JDBC driver returns a value of `-2`.

In such a scenario, with automatic setting of values on an underlying `PreparedStatement`, the corresponding JDBC type for each value needs to be derived from the given Java type. While this usually works well, there is a potential for issues (for example, with Map-contained `null` values). Spring, by default, calls `ParameterMetaData.getParameterType` in such a case, which can be expensive with your JDBC driver. You should use a recent driver version and consider setting the `spring.jdbc.getParameterType.ignore` property to `true` (as a JVM system property or via the `SpringProperties` mechanism) if you encounter a performance issue (as reported on Oracle 12c, JBoss, and PostgreSQL).

Alternatively, you might consider specifying the corresponding JDBC types explicitly, either through a `BatchPreparedStatementSetter` (as shown earlier), through an explicit type array given to a `List<Object[]>` based call, through `registerSqlType` calls on a custom `MapSqlParameterSource` instance, or through a `BeanPropertySqlParameterSource` that derives the SQL type from the Java-declared property type even for a null value.

Batch Operations with Multiple Batches

The preceding example of a batch update deals with batches that are so large that you want to break them up into several smaller batches. You can do this with the methods mentioned earlier by making multiple calls to the `batchUpdate` method, but there is now a more convenient method. This method takes, in addition to the SQL statement, a `Collection` of objects that contain the parameters, the number of updates to make for each batch, and a `ParameterizedPreparedStatementSetter` to set the values for the parameters of the prepared statement. The framework loops over the provided values and breaks the update calls into batches of the size specified.

The following example shows a batch update that uses a batch size of 100:

Java

```
public class JdbcActorDao implements ActorDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int[][] batchUpdate(final Collection<Actor> actors) {
        int[][] updateCounts = jdbcTemplate.batchUpdate(
                "update t_actor set first_name = ?, last_name = ? where id = ?",
                actors,
                100,
                (PreparedStatement ps, Actor actor) -> {
                    ps.setString(1, actor.getFirstName());
                    ps.setString(2, actor.getLastName());
                    ps.setLong(3, actor.getId().longValue());
                });
        return updateCounts;
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val jdbcTemplate = JdbcTemplate(dataSource)

    fun batchUpdate(actors: List<Actor>): Array<IntArray> {
        return jdbcTemplate.batchUpdate(
                    "update t_actor set first_name = ?, last_name = ? where id = ?",
                    actors, 100) { ps, argument ->
            ps.setString(1, argument.firstName)
            ps.setString(2, argument.lastName)
            ps.setLong(3, argument.id)
        }
    }

    // ... additional methods
}
```

The batch update method for this call returns an array of `int` arrays that contains an array entry for each batch with an array of the number of affected rows for each update. The top-level array's length indicates the number of batches run, and the second level array's length indicates the number of updates in that batch. The number of updates in each batch should be the batch size

provided for all batches (except that the last one that might be less), depending on the total number of update objects provided. The update count for each update statement is the one reported by the JDBC driver. If the count is not available, the JDBC driver returns a value of `-2`.

4.3.6. Simplifying JDBC Operations with the `SimpleJdbc` Classes

The `SimpleJdbcInsert` and `SimpleJdbcCall` classes provide a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver. This means that you have less to configure up front, although you can override or turn off the metadata processing if you prefer to provide all the details in your code.

Inserting Data by Using `SimpleJdbcInsert`

We start by looking at the `SimpleJdbcInsert` class with the minimal amount of configuration options. You should instantiate the `SimpleJdbcInsert` in the data access layer's initialization method. For this example, the initializing method is the `setDataSource` method. You do not need to subclass the `SimpleJdbcInsert` class. Instead, you can create a new instance and set the table name by using the `withTableName` method. Configuration methods for this class follow the `fluid` style that returns the instance of the `SimpleJdbcInsert`, which lets you chain all configuration methods. The following example uses only one configuration method (we show examples of multiple methods later):

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.insertActor = new SimpleJdbcInsert(dataSource).withTableName("t_actor");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(3);
        parameters.put("id", actor.getId());
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        insertActor.execute(parameters);
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val insertActor = SimpleJdbcInsert(dataSource).withTableName("t_actor")

    fun add(actor: Actor) {
        val parameters = mutableMapOf<String, Any>()
        parameters["id"] = actor.id
        parameters["first_name"] = actor.firstName
        parameters["last_name"] = actor.lastName
        insertActor.execute(parameters)
    }

    // ... additional methods
}
```

The `execute` method used here takes a plain `java.util.Map` as its only parameter. The important thing to note here is that the keys used for the `Map` must match the column names of the table, as defined in the database. This is because we read the metadata to construct the actual insert statement.

Retrieving Auto-generated Keys by Using `SimpleJdbcInsert`

The next example uses the same insert as the preceding example, but, instead of passing in the `id`, it retrieves the auto-generated key and sets it on the new `Actor` object. When it creates the `SimpleJdbcInsert`, in addition to specifying the table name, it specifies the name of the generated key column with the `usingGeneratedKeyColumns` method. The following listing shows how it works:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val insertActor = SimpleJdbcInsert(dataSource)
            .withTableName("t_actor").usingGeneratedKeyColumns("id")

    fun add(actor: Actor): Actor {
        val parameters = mapOf(
                "first_name" to actor.firstName,
                "last_name" to actor.lastName)
        val newId = insertActor.executeAndReturnKey(parameters);
        return actor.copy(id = newId.toLong())
    }

    // ... additional methods
}
```

The main difference when you run the insert by using this second approach is that you do not add the `id` to the `Map`, and you call the `executeAndReturnKey` method. This returns a `java.lang.Number` object with which you can create an instance of the numerical type that is used in your domain class. You cannot rely on all databases to return a specific Java class here. `java.lang.Number` is the base class that you can rely on. If you have multiple auto-generated columns or the generated values are non-numeric, you can use a `KeyHolder` that is returned from the `executeAndReturnKeyHolder` method.

Specifying Columns for a `SimpleJdbcInsert`

You can limit the columns for an insert by specifying a list of column names with the `usingColumns` method, as the following example shows:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingColumns("first_name", "last_name")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        Map<String, Object> parameters = new HashMap<String, Object>(2);
        parameters.put("first_name", actor.getFirstName());
        parameters.put("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val insertActor = SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingColumns("first_name", "last_name")
            .usingGeneratedKeyColumns("id")

    fun add(actor: Actor): Actor {
        val parameters = mapOf(
                "first_name" to actor.firstName,
                "last_name" to actor.lastName)
        val newId = insertActor.executeAndReturnKey(parameters);
        return actor.copy(id = newId.toLong())
    }

    // ... additional methods
}
```

The execution of the insert is the same as if you had relied on the metadata to determine which

columns to use.

Using `SqlParameterSource` to Provide Parameter Values

Using a `Map` to provide parameter values works fine, but it is not the most convenient class to use. Spring provides a couple of implementations of the `SqlParameterSource` interface that you can use instead. The first one is `BeanPropertySqlParameterSource`, which is a very convenient class if you have a JavaBean-compliant class that contains your values. It uses the corresponding getter method to extract the parameter values. The following example shows how to use `BeanPropertySqlParameterSource`:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new BeanPropertySqlParameterSource(actor);
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val insertActor = SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingGeneratedKeyColumns("id")

    fun add(actor: Actor): Actor {
        val parameters = BeanPropertySqlParameterSource(actor)
        val newId = insertActor.executeAndReturnKey(parameters)
        return actor.copy(id = newId.toLong())
    }

    // ... additional methods
}
```

Another option is the `MapSqlParameterSource` that resembles a `Map` but provides a more convenient

`addValue` method that can be chained. The following example shows how to use it:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcInsert insertActor;

    public void setDataSource(DataSource dataSource) {
        this.insertActor = new SimpleJdbcInsert(dataSource)
                .withTableName("t_actor")
                .usingGeneratedKeyColumns("id");
    }

    public void add(Actor actor) {
        SqlParameterSource parameters = new MapSqlParameterSource()
                .addValue("first_name", actor.getFirstName())
                .addValue("last_name", actor.getLastName());
        Number newId = insertActor.executeAndReturnKey(parameters);
        actor.setId(newId.longValue());
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val insertActor = SimpleJdbcInsert(dataSource)
            .withTableName("t_actor")
            .usingGeneratedKeyColumns("id")

    fun add(actor: Actor): Actor {
        val parameters = MapSqlParameterSource()
                    .addValue("first_name", actor.firstName)
                    .addValue("last_name", actor.lastName)
        val newId = insertActor.executeAndReturnKey(parameters)
        return actor.copy(id = newId.toLong())
    }

    // ... additional methods
}
```

As you can see, the configuration is the same. Only the executing code has to change to use these alternative input classes.

Calling a Stored Procedure with `SimpleJdbcCall`

The `SimpleJdbcCall` class uses metadata in the database to look up names of `in` and `out` parameters

so that you do not have to explicitly declare them. You can declare parameters if you prefer to do that or if you have parameters (such as `ARRAY` or `STRUCT`) that do not have an automatic mapping to a Java class. The first example shows a simple procedure that returns only scalar values in `VARCHAR` and `DATE` format from a MySQL database. The example procedure reads a specified actor entry and returns `first_name`, `last_name`, and `birth_date` columns in the form of `out` parameters. The following listing shows the first example:

```
CREATE PROCEDURE read_actor (
    IN in_id INTEGER,
    OUT out_first_name VARCHAR(100),
    OUT out_last_name VARCHAR(100),
    OUT out_birth_date DATE)
BEGIN
    SELECT first_name, last_name, birth_date
    INTO out_first_name, out_last_name, out_birth_date
    FROM t_actor where id = in_id;
END;
```

The `in_id` parameter contains the `id` of the actor that you are looking up. The `out` parameters return the data read from the table.

You can declare `SimpleJdbcCall` in a manner similar to declaring `SimpleJdbcInsert`. You should instantiate and configure the class in the initialization method of your data-access layer. Compared to the `StoredProcedure` class, you need not create a subclass and you need not to declare parameters that can be looked up in the database metadata. The following example of a `SimpleJdbcCall` configuration uses the preceding stored procedure (the only configuration option, in addition to the `DataSource`, is the name of the stored procedure):

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        this.procReadActor = new SimpleJdbcCall(dataSource)
                .withProcedureName("read_actor");
    }

    public Actor readActor(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        Map out = procReadActor.execute(in);
        Actor actor = new Actor();
        actor.setId(id);
        actor.setFirstName((String) out.get("out_first_name"));
        actor.setLastName((String) out.get("out_last_name"));
        actor.setBirthDate((Date) out.get("out_birth_date"));
        return actor;
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val procReadActor = SimpleJdbcCall(dataSource)
            .withProcedureName("read_actor")


    fun readActor(id: Long): Actor {
        val source = MapSqlParameterSource().addValue("in_id", id)
        val output = procReadActor.execute(source)
        return Actor(
                id,
                output["out_first_name"] as String,
                output["out_last_name"] as String,
                output["out_birth_date"] as Date)
    }

        // ... additional methods
}
```

The code you write for the execution of the call involves creating an `SqlParameterSource` containing the IN parameter. You must match the name provided for the input value with that of the parameter name declared in the stored procedure. The case does not have to match because you

use metadata to determine how database objects should be referred to in a stored procedure. What is specified in the source for the stored procedure is not necessarily the way it is stored in the database. Some databases transform names to all upper case, while others use lower case or use the case as specified.

The `execute` method takes the IN parameters and returns a `Map` that contains any `out` parameters keyed by the name, as specified in the stored procedure. In this case, they are `out_first_name`, `out_last_name`, and `out_birth_date`.

The last part of the `execute` method creates an `Actor` instance to use to return the data retrieved. Again, it is important to use the names of the `out` parameters as they are declared in the stored procedure. Also, the case in the names of the `out` parameters stored in the results map matches that of the `out` parameter names in the database, which could vary between databases. To make your code more portable, you should do a case-insensitive lookup or instruct Spring to use a `LinkedCaseInsensitiveMap`. To do the latter, you can create your own `JdbcTemplate` and set the `setResultsMapCaseInsensitive` property to `true`. Then you can pass this customized `JdbcTemplate` instance into the constructor of your `SimpleJdbcCall`. The following example shows this configuration:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
                .withProcedureName("read_actor");
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private var procReadActor = SimpleJdbcCall(JdbcTemplate(dataSource).apply {
        isResultsMapCaseInsensitive = true
    }).withProcedureName("read_actor")

    // ... additional methods
}
```

By taking this action, you avoid conflicts in the case used for the names of your returned `out` parameters.

Explicitly Declaring Parameters to Use for a `SimpleJdbcCall`

Earlier in this chapter, we described how parameters are deduced from metadata, but you can declare them explicitly if you wish. You can do so by creating and configuring `SimpleJdbcCall` with the `declareParameters` method, which takes a variable number of `SqlParameter` objects as input. See the next section for details on how to define an `SqlParameter`.

Explicit declarations are necessary if the database you use is not a Spring-supported database. Currently, Spring supports metadata lookup of stored procedure calls for the following databases: Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase. We also support metadata lookup of stored functions for MySQL, Microsoft SQL Server, and Oracle.

You can opt to explicitly declare one, some, or all of the parameters. The parameter metadata is still used where you do not explicitly declare parameters. To bypass all processing of metadata lookups for potential parameters and use only the declared parameters, you can call the method `withoutProcedureColumnMetaDataAccess` as part of the declaration. Suppose that you have two or more different call signatures declared for a database function. In this case, you call `useInParameterNames` to specify the list of IN parameter names to include for a given signature.

The following example shows a fully declared procedure call and uses the information from the preceding example:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadActor;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadActor = new SimpleJdbcCall(jdbcTemplate)
                .withProcedureName("read_actor")
                .withoutProcedureColumnMetaDataAccess()
                .useInParameterNames("in_id")
                .declareParameters(
                        new SqlParameter("in_id", Types.NUMERIC),
                        new SqlOutParameter("out_first_name", Types.VARCHAR),
                        new SqlOutParameter("out_last_name", Types.VARCHAR),
                        new SqlOutParameter("out_birth_date", Types.DATE)
                );
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

        private val procReadActor = SimpleJdbcCall(JdbcTemplate(dataSource).apply {
            isResultsMapCaseInsensitive = true
        }).withProcedureName("read_actor")
                .withoutProcedureColumnMetaDataAccess()
                .useInParameterNames("in_id")
                .declareParameters(
                        SqlParameter("in_id", Types.NUMERIC),
                        SqlOutParameter("out_first_name", Types.VARCHAR),
                        SqlOutParameter("out_last_name", Types.VARCHAR),
                        SqlOutParameter("out_birth_date", Types.DATE)
    )

        // ... additional methods
}
```

The execution and end results of the two examples are the same. The second example specifies all details explicitly rather than relying on metadata.

How to Define `SqlParameters`

To define a parameter for the `SimpleJdbc` classes and also for the RDBMS operations classes (covered in Modeling JDBC Operations as Java Objects) you can use `SqlParameter` or one of its subclasses. To do so, you typically specify the parameter name and SQL type in the constructor. The SQL type is specified by using the `java.sql.Types` constants. Earlier in this chapter, we saw declarations similar to the following:

Java

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

Kotlin

```
SqlParameter("in_id", Types.NUMERIC),
SqlOutParameter("out_first_name", Types.VARCHAR),
```

The first line with the `SqlParameter` declares an IN parameter. You can use IN parameters for both stored procedure calls and for queries by using the `SqlQuery` and its subclasses (covered in Understanding `SqlQuery`).

The second line (with the `SqlOutParameter`) declares an `out` parameter to be used in a stored procedure call. There is also an `SqlInOutParameter` for `InOut` parameters (parameters that provide an IN value to the procedure and that also return a value).

Only parameters declared as `SqlParameter` and `SqlInOutParameter` are used to provide input values. This is different from the `StoredProcedure` class, which (for backwards compatibility reasons) lets input values be provided for parameters declared as `SqlOutParameter`.

For IN parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For `out` parameters, you can provide a `RowMapper` to handle mapping of rows returned from a `REF` cursor. Another option is to specify an `SqlReturnType` that provides an opportunity to define customized handling of the return values.

Calling a Stored Function by Using `SimpleJdbcCall`

You can call a stored function in almost the same way as you call a stored procedure, except that you provide a function name rather than a procedure name. You use the `withFunctionName` method as part of the configuration to indicate that you want to make a call to a function, and the corresponding string for a function call is generated. A specialized call (`executeFunction`) is used to run the function, and it returns the function return value as an object of a specified type, which means you do not have to retrieve the return value from the results map. A similar convenience method (named `executeObject`) is also available for stored procedures that have only one `out` parameter. The following example (for MySQL) is based on a stored function named `get_actor_name` that returns an actor's full name:

```
CREATE FUNCTION get_actor_name (in_id INTEGER)
RETURNS VARCHAR(200) READS SQL DATA
BEGIN
    DECLARE out_name VARCHAR(200);
    SELECT concat(first_name, ' ', last_name)
        INTO out_name
        FROM t_actor where id = in_id;
    RETURN out_name;
END;
```

To call this function, we again create a `SimpleJdbcCall` in the initialization method, as the following example shows:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall funcGetActorName;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.funcGetActorName = new SimpleJdbcCall(jdbcTemplate)
                .withFunctionName("get_actor_name");
    }

    public String getActorName(Long id) {
        SqlParameterSource in = new MapSqlParameterSource()
                .addValue("in_id", id);
        String name = funcGetActorName.executeFunction(String.class, in);
        return name;
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

    private val jdbcTemplate = JdbcTemplate(dataSource).apply {
        isResultsMapCaseInsensitive = true
    }
    private val funcGetActorName = SimpleJdbcCall(jdbcTemplate)
            .withFunctionName("get_actor_name")

    fun getActorName(id: Long): String {
        val source = MapSqlParameterSource().addValue("in_id", id)
        return funcGetActorName.executeFunction(String::class.java, source)
    }

    // ... additional methods
}
```

The `executeFunction` method used returns a `String` that contains the return value from the function call.

Returning a `ResultSet` or REF Cursor from a `SimpleJdbcCall`

Calling a stored procedure or function that returns a result set is a bit tricky. Some databases return result sets during the JDBC results processing, while others require an explicitly registered `out` parameter of a specific type. Both approaches need additional processing to loop over the result set

and process the returned rows. With the `SimpleJdbcCall`, you can use the `returningResultSet` method and declare a `RowMapper` implementation to be used for a specific parameter. If the result set is returned during the results processing, there are no names defined, so the returned results must match the order in which you declare the `RowMapper` implementations. The name specified is still used to store the processed list of results in the results map that is returned from the `execute` statement.

The next example (for MySQL) uses a stored procedure that takes no IN parameters and returns all rows from the `t_actor` table:

```
CREATE PROCEDURE read_all_actors()
BEGIN
 SELECT a.id, a.first_name, a.last_name, a.birth_date FROM t_actor a;
END;
```

To call this procedure, you can declare the `RowMapper`. Because the class to which you want to map follows the JavaBean rules, you can use a `BeanPropertyRowMapper` that is created by passing in the required class to map to in the `newInstance` method. The following example shows how to do so:

Java

```
public class JdbcActorDao implements ActorDao {

    private SimpleJdbcCall procReadAllActors;

    public void setDataSource(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.setResultsMapCaseInsensitive(true);
        this.procReadAllActors = new SimpleJdbcCall(jdbcTemplate)
                .withProcedureName("read_all_actors")
                .returningResultSet("actors",
                BeanPropertyRowMapper.newInstance(Actor.class));
    }

    public List getActorsList() {
        Map m = procReadAllActors.execute(new HashMap<String, Object>(0));
        return (List) m.get("actors");
    }

    // ... additional methods
}
```

Kotlin

```
class JdbcActorDao(dataSource: DataSource) : ActorDao {

        private val procReadAllActors = SimpleJdbcCall(JdbcTemplate(dataSource).apply
{
            isResultsMapCaseInsensitive = true
        }).withProcedureName("read_all_actors")
                .returningResultSet("actors",
                        BeanPropertyRowMapper.newInstance(Actor::class.java))

    fun getActorsList(): List<Actor> {
        val m = procReadAllActors.execute(mapOf<String, Any>())
        return m["actors"] as List<Actor>
    }

    // ... additional methods
}
```

The `execute` call passes in an empty `Map`, because this call does not take any parameters. The list of actors is then retrieved from the results map and returned to the caller.

4.3.7. Modeling JDBC Operations as Java Objects

The `org.springframework.jdbc.object` package contains classes that let you access the database in a more object-oriented manner. As an example, you can run queries and get the results back as a list that contains business objects with the relational column data mapped to the properties of the business object. You can also run stored procedures and run update, delete, and insert statements.

Many Spring developers believe that the various RDBMS operation classes described below (with the exception of the `StoredProcedure` class) can often be replaced with straight `JdbcTemplate` calls. Often, it is simpler to write a DAO method that calls a method on a `JdbcTemplate` directly (as opposed to encapsulating a query as a full-blown class).

However, if you are getting measurable value from using the RDBMS operation classes, you should continue to use these classes.

Understanding `SqlQuery`

`SqlQuery` is a reusable, thread-safe class that encapsulates an SQL query. Subclasses must implement the `newRowMapper(..)` method to provide a `RowMapper` instance that can create one object per row obtained from iterating over the `ResultSet` that is created during the execution of the query. The `SqlQuery` class is rarely used directly, because the `MappingSqlQuery` subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend `SqlQuery` are `MappingSqlQueryWithParameters` and `UpdatableSqlQuery`.

Using `MappingSqlQuery`

`MappingSqlQuery` is a reusable query in which concrete subclasses must implement the abstract `mapRow(..)` method to convert each row of the supplied `ResultSet` into an object of the type specified. The following example shows a custom query that maps the data from the `t_actor` relation to an instance of the `Actor` class:

Java

```
public class ActorMappingQuery extends MappingSqlQuery<Actor> {

    public ActorMappingQuery(DataSource ds) {
        super(ds, "select id, first_name, last_name from t_actor where id = ?");
        declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }

    @Override
    protected Actor mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Actor actor = new Actor();
        actor.setId(rs.getLong("id"));
        actor.setFirstName(rs.getString("first_name"));
        actor.setLastName(rs.getString("last_name"));
        return actor;
    }
}
```

Kotlin

```
class ActorMappingQuery(ds: DataSource) : MappingSqlQuery<Actor>(ds, "select id,
first_name, last_name from t_actor where id = ?") {

    init {
        declareParameter(SqlParameter("id", Types.INTEGER))
        compile()
    }

    override fun mapRow(rs: ResultSet, rowNumber: Int) = Actor(
            rs.getLong("id"),
            rs.getString("first_name"),
            rs.getString("last_name")
    )
}
```

The class extends `MappingSqlQuery` parameterized with the `Actor` type. The constructor for this customer query takes a `DataSource` as the only parameter. In this constructor, you can call the constructor on the superclass with the `DataSource` and the SQL that should be run to retrieve the rows for this query. This SQL is used to create a `PreparedStatement`, so it may contain placeholders for any parameters to be passed in during execution. You must declare each parameter by using the `declareParameter` method passing in an `SqlParameter`. The `SqlParameter` takes a name, and the JDBC

type as defined in `java.sql.Types`. After you define all parameters, you can call the `compile()` method so that the statement can be prepared and later run. This class is thread-safe after it is compiled, so, as long as these instances are created when the DAO is initialized, they can be kept as instance variables and be reused. The following example shows how to define such a class:

Java

```
private ActorMappingQuery actorMappingQuery;

@Autowired
public void setDataSource(DataSource dataSource) {
    this.actorMappingQuery = new ActorMappingQuery(dataSource);
}

public Customer getCustomer(Long id) {
    return actorMappingQuery.findObject(id);
}
```

Kotlin

```
private val actorMappingQuery = ActorMappingQuery(dataSource)

fun getCustomer(id: Long) = actorMappingQuery.findObject(id)
```

The method in the preceding example retrieves the customer with the `id` that is passed in as the only parameter. Since we want only one object to be returned, we call the `findObject` convenience method with the `id` as the parameter. If we had instead a query that returned a list of objects and took additional parameters, we would use one of the `execute` methods that takes an array of parameter values passed in as varargs. The following example shows such a method:

Java

```
public List<Actor> searchForActors(int age, String namePattern) {
    List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);
    return actors;
}
```

Kotlin

```
fun searchForActors(age: Int, namePattern: String) =
            actorSearchMappingQuery.execute(age, namePattern)
```

Using `SqlUpdate`

The `SqlUpdate` class encapsulates an SQL update. As with a query, an update object is reusable, and, as with all `RdbmsOperation` classes, an update can have parameters and is defined in SQL. This class provides a number of `update(..)` methods analogous to the `execute(..)` methods of query objects. The `SqlUpdate` class is concrete. It can be subclassed — for example, to add a custom update method.

However, you do not have to subclass the `SqlUpdate` class, since it can easily be parameterized by setting SQL and declaring parameters. The following example creates a custom update method named `execute`:

Java

```
import java.sql.Types;
import javax.sql.DataSource;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter("creditRating", Types.NUMERIC));
        declareParameter(new SqlParameter("id", Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int execute(int id, int rating) {
        return update(rating, id);
    }
}
```

Kotlin

```
import java.sql.Types
import javax.sql.DataSource
import org.springframework.jdbc.core.SqlParameter
import org.springframework.jdbc.object.SqlUpdate

class UpdateCreditRating(ds: DataSource) : SqlUpdate() {

    init {
        setDataSource(ds)
        sql = "update customer set credit_rating = ? where id = ?"
        declareParameter(SqlParameter("creditRating", Types.NUMERIC))
        declareParameter(SqlParameter("id", Types.NUMERIC))
        compile()
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    fun execute(id: Int, rating: Int): Int {
        return update(rating, id)
    }
}
```

Using `StoredProcedure`

The `StoredProcedure` class is an `abstract` superclass for object abstractions of RDBMS stored procedures.

The inherited `sql` property is the name of the stored procedure in the RDBMS.

To define a parameter for the `StoredProcedure` class, you can use an `SqlParameter` or one of its subclasses. You must specify the parameter name and SQL type in the constructor, as the following code snippet shows:

Java

```
new SqlParameter("in_id", Types.NUMERIC),
new SqlOutParameter("out_first_name", Types.VARCHAR),
```

Kotlin

```
SqlParameter("in_id", Types.NUMERIC),
SqlOutParameter("out_first_name", Types.VARCHAR),
```

The SQL type is specified using the `java.sql.Types` constants.

The first line (with the `SqlParameter`) declares an IN parameter. You can use IN parameters both for stored procedure calls and for queries using the `SqlQuery` and its subclasses (covered in Understanding `SqlQuery`).

The second line (with the `SqlOutParameter`) declares an `out` parameter to be used in the stored procedure call. There is also an `SqlInOutParameter` for `InOut` parameters (parameters that provide an `in` value to the procedure and that also return a value).

For `in` parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For `out` parameters, you can provide a `RowMapper` to handle mapping of rows returned from a `REF` cursor. Another option is to specify an `SqlReturnType` that lets you define customized handling of the return values.

The next example of a simple DAO uses a `StoredProcedure` to call a function (`sysdate()`), which comes with any Oracle database. To use the stored procedure functionality, you have to create a class that extends `StoredProcedure`. In this example, the `StoredProcedure` class is an inner class. However, if you need to reuse the `StoredProcedure`, you can declare it as a top-level class. This example has no input parameters, but an output parameter is declared as a date type by using the `SqlOutParameter` class. The `execute()` method runs the procedure and extracts the returned date from the results `Map`. The results `Map` has an entry for each declared output parameter (in this case, only one) by using the parameter name as the key. The following listing shows our custom StoredProcedure class:

Java

```
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class StoredProcedureDao {

    private GetSysdateProcedure getSysdate;

    @Autowired
    public void init(DataSource dataSource) {
        this.getSysdate = new GetSysdateProcedure(dataSource);
    }

    public Date getSysdate() {
        return getSysdate.execute();
    }

    private class GetSysdateProcedure extends StoredProcedure {

        private static final String SQL = "sysdate";

        public GetSysdateProcedure(DataSource dataSource) {
            setDataSource(dataSource);
            setFunction(true);
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.DATE));
            compile();
        }

        public Date execute() {
            // the 'sysdate' sproc has no input parameters, so an empty Map is
supplied...
            Map<String, Object> results = execute(new HashMap<String, Object>());
            Date sysdate = (Date) results.get("date");
            return sysdate;
        }
    }

}
```

Kotlin

```
import java.sql.Types
import java.util.Date
import java.util.Map
import javax.sql.DataSource
import org.springframework.jdbc.core.SqlOutParameter
import org.springframework.jdbc.object.StoredProcedure

class StoredProcedureDao(dataSource: DataSource) {

    private val SQL = "sysdate"

    private val getSysdate = GetSysdateProcedure(dataSource)

    val sysdate: Date
        get() = getSysdate.execute()

    private inner class GetSysdateProcedure(dataSource: DataSource) :
StoredProcedure() {

        init {
            setDataSource(dataSource)
            isFunction = true
            sql = SQL
            declareParameter(SqlOutParameter("date", Types.DATE))
            compile()
        }

        fun execute(): Date {
            // the 'sysdate' sproc has no input parameters, so an empty Map is
supplied...
            val results = execute(mutableMapOf<String, Any>())
            return results["date"] as Date
        }
    }
}
```

The following example of a `StoredProcedure` has two output parameters (in this case, Oracle REF cursors):

Java

```
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new
TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new
GenreMapper()));
        compile();
    }

    public Map<String, Object> execute() {
        // again, this sproc has no input parameters, so an empty Map is supplied
        return super.execute(new HashMap<String, Object>());
    }
}
```

Kotlin

```
import java.util.HashMap
import javax.sql.DataSource
import oracle.jdbc.OracleTypes
import org.springframework.jdbc.core.SqlOutParameter
import org.springframework.jdbc.object.StoredProcedure

class TitlesAndGenresStoredProcedure(dataSource: DataSource) :
StoredProcedure(dataSource, SPROC_NAME) {

    companion object {
        private const val SPROC_NAME = "AllTitlesAndGenres"
    }

    init {
        declareParameter(SqlOutParameter("titles", OracleTypes.CURSOR, TitleMapper()))
        declareParameter(SqlOutParameter("genres", OracleTypes.CURSOR, GenreMapper()))
        compile()
    }

    fun execute(): Map<String, Any> {
        // again, this sproc has no input parameters, so an empty Map is supplied
        return super.execute(HashMap<String, Any>())
    }
}
```

Notice how the overloaded variants of the `declareParameter(..)` method that have been used in the `TitlesAndGenresStoredProcedure` constructor are passed `RowMapper` implementation instances. This is a very convenient and powerful way to reuse existing functionality. The next two examples provide code for the two `RowMapper` implementations.

The `TitleMapper` class maps a `ResultSet` to a `Title` domain object for each row in the supplied `ResultSet`, as follows:

Java

```
import java.sql.ResultSet;
import java.sql.SQLException;
import com.foo.domain.Title;
import org.springframework.jdbc.core.RowMapper;

public final class TitleMapper implements RowMapper<Title> {

    public Title mapRow(ResultSet rs, int rowNum) throws SQLException {
        Title title = new Title();
        title.setId(rs.getLong("id"));
        title.setName(rs.getString("name"));
        return title;
    }
}
```

Kotlin

```
import java.sql.ResultSet
import com.foo.domain.Title
import org.springframework.jdbc.core.RowMapper

class TitleMapper : RowMapper<Title> {

    override fun mapRow(rs: ResultSet, rowNum: Int) =
            Title(rs.getLong("id"), rs.getString("name"))
}
```

The `GenreMapper` class maps a `ResultSet` to a `Genre` domain object for each row in the supplied `ResultSet`, as follows:

Java

```
import java.sql.ResultSet;
import java.sql.SQLException;
import com.foo.domain.Genre;
import org.springframework.jdbc.core.RowMapper;

public final class GenreMapper implements RowMapper<Genre> {

    public Genre mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

Kotlin

```
import java.sql.ResultSet
import com.foo.domain.Genre
import org.springframework.jdbc.core.RowMapper

class GenreMapper : RowMapper<Genre> {

    override fun mapRow(rs: ResultSet, rowNum: Int): Genre {
        return Genre(rs.getString("name"))
    }
}
```

To pass parameters to a stored procedure that has one or more input parameters in its definition in the RDBMS, you can code a strongly typed `execute(..)` method that would delegate to the untyped `execute(Map)` method in the superclass, as the following example shows:

Java

```
import java.sql.Types;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import oracle.jdbc.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.StoredProcedure;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new
TitleMapper()));
        compile();
    }

    public Map<String, Object> execute(Date cutoffDate) {
        Map<String, Object> inputs = new HashMap<String, Object>();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

Kotlin

```
import java.sql.Types
import java.util.Date
import javax.sql.DataSource
import oracle.jdbc.OracleTypes
import org.springframework.jdbc.core.SqlOutParameter
import org.springframework.jdbc.core.SqlParameter
import org.springframework.jdbc.object.StoredProcedure

class TitlesAfterDateStoredProcedure(dataSource: DataSource) :
StoredProcedure(dataSource, SPROC_NAME) {

    companion object {
        private const val SPROC_NAME = "TitlesAfterDate"
        private const val CUTOFF_DATE_PARAM = "cutoffDate"
    }

    init {
        declareParameter(SqlParameter(CUTOFF_DATE_PARAM, Types.DATE))
        declareParameter(SqlOutParameter("titles", OracleTypes.CURSOR, TitleMapper()))
        compile()
    }

    fun execute(cutoffDate: Date) = super.execute(
            mapOf<String, Any>(CUTOFF_DATE_PARAM to cutoffDate))
}
```

4.3.8. Common Problems with Parameter and Data Value Handling

Common problems with parameters and data values exist in the different approaches provided by Spring Framework's JDBC support. This section covers how to address them.

Providing SQL Type Information for Parameters

Usually, Spring determines the SQL type of the parameters based on the type of parameter passed in. It is possible to explicitly provide the SQL type to be used when setting parameter values. This is sometimes necessary to correctly set `NULL` values.

You can provide SQL type information in several ways:

- Many update and query methods of the `JdbcTemplate` take an additional parameter in the form of an `int` array. This array is used to indicate the SQL type of the corresponding parameter by using constant values from the `java.sql.Types` class. Provide one entry for each parameter.
- You can use the `SqlParameterValue` class to wrap the parameter value that needs this additional information. To do so, create a new instance for each value and pass in the SQL type and the parameter value in the constructor. You can also provide an optional scale parameter for numeric values.
- For methods that work with named parameters, you can use the `SqlParameterSource` classes,

`BeanPropertySqlParameterSource` or `MapSqlParameterSource`. They both have methods for registering the SQL type for any of the named parameter values.

Handling BLOB and CLOB objects

You can store images, other binary data, and large chunks of text in the database. These large objects are called BLOBs (Binary Large OBject) for binary data and CLOBs (Character Large OBject) for character data. In Spring, you can handle these large objects by using the `JdbcTemplate` directly and also when using the higher abstractions provided by RDBMS Objects and the `SimpleJdbc` classes. All of these approaches use an implementation of the `LobHandler` interface for the actual management of the LOB (Large OBject) data. `LobHandler` provides access to a `LobCreator` class, through the `getLobCreator` method, that is used for creating new LOB objects to be inserted.

`LobCreator` and `LobHandler` provide the following support for LOB input and output:

- BLOB
  - `byte[]`: `getBlobAsBytes` and `setBlobAsBytes`
  - `InputStream`: `getBlobAsBinaryStream` and `setBlobAsBinaryStream`
- CLOB
  - `String`: `getClobAsString` and `setClobAsString`
  - `InputStream`: `getClobAsAsciiStream` and `setClobAsAsciiStream`
  - `Reader`: `getClobAsCharacterStream` and `setClobAsCharacterStream`

The next example shows how to create and insert a BLOB. Later we show how to read it back from the database.

This example uses a `JdbcTemplate` and an implementation of the `AbstractLobCreatingPreparedStatementCallback`. It implements one method, `setValues`. This method provides a `LobCreator` that we use to set the values for the LOB columns in your SQL insert statement.

For this example, we assume that there is a variable, `lobHandler`, that is already set to an instance of a `DefaultLobHandler`. You typically set this value through dependency injection.

The following example shows how to create and insert a BLOB:

Java

```
final File blobIn = new File("spring2004.jpg");
final InputStream blobIs = new FileInputStream(blobIn);
final File clobIn = new File("large.txt");
final InputStream clobIs = new FileInputStream(clobIn);
final InputStreamReader clobReader = new InputStreamReader(clobIs);

jdbcTemplate.execute(
    "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
    new AbstractLobCreatingPreparedStatementCallback(lobHandler) {  ①
        protected void setValues(PreparedStatement ps, LobCreator lobCreator) throws
SQLException {
            ps.setLong(1, 1L);
            lobCreator.setClobAsCharacterStream(ps, 2, clobReader,
(int)clobIn.length());  ②
            lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length());  ③
        }
    }
);

blobIs.close();
clobReader.close();
```

① Pass in the `lobHandler` that (in this example) is a plain `DefaultLobHandler`.

② Using the method `setClobAsCharacterStream` to pass in the contents of the CLOB.

③ Using the method `setBlobAsBinaryStream` to pass in the contents of the BLOB.

Kotlin

```
val blobIn = File("spring2004.jpg")
val blobIs = FileInputStream(blobIn)
val clobIn = File("large.txt")
val clobIs = FileInputStream(clobIn)
val clobReader = InputStreamReader(clobIs)

jdbcTemplate.execute(
        "INSERT INTO lob_table (id, a_clob, a_blob) VALUES (?, ?, ?)",
        object: AbstractLobCreatingPreparedStatementCallback(lobHandler) {  ①
            override fun setValues(ps: PreparedStatement, lobCreator: LobCreator) {
                ps.setLong(1, 1L)
                lobCreator.setClobAsCharacterStream(ps, 2, clobReader,
clobIn.length().toInt())  ②
                lobCreator.setBlobAsBinaryStream(ps, 3, blobIs,
blobIn.length().toInt())  ③
            }
        }
)
blobIs.close()
clobReader.close()
```

① Pass in the `lobHandler` that (in this example) is a plain `DefaultLobHandler`.

② Using the method `setClobAsCharacterStream` to pass in the contents of the CLOB.

③ Using the method `setBlobAsBinaryStream` to pass in the contents of the BLOB.

If you invoke the `setBlobAsBinaryStream`, `setClobAsAsciiStream`, or `setClobAsCharacterStream` method on the `LobCreator` returned from `DefaultLobHandler.getLobCreator()`, you can optionally specify a negative value for the `contentLength` argument. If the specified content length is negative, the `DefaultLobHandler` uses the JDBC 4.0 variants of the set-stream methods without a length parameter. Otherwise, it passes the specified length on to the driver.

See the documentation for the JDBC driver you use to verify that it supports streaming a LOB without providing the content length.

Now it is time to read the LOB data from the database. Again, you use a `JdbcTemplate` with the same instance variable `lobHandler` and a reference to a `DefaultLobHandler`. The following example shows how to do so:

Java

```
List<Map<String, Object>> l = jdbcTemplate.query("select id, a_clob, a_blob from
lob_table",
    new RowMapper<Map<String, Object>>() {
        public Map<String, Object> mapRow(ResultSet rs, int i) throws SQLException {
            Map<String, Object> results = new HashMap<String, Object>();
            String clobText = lobHandler.getClobAsString(rs, "a_clob");  ①
            results.put("CLOB", clobText);
            byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob");  ②
            results.put("BLOB", blobBytes);
            return results;
        }
    });
```

① Using the method `getClobAsString` to retrieve the contents of the CLOB.

② Using the method `getBlobAsBytes` to retrieve the contents of the BLOB.

Kotlin

```
val l = jdbcTemplate.query("select id, a_clob, a_blob from lob_table") { rs, _ ->
    val clobText = lobHandler.getClobAsString(rs, "a_clob")  ①
    val blobBytes = lobHandler.getBlobAsBytes(rs, "a_blob")  ②
    mapOf("CLOB" to clobText, "BLOB" to blobBytes)
}
```

① Using the method `getClobAsString` to retrieve the contents of the CLOB.

② Using the method `getBlobAsBytes` to retrieve the contents of the BLOB.

Passing in Lists of Values for IN Clause

The SQL standard allows for selecting rows based on an expression that includes a variable list of values. A typical example would be `select * from T_ACTOR where id in (1, 2, 3)`. This variable list is not directly supported for prepared statements by the JDBC standard. You cannot declare a variable number of placeholders. You need a number of variations with the desired number of placeholders prepared, or you need to generate the SQL string dynamically once you know how many placeholders are required. The named parameter support provided in the `NamedParameterJdbcTemplate` and `JdbcTemplate` takes the latter approach. You can pass in the values as a `java.util.List` of primitive objects. This list is used to insert the required placeholders and pass in the values during statement execution.

Be careful when passing in many values. The JDBC standard does not guarantee that you can use more than 100 values for an `in` expression list. Various databases exceed this number, but they usually have a hard limit for how many values are allowed. For example, Oracle's limit is 1000.

In addition to the primitive values in the value list, you can create a `java.util.List` of object arrays. This list can support multiple expressions being defined for the `in` clause, such as `select * from T_ACTOR where (id, last_name) in ((1, 'Johnson'), (2, 'Harrop'))`. This, of course, requires that

your database supports this syntax.

Handling Complex Types for Stored Procedure Calls

When you call stored procedures, you can sometimes use complex types specific to the database. To accommodate these types, Spring provides a `SqlReturnType` for handling them when they are returned from the stored procedure call and `SqlTypeValue` when they are passed in as a parameter to the stored procedure.

The `SqlReturnType` interface has a single method (named `getTypeValue`) that must be implemented. This interface is used as part of the declaration of an `SqlOutParameter`. The following example shows returning the value of an Oracle `STRUCT` object of the user declared type `ITEM_TYPE`:

Java

```
public class TestItemStoredProcedure extends StoredProcedure {

    public TestItemStoredProcedure(DataSource dataSource) {
        // ...
        declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT, "ITEM_TYPE",
            (CallableStatement cs, int colIndx, int sqlType, String typeName) -> {
                STRUCT struct = (STRUCT) cs.getObject(colIndx);
                Object[] attr = struct.getAttributes();
                TestItem item = new TestItem();
                item.setId(((Number) attr[0]).longValue());
                item.setDescription((String) attr[1]);
                item.setExpirationDate((java.util.Date) attr[2]);
                return item;
            }));
        // ...
    }
```

Kotlin

```
class TestItemStoredProcedure(dataSource: DataSource) : StoredProcedure() {

    init {
        // ...
        declareParameter(SqlOutParameter("item", OracleTypes.STRUCT, "ITEM_TYPE") {
cs, colIndx, sqlType, typeName ->
            val struct = cs.getObject(colIndx) as STRUCT
            val attr = struct.getAttributes()
            TestItem((attr[0] as Long, attr[1] as String, attr[2] as Date)
        })
        // ...
    }
}
```

You can use `SqlTypeValue` to pass the value of a Java object (such as `TestItem`) to a stored procedure. The `SqlTypeValue` interface has a single method (named `createTypeValue`) that you must implement.

The active connection is passed in, and you can use it to create database-specific objects, such as `StructDescriptor` instances or `ArrayDescriptor` instances. The following example creates a `StructDescriptor` instance:

Java

```
final TestItem testItem = new TestItem(123L, "A test item",
        new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"));

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName)
throws SQLException {
        StructDescriptor itemDescriptor = new StructDescriptor(typeName, conn);
        Struct item = new STRUCT(itemDescriptor, conn,
        new Object[] {
            testItem.getId(),
            testItem.getDescription(),
            new java.sql.Date(testItem.getExpirationDate().getTime())
        });
        return item;
    }
};
```

Kotlin

```
val (id, description, expirationDate) = TestItem(123L, "A test item",
        SimpleDateFormat("yyyy-M-d").parse("2010-12-31"))

val value = object : AbstractSqlTypeValue() {
    override fun createTypeValue(conn: Connection, sqlType: Int, typeName: String?):
Any {
        val itemDescriptor = StructDescriptor(typeName, conn)
        return STRUCT(itemDescriptor, conn,
                arrayOf(id, description, java.sql.Date(expirationDate.time)))
    }
}
```

You can now add this `SqlTypeValue` to the `Map` that contains the input parameters for the `execute` call of the stored procedure.

Another use for the `SqlTypeValue` is passing in an array of values to an Oracle stored procedure. Oracle has its own internal `ARRAY` class that must be used in this case, and you can use the `SqlTypeValue` to create an instance of the Oracle `ARRAY` and populate it with values from the Java `ARRAY`, as the following example shows:

Java

```
final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {
    protected Object createTypeValue(Connection conn, int sqlType, String typeName)
throws SQLException {
        ArrayDescriptor arrayDescriptor = new ArrayDescriptor(typeName, conn);
        ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);
        return idArray;
    }
};
```

Kotlin

```
class TestItemStoredProcedure(dataSource: DataSource) : StoredProcedure() {

    init {
        val ids = arrayOf(1L, 2L)
        val value = object : AbstractSqlTypeValue() {
            override fun createTypeValue(conn: Connection, sqlType: Int, typeName:
String?): Any {
                val arrayDescriptor = ArrayDescriptor(typeName, conn)
                return ARRAY(arrayDescriptor, conn, ids)
            }
        }
    }
}
```

4.3.9. Embedded Database Support

The `org.springframework.jdbc.datasource.embedded` package provides support for embedded Java database engines. Support for HSQL, H2, and Derby is provided natively. You can also use an extensible API to plug in new embedded database types and `DataSource` implementations.

Why Use an Embedded Database?

An embedded database can be useful during the development phase of a project because of its lightweight nature. Benefits include ease of configuration, quick startup time, testability, and the ability to rapidly evolve your SQL during development.

Creating an Embedded Database by Using Spring XML

If you want to expose an embedded database instance as a bean in a Spring `ApplicationContext`, you can use the `embedded-database` tag in the `spring-jdbc` namespace:

```
<jdbc:embedded-database id="dataSource" generate-name="true">
    <jdbc:script location="classpath:schema.sql"/>
    <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

The preceding configuration creates an embedded HSQL database that is populated with SQL from the `schema.sql` and `test-data.sql` resources in the root of the classpath. In addition, as a best practice, the embedded database is assigned a uniquely generated name. The embedded database is made available to the Spring container as a bean of type `javax.sql.DataSource` that can then be injected into data access objects as needed.

Creating an Embedded Database Programmatically

The `EmbeddedDatabaseBuilder` class provides a fluent API for constructing an embedded database programmatically. You can use this when you need to create an embedded database in a stand-alone environment or in a stand-alone integration test, as in the following example:

Java

```
EmbeddedDatabase db = new EmbeddedDatabaseBuilder()
        .generateUniqueName(true)
        .setType(H2)
        .setScriptEncoding("UTF-8")
        .ignoreFailedDrops(true)
        .addScript("schema.sql")
        .addScripts("user_data.sql", "country_data.sql")
        .build();

// perform actions against the db (EmbeddedDatabase extends javax.sql.DataSource)

db.shutdown()
```

Kotlin

```
val db = EmbeddedDatabaseBuilder()
        .generateUniqueName(true)
        .setType(H2)
        .setScriptEncoding("UTF-8")
        .ignoreFailedDrops(true)
        .addScript("schema.sql")
        .addScripts("user_data.sql", "country_data.sql")
        .build()

// perform actions against the db (EmbeddedDatabase extends javax.sql.DataSource)

db.shutdown()
```

See the javadoc for `EmbeddedDatabaseBuilder` for further details on all supported options.

You can also use the `EmbeddedDatabaseBuilder` to create an embedded database by using Java configuration, as the following example shows:

Java

```
@Configuration
public class DataSourceConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .setType(H2)
                .setScriptEncoding("UTF-8")
                .ignoreFailedDrops(true)
                .addScript("schema.sql")
                .addScripts("user_data.sql", "country_data.sql")
                .build();
    }
}
```

Kotlin

```
@Configuration
class DataSourceConfig {

    @Bean
    fun dataSource(): DataSource {
        return EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .setType(H2)
                .setScriptEncoding("UTF-8")
                .ignoreFailedDrops(true)
                .addScript("schema.sql")
                .addScripts("user_data.sql", "country_data.sql")
                .build()
    }
}
```

Selecting the Embedded Database Type

This section covers how to select one of the three embedded databases that Spring supports. It includes the following topics:

- Using HSQL
- Using H2
- Using Derby

Using HSQL

Spring supports HSQL 1.8.0 and above. HSQL is the default embedded database if no type is explicitly specified. To specify HSQL explicitly, set the `type` attribute of the `embedded-database` tag to `HSQL`. If you use the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.HSQL`.

Using H2

Spring supports the H2 database. To enable H2, set the `type` attribute of the `embedded-database` tag to `H2`. If you use the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.H2`.

Using Derby

Spring supports Apache Derby 10.5 and above. To enable Derby, set the `type` attribute of the `embedded-database` tag to `DERBY`. If you use the builder API, call the `setType(EmbeddedDatabaseType)` method with `EmbeddedDatabaseType.DERBY`.

Testing Data Access Logic with an Embedded Database

Embedded databases provide a lightweight way to test data access code. The next example is a data access integration test template that uses an embedded database. Using such a template can be useful for one-offs when the embedded database does not need to be reused across test classes. However, if you wish to create an embedded database that is shared within a test suite, consider using the Spring TestContext Framework and configuring the embedded database as a bean in the Spring `ApplicationContext` as described in Creating an Embedded Database by Using Spring XML and Creating an Embedded Database Programmatically. The following listing shows the test template:

Java

```
public class DataAccessIntegrationTestTemplate {

    private EmbeddedDatabase db;

    @BeforeEach
    public void setUp() {
        // creates an HSQL in-memory database populated from default scripts
        // classpath:schema.sql and classpath:data.sql
        db = new EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .addDefaultScripts()
                .build();
    }

    @Test
    public void testDataAccess() {
        JdbcTemplate template = new JdbcTemplate(db);
        template.query( /* ... */ );
    }

    @AfterEach
    public void tearDown() {
        db.shutdown();
    }

}
```

Kotlin

```
class DataAccessIntegrationTestTemplate {

    private lateinit var db: EmbeddedDatabase

    @BeforeEach
    fun setUp() {
        // creates an HSQL in-memory database populated from default scripts
        // classpath:schema.sql and classpath:data.sql
        db = EmbeddedDatabaseBuilder()
                .generateUniqueName(true)
                .addDefaultScripts()
                .build()
    }

    @Test
    fun testDataAccess() {
        val template = JdbcTemplate(db)
        template.query( /* ... */)
    }

    @AfterEach
    fun tearDown() {
        db.shutdown()
    }
}
```

Generating Unique Names for Embedded Databases

Development teams often encounter errors with embedded databases if their test suite inadvertently attempts to recreate additional instances of the same database. This can happen quite easily if an XML configuration file or `@Configuration` class is responsible for creating an embedded database and the corresponding configuration is then reused across multiple testing scenarios within the same test suite (that is, within the same JVM process) — for example, integration tests against embedded databases whose `ApplicationContext` configuration differs only with regard to which bean definition profiles are active.

The root cause of such errors is the fact that Spring's `EmbeddedDatabaseFactory` (used internally by both the `<jdbc:embedded-database>` XML namespace element and the `EmbeddedDatabaseBuilder` for Java configuration) sets the name of the embedded database to `testdb` if not otherwise specified. For the case of `<jdbc:embedded-database>`, the embedded database is typically assigned a name equal to the bean's `id` (often, something like `dataSource`). Thus, subsequent attempts to create an embedded database do not result in a new database. Instead, the same JDBC connection URL is reused, and attempts to create a new embedded database actually point to an existing embedded database created from the same configuration.

To address this common issue, Spring Framework 4.2 provides support for generating unique names for embedded databases. To enable the use of generated names, use one of the following

options.

- `EmbeddedDatabaseFactory.setGenerateUniqueDatabaseName()`
- `EmbeddedDatabaseBuilder.generateUniqueName()`
- `<jdbc:embedded-database generate-name="true" … >`

Extending the Embedded Database Support

You can extend Spring JDBC embedded database support in two ways:

- Implement `EmbeddedDatabaseConfigurer` to support a new embedded database type.
- Implement `DataSourceFactory` to support a new `DataSource` implementation, such as a connection pool to manage embedded database connections.

We encourage you to contribute extensions to the Spring community at GitHub Issues.

4.3.10. Initializing a `DataSource`

The `org.springframework.jdbc.datasource.init` package provides support for initializing an existing `DataSource`. The embedded database support provides one option for creating and initializing a `DataSource` for an application. However, you may sometimes need to initialize an instance that runs on a server somewhere.

Initializing a Database by Using Spring XML

If you want to initialize a database and you can provide a reference to a `DataSource` bean, you can use the `initialize-database` tag in the `spring-jdbc` namespace:

```
<jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:com/foo/sql/db-schema.sql"/>
    <jdbc:script location="classpath:com/foo/sql/db-test-data.sql"/>
</jdbc:initialize-database>
```

The preceding example runs the two specified scripts against the database. The first script creates a schema, and the second populates tables with a test data set. The script locations can also be patterns with wildcards in the usual Ant style used for resources in Spring (for example, `classpath*:/com/foo/**/sql/*-data.sql`). If you use a pattern, the scripts are run in the lexical order of their URL or filename.

The default behavior of the database initializer is to unconditionally run the provided scripts. This may not always be what you want — for instance, if you run the scripts against a database that already has test data in it. The likelihood of accidentally deleting data is reduced by following the common pattern (shown earlier) of creating the tables first and then inserting the data. The first step fails if the tables already exist.

However, to gain more control over the creation and deletion of existing data, the XML namespace provides a few additional options. The first is a flag to switch the initialization on and off. You can set this according to the environment (such as pulling a boolean value from system properties or

from an environment bean). The following example gets a value from a system property:

```
<jdbc:initialize-database data-source="dataSource"
    enabled="#{systemProperties.INITIALIZE_DATABASE}"> ①
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

① Get the value for `enabled` from a system property called `INITIALIZE_DATABASE`.

The second option to control what happens with existing data is to be more tolerant of failures. To this end, you can control the ability of the initializer to ignore certain errors in the SQL it runs from the scripts, as the following example shows:

```
<jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS">
    <jdbc:script location="..."/>
</jdbc:initialize-database>
```

In the preceding example, we are saying that we expect that, sometimes, the scripts are run against an empty database, and there are some `DROP` statements in the scripts that would, therefore, fail. So failed SQL `DROP` statements will be ignored, but other failures will cause an exception. This is useful if your SQL dialect doesn't support `DROP … IF EXISTS` (or similar) but you want to unconditionally remove all test data before re-creating it. In that case the first script is usually a set of `DROP` statements, followed by a set of `CREATE` statements.

The `ignore-failures` option can be set to `NONE` (the default), `DROPS` (ignore failed drops), or `ALL` (ignore all failures).

Each statement should be separated by `;` or a new line if the `;` character is not present at all in the script. You can control that globally or script by script, as the following example shows:

```
<jdbc:initialize-database data-source="dataSource" separator="@@"> ①
    <jdbc:script location="classpath:com/myapp/sql/db-schema.sql" separator=";"/> ②
    <jdbc:script location="classpath:com/myapp/sql/db-test-data-1.sql"/>
    <jdbc:script location="classpath:com/myapp/sql/db-test-data-2.sql"/>
</jdbc:initialize-database>
```

① Set the separator scripts to `@@`.

② Set the separator for `db-schema.sql` to `;`.

In this example, the two `test-data` scripts use `@@` as statement separator and only the `db-schema.sql` uses `;`. This configuration specifies that the default separator is `@@` and overrides that default for the `db-schema` script.

If you need more control than you get from the XML namespace, you can use the `DataSourceInitializer` directly and define it as a component in your application.

Initialization of Other Components that Depend on the Database

A large class of applications (those that do not use the database until after the Spring context has started) can use the database initializer with no further complications. If your application is not one of those, you might need to read the rest of this section.

The database initializer depends on a `DataSource` instance and runs the scripts provided in its initialization callback (analogous to an `init-method` in an XML bean definition, a `@PostConstruct` method in a component, or the `afterPropertiesSet()` method in a component that implements `InitializingBean`). If other beans depend on the same data source and use the data source in an initialization callback, there might be a problem because the data has not yet been initialized. A common example of this is a cache that initializes eagerly and loads data from the database on application startup.

To get around this issue, you have two options: change your cache initialization strategy to a later phase or ensure that the database initializer is initialized first.

Changing your cache initialization strategy might be easy if the application is in your control and not otherwise. Some suggestions for how to implement this include:

- Make the cache initialize lazily on first usage, which improves application startup time.
- Have your cache or a separate component that initializes the cache implement `Lifecycle` or `SmartLifecycle`. When the application context starts, you can automatically start a `SmartLifecycle` by setting its `autoStartup` flag, and you can manually start a `Lifecycle` by calling `ConfigurableApplicationContext.start()` on the enclosing context.
- Use a Spring `ApplicationEvent` or similar custom observer mechanism to trigger the cache initialization. `ContextRefreshedEvent` is always published by the context when it is ready for use (after all beans have been initialized), so that is often a useful hook (this is how the `SmartLifecycle` works by default).

Ensuring that the database initializer is initialized first can also be easy. Some suggestions on how to implement this include:

- Rely on the default behavior of the Spring `BeanFactory`, which is that beans are initialized in registration order. You can easily arrange that by adopting the common practice of a set of `<import/>` elements in XML configuration that order your application modules and ensuring that the database and database initialization are listed first.
- Separate the `DataSource` and the business components that use it and control their startup order by putting them in separate `ApplicationContext` instances (for example, the parent context contains the `DataSource`, and the child context contains the business components). This structure is common in Spring web applications but can be more generally applied.

4.4. Data Access with R2DBC

R2DBC ("Reactive Relational Database Connectivity") is a community-driven specification effort to standardize access to SQL databases using reactive patterns.

4.4.1. Package Hierarchy

The Spring Framework's R2DBC abstraction framework consists of two different packages:

- `core`: The `org.springframework.r2dbc.core` package contains the `DatabaseClient` class plus a variety of related classes. See Using the R2DBC Core Classes to Control Basic R2DBC Processing and Error Handling.
- `connection`: The `org.springframework.r2dbc.connection` package contains a utility class for easy `ConnectionFactory` access and various simple `ConnectionFactory` implementations that you can use for testing and running unmodified R2DBC. See Controlling Database Connections.

4.4.2. Using the R2DBC Core Classes to Control Basic R2DBC Processing and Error Handling

This section covers how to use the R2DBC core classes to control basic R2DBC processing, including error handling. It includes the following topics:

- Using `DatabaseClient`
- Executing Statements
- Querying (`SELECT`)
- Updating (`INSERT`, `UPDATE`, and `DELETE`) with `DatabaseClient`
- Statement Filters
- Retrieving Auto-generated Keys

Using `DatabaseClient`

`DatabaseClient` is the central class in the R2DBC core package. It handles the creation and release of resources, which helps to avoid common errors, such as forgetting to close the connection. It performs the basic tasks of the core R2DBC workflow (such as statement creation and execution), leaving application code to provide SQL and extract results. The `DatabaseClient` class:

- Runs SQL queries
- Update statements and stored procedure calls
- Performs iteration over `Result` instances
- Catches R2DBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package. (See Consistent Exception Hierarchy.)

The client has a functional, fluent API using reactive types for declarative composition.

When you use the `DatabaseClient` for your code, you need only to implement `java.util.function` interfaces, giving them a clearly defined contract. Given a `Connection` provided by the `DatabaseClient` class, a `Function` callback creates a `Publisher`. The same is true for mapping functions that extract a `Row` result.

You can use `DatabaseClient` within a DAO implementation through direct instantiation with a `ConnectionFactory` reference, or you can configure it in a Spring IoC container and give it to DAOs as

a bean reference.

The simplest way to create a `DatabaseClient` object is through a static factory method, as follows:

Java

```
DatabaseClient client = DatabaseClient.create(connectionFactory);
```

Kotlin

```
val client = DatabaseClient.create(connectionFactory)
```

The `ConnectionFactory` should always be configured as a bean in the Spring IoC container.

The preceding method creates a `DatabaseClient` with default settings.

You can also obtain a `Builder` instance from `DatabaseClient.builder()`. You can customize the client by calling the following methods:

- `⋯.bindMarkers(⋯)`: Supply a specific `BindMarkersFactory` to configure named parameter to database bind marker translation.
- `⋯.executeFunction(⋯)`: Set the `ExecuteFunction` how `Statement` objects get run.
- `⋯.namedParameters(false)`: Disable named parameter expansion. Enabled by default.

Dialects are resolved by `BindMarkersFactoryResolver` from a `ConnectionFactory`, typically by inspecting `ConnectionFactoryMetadata`.
You can let Spring auto-discover your `BindMarkersFactory` by registering a class that implements `org.springframework.r2dbc.core.binding.BindMarkersFactoryResolver$BindMarkerFactoryProvider` through `META-INF/spring.factories`. `BindMarkersFactoryResolver` discovers bind marker provider implementations from the class path using Spring's `SpringFactoriesLoader`.

Currently supported databases are:

- H2
- MariaDB
- Microsoft SQL Server
- MySQL
- Postgres

All SQL issued by this class is logged at the `DEBUG` level under the category corresponding to the fully qualified class name of the client instance (typically `DefaultDatabaseClient`). Additionally, each execution registers a checkpoint in the reactive sequence to aid debugging.

The following sections provide some examples of `DatabaseClient` usage. These examples are not an exhaustive list of all of the functionality exposed by the `DatabaseClient`. See the attendant javadoc for that.

Executing Statements

`DatabaseClient` provides the basic functionality of running a statement. The following example shows what you need to include for minimal but fully functional code that creates a new table:

Java

```
Mono<Void> completion = client.sql("CREATE TABLE person (id VARCHAR(255) PRIMARY KEY,
name VARCHAR(255), age INTEGER);")
        .then();
```

Kotlin

```
client.sql("CREATE TABLE person (id VARCHAR(255) PRIMARY KEY, name VARCHAR(255), age
INTEGER);")
        .await()
```

`DatabaseClient` is designed for convenient, fluent usage. It exposes intermediate, continuation, and terminal methods at each stage of the execution specification. The preceding example above uses `then()` to return a completion `Publisher` that completes as soon as the query (or queries, if the SQL query contains multiple statements) completes.

`execute(…)` accepts either the SQL query string or a query `Supplier<String>` to defer the actual query creation until execution.

Querying (`SELECT`)

SQL queries can return values through `Row` objects or the number of affected rows. `DatabaseClient` can return the number of updated rows or the rows themselves, depending on the issued query.

The following query gets the `id` and `name` columns from a table:

Java

```
Mono<Map<String, Object>> first = client.sql("SELECT id, name FROM person")
        .fetch().first();
```

Kotlin

```
val first = client.sql("SELECT id, name FROM person")
        .fetch().awaitSingle()
```

The following query uses a bind variable:

Java

```
Mono<Map<String, Object>> first = client.sql("SELECT id, name FROM person WHERE
first_name = :fn")
        .bind("fn", "Joe")
        .fetch().first();
```

Kotlin

```
val first = client.sql("SELECT id, name FROM person WHERE WHERE first_name = :fn")
        .bind("fn", "Joe")
        .fetch().awaitSingle()
```

You might have noticed the use of `fetch()` in the example above. `fetch()` is a continuation operator that lets you specify how much data you want to consume.

Calling `first()` returns the first row from the result and discards remaining rows. You can consume data with the following operators:

- `first()` return the first row of the entire result. Its Kotlin Coroutine variant is named `awaitSingle()` for non-nullable return values and `awaitSingleOrNull()` if the value is optional.
- `one()` returns exactly one result and fails if the result contains more rows. Using Kotlin Coroutines, `awaitOne()` for exactly one value or `awaitOneOrNull()` if the value may be `null`.
- `all()` returns all rows of the result. When using Kotlin Coroutines, use `flow()`.
- `rowsUpdated()` returns the number of affected rows (`INSERT`/`UPDATE`/`DELETE` count). Its Kotlin Coroutine variant is named `awaitRowsUpdated()`.

Without specifying further mapping details, queries return tabular results as `Map` whose keys are case-insensitive column names that map to their column value.

You can take control over result mapping by supplying a `Function<Row, T>` that gets called for each `Row` so it can can return arbitrary values (singular values, collections and maps, and objects).

The following example extracts the `name` column and emits its value:

Java

```
Flux<String> names = client.sql("SELECT name FROM person")
        .map(row -> row.get("name", String.class))
        .all();
```

Kotlin

```
val names = client.sql("SELECT name FROM person")
        .map{ row: Row -> row.get("name", String.class) }
        .flow()
```

What about `null`?

Relational database results can contain `null` values. The Reactive Streams specification forbids the emission of `null` values. That requirement mandates proper `null` handling in the extractor function. While you can obtain `null` values from a `Row`, you must not emit a `null` value. You must wrap any `null` values in an object (for example, `Optional` for singular values) to make sure a `null` value is never returned directly by your extractor function.

Updating (`INSERT`, `UPDATE`, and `DELETE`) with `DatabaseClient`

The only difference of modifying statements is that these statements typically do not return tabular data so you use `rowsUpdated()` to consume results.

The following example shows an `UPDATE` statement that returns the number of updated rows:

Java

```
Mono<Integer> affectedRows = client.sql("UPDATE person SET first_name = :fn")
        .bind("fn", "Joe")
        .fetch().rowsUpdated();
```

Kotlin

```
val affectedRows = client.sql("UPDATE person SET first_name = :fn")
        .bind("fn", "Joe")
        .fetch().awaitRowsUpdated()
```

Binding Values to Queries

A typical application requires parameterized SQL statements to select or update rows according to some input. These are typically `SELECT` statements constrained by a `WHERE` clause or `INSERT` and `UPDATE` statements that accept input parameters. Parameterized statements bear the risk of SQL injection if parameters are not escaped properly. `DatabaseClient` leverages R2DBC's `bind` API to eliminate the risk of SQL injection for query parameters. You can provide a parameterized SQL statement with the `execute(…)` operator and bind parameters to the actual `Statement`. Your R2DBC driver then runs the statement by using prepared statements and parameter substitution.

Parameter binding supports two binding strategies:

- By Index, using zero-based parameter indexes.
- By Name, using the placeholder name.

The following example shows parameter binding for a query:

```
db.sql("INSERT INTO person (id, name, age) VALUES(:id, :name, :age)")
    .bind("id", "joe")
    .bind("name", "Joe")
    .bind("age", 34);
```

R2DBC Native Bind Markers

R2DBC uses database-native bind markers that depend on the actual database vendor. As an example, Postgres uses indexed markers, such as `$1`, `$2`, `$n`. Another example is SQL Server, which uses named bind markers prefixed with `@`.

This is different from JDBC, which requires `?` as bind markers. In JDBC, the actual drivers translate `?` bind markers to database-native markers as part of their statement execution.

Spring Framework's R2DBC support lets you use native bind markers or named bind markers with the `:name` syntax.

Named parameter support leverages a `BindMarkersFactory` instance to expand named parameters to native bind markers at the time of query execution, which gives you a certain degree of query portability across various database vendors.

The query-preprocessor unrolls named `Collection` parameters into a series of bind markers to remove the need of dynamic query creation based on the number of arguments. Nested object arrays are expanded to allow usage of (for example) select lists.

Consider the following query:

```
SELECT id, name, state FROM table WHERE (name, age) IN (('John', 35), ('Ann', 50))
```

The preceding query can be parametrized and run as follows:

Java

```
List<Object[]> tuples = new ArrayList<>();
tuples.add(new Object[] {"John", 35});
tuples.add(new Object[] {"Ann",  50});

client.sql("SELECT id, name, state FROM table WHERE (name, age) IN (:tuples)")
    .bind("tuples", tuples);
```

Kotlin

```
val tuples: MutableList<Array<Any>> = ArrayList()
tuples.add(arrayOf("John", 35))
tuples.add(arrayOf("Ann", 50))

client.sql("SELECT id, name, state FROM table WHERE (name, age) IN (:tuples)")
    .bind("tuples", tuples)
```

Usage of select lists is vendor-dependent.

The following example shows a simpler variant using `IN` predicates:

Java

```
client.sql("SELECT id, name, state FROM table WHERE age IN (:ages)")
    .bind("ages", Arrays.asList(35, 50));
```

Kotlin

```
val tuples: MutableList<Array<Any>> = ArrayList()
tuples.add(arrayOf("John", 35))
tuples.add(arrayOf("Ann", 50))

client.sql("SELECT id, name, state FROM table WHERE age IN (:ages)")
    .bind("tuples", arrayOf(35, 50))
```

R2DBC itself does not support Collection-like values. Nevertheless, expanding a given `List` in the example above works for named parameters in Spring's R2DBC support, e.g. for use in `IN` clauses as shown above. However, inserting or updating array-typed columns (e.g. in Postgres) requires an array type that is supported by the underlying R2DBC driver: typically a Java array, e.g. `String[]` to update a `text[]` column. Do not pass `Collection<String>` or the like as an array parameter.

Statement Filters

Sometimes it you need to fine-tune options on the actual `Statement` before it gets run. Register a `Statement` filter (`StatementFilterFunction`) through `DatabaseClient` to intercept and modify statements in their execution, as the following example shows:

Java

```
client.sql("INSERT INTO table (name, state) VALUES(:name, :state)")
    .filter((s, next) -> next.execute(s.returnGeneratedValues("id")))
    .bind("name", …)
    .bind("state", …);
```

Kotlin

```
client.sql("INSERT INTO table (name, state) VALUES(:name, :state)")
            .filter { s: Statement, next: ExecuteFunction ->
next.execute(s.returnGeneratedValues("id")) }
            .bind("name", …)
            .bind("state", …)
```

`DatabaseClient` exposes also simplified `filter(…)` overload accepting `Function<Statement, Statement>`:

Java

```
client.sql("INSERT INTO table (name, state) VALUES(:name, :state)")
    .filter(statement -> s.returnGeneratedValues("id"));

client.sql("SELECT id, name, state FROM table")
    .filter(statement -> s.fetchSize(25));
```

Kotlin

```
client.sql("INSERT INTO table (name, state) VALUES(:name, :state)")
    .filter { statement -> s.returnGeneratedValues("id") }

client.sql("SELECT id, name, state FROM table")
    .filter { statement -> s.fetchSize(25) }
```

`StatementFilterFunction` implementations allow filtering of the `Statement` and filtering of `Result` objects.

`DatabaseClient` Best Practices

Instances of the `DatabaseClient` class are thread-safe, once configured. This is important because it means that you can configure a single instance of a `DatabaseClient` and then safely inject this shared reference into multiple DAOs (or repositories). The `DatabaseClient` is stateful, in that it maintains a reference to a `ConnectionFactory`, but this state is not conversational state.

A common practice when using the `DatabaseClient` class is to configure a `ConnectionFactory` in your Spring configuration file and then dependency-inject that shared `ConnectionFactory` bean into your DAO classes. The `DatabaseClient` is created in the setter for the `ConnectionFactory`. This leads to DAOs that resemble the following:

Java

```
public class R2dbcCorporateEventDao implements CorporateEventDao {

    private DatabaseClient databaseClient;

    public void setConnectionFactory(ConnectionFactory connectionFactory) {
        this.databaseClient = DatabaseClient.create(connectionFactory);
    }

    // R2DBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

Kotlin

```
class R2dbcCorporateEventDao(connectionFactory: ConnectionFactory) : CorporateEventDao
{

    private val databaseClient = DatabaseClient.create(connectionFactory)

    // R2DBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

An alternative to explicit configuration is to use component-scanning and annotation support for dependency injection. In this case, you can annotate the class with `@Component` (which makes it a candidate for component-scanning) and annotate the `ConnectionFactory` setter method with `@Autowired`. The following example shows how to do so:

Java

```
@Component ①
public class R2dbcCorporateEventDao implements CorporateEventDao {

    private DatabaseClient databaseClient;

    @Autowired ②
    public void setConnectionFactory(ConnectionFactory connectionFactory) {
        this.databaseClient = DatabaseClient.create(connectionFactory); ③
    }

    // R2DBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

① Annotate the class with `@Component`.

② Annotate the `ConnectionFactory` setter method with `@Autowired`.

③ Create a new `DatabaseClient` with the `ConnectionFactory`.

Kotlin

```
@Component ①
class R2dbcCorporateEventDao(connectionFactory: ConnectionFactory) : CorporateEventDao
{ ②

    private val databaseClient = DatabaseClient(connectionFactory) ③

    // R2DBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

① Annotate the class with `@Component`.

② Constructor injection of the `ConnectionFactory`.

③ Create a new `DatabaseClient` with the `ConnectionFactory`.

Regardless of which of the above template initialization styles you choose to use (or not), it is seldom necessary to create a new instance of a `DatabaseClient` class each time you want to run SQL. Once configured, a `DatabaseClient` instance is thread-safe. If your application accesses multiple databases, you may want multiple `DatabaseClient` instances, which requires multiple `ConnectionFactory` and, subsequently, multiple differently configured `DatabaseClient` instances.

4.4.3. Retrieving Auto-generated Keys

`INSERT` statements may generate keys when inserting rows into a table that defines an auto-increment or identity column. To get full control over the column name to generate, simply register a `StatementFilterFunction` that requests the generated key for the desired column.

Java

```
Mono<Integer> generatedId = client.sql("INSERT INTO table (name, state) VALUES(:name,
:state)")
    .filter(statement -> s.returnGeneratedValues("id"))
        .map(row -> row.get("id", Integer.class))
        .first();

// generatedId emits the generated key once the INSERT statement has finished
```

Kotlin

```
val generatedId = client.sql("INSERT INTO table (name, state) VALUES(:name, :state)")
    .filter { statement -> s.returnGeneratedValues("id") }
        .map { row -> row.get("id", Integer.class) }
        .awaitOne()

// generatedId emits the generated key once the INSERT statement has finished
```

4.4.4. Controlling Database Connections

This section covers:

- Using `ConnectionFactory`
- Using `ConnectionFactoryUtils`
- Using `SingleConnectionFactory`
- Using `TransactionAwareConnectionFactoryProxy`
- Using `R2dbcTransactionManager`

Using `ConnectionFactory`

Spring obtains an R2DBC connection to the database through a `ConnectionFactory`. A `ConnectionFactory` is part of the R2DBC specification and is a common entry-point for drivers. It lets a container or a framework hide connection pooling and transaction management issues from the application code. As a developer, you need not know details about how to connect to the database. That is the responsibility of the administrator who sets up the `ConnectionFactory`. You most likely fill both roles as you develop and test code, but you do not necessarily have to know how the production data source is configured.

When you use Spring's R2DBC layer, you can can configure your own with a connection pool implementation provided by a third party. A popular implementation is R2DBC Pool (`r2dbc-pool`). Implementations in the Spring distribution are meant only for testing purposes and do not provide pooling.

To configure a `ConnectionFactory`:

1. Obtain a connection with `ConnectionFactory` as you typically obtain an R2DBC `ConnectionFactory`.
2. Provide an R2DBC URL (See the documentation for your driver for the correct value).

The following example shows how to configure a `ConnectionFactory`:

Java

```
ConnectionFactory factory =
ConnectionFactories.get("r2dbc:h2:mem:///test?options=DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE");
```

Kotlin

```
val factory = ConnectionFactories.get("r2dbc:h2:mem:///test?options=DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE");
```

Using `ConnectionFactoryUtils`

The `ConnectionFactoryUtils` class is a convenient and powerful helper class that provides `static` methods to obtain connections from `ConnectionFactory` and close connections (if necessary).

It supports subscriber `Context`-bound connections with, for example `R2dbcTransactionManager`.

Using `SingleConnectionFactory`

The `SingleConnectionFactory` class is an implementation of `DelegatingConnectionFactory` interface that wraps a single `Connection` that is not closed after each use.

If any client code calls `close` on the assumption of a pooled connection (as when using persistence tools), you should set the `suppressClose` property to `true`. This setting returns a close-suppressing proxy that wraps the physical connection. Note that you can no longer cast this to a native `Connection` or a similar object.

`SingleConnectionFactory` is primarily a test class and may be used for specific requirements such as pipelining if your R2DBC driver permits for such use. In contrast to a pooled `ConnectionFactory`, it reuses the same connection all the time, avoiding excessive creation of physical connections.

Using `TransactionAwareConnectionFactoryProxy`

`TransactionAwareConnectionFactoryProxy` is a proxy for a target `ConnectionFactory`. The proxy wraps that target `ConnectionFactory` to add awareness of Spring-managed transactions.

Using this class is required if you use a R2DBC client that is not integrated otherwise with Spring's R2DBC support. In this case, you can still use this client and, at the same time, have this client participating in Spring managed transactions. It is generally preferable to integrate a R2DBC client with proper access to `ConnectionFactoryUtils` for resource management.

See the `TransactionAwareConnectionFactoryProxy` javadoc for more details.

Using `R2dbcTransactionManager`

The `R2dbcTransactionManager` class is a `ReactiveTransactionManager` implementation for single R2DBC datasources. It binds an R2DBC connection from the specified connection factory to the subscriber `Context`, potentially allowing for one subscriber connection for each connection factory.

Application code is required to retrieve the R2DBC connection through `ConnectionFactoryUtils.getConnection(ConnectionFactory)`, instead of R2DBC's standard `ConnectionFactory.create()`.

All framework classes (such as `DatabaseClient`) use this strategy implicitly. If not used with this transaction manager, the lookup strategy behaves exactly like the common one. Thus, it can be used in any case.

The `R2dbcTransactionManager` class supports custom isolation levels that get applied to the connection.

4.5. Object Relational Mapping (ORM) Data Access

This section covers data access when you use Object Relational Mapping (ORM).

4.5.1. Introduction to ORM with Spring

The Spring Framework supports integration with the Java Persistence API (JPA) and supports native Hibernate for resource management, data access object (DAO) implementations, and transaction strategies. For example, for Hibernate, there is first-class support with several convenient IoC features that address many typical Hibernate integration issues. You can configure all of the supported features for OR (object relational) mapping tools through Dependency Injection. They can participate in Spring's resource and transaction management, and they comply with Spring's generic transaction and DAO exception hierarchies. The recommended integration style is to code DAOs against plain Hibernate or JPA APIs.

Spring adds significant enhancements to the ORM layer of your choice when you create data access applications. You can leverage as much of the integration support as you wish, and you should compare this integration effort with the cost and risk of building a similar infrastructure in-house. You can use much of the ORM support as you would a library, regardless of technology, because everything is designed as a set of reusable JavaBeans. ORM in a Spring IoC container facilitates configuration and deployment. Thus, most examples in this section show configuration inside a Spring container.

The benefits of using the Spring Framework to create your ORM DAOs include:

- **Easier testing.** Spring's IoC approach makes it easy to swap the implementations and configuration locations of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.
- **Common data access exceptions.** Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime `DataAccessException` hierarchy. This feature lets you handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches, throws, and exception declarations. You can still trap and handle exceptions as necessary. Remember that JDBC exceptions (including DB-specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.
- **General resource management.** Spring application contexts can handle the location and configuration of Hibernate `SessionFactory` instances, JPA `EntityManagerFactory` instances, JDBC `DataSource` instances, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy, and safe handling of persistence resources. For example, related code that uses Hibernate generally needs to use the same Hibernate `Session` to ensure efficiency and proper transaction handling. Spring makes it easy to create and bind a `Session` to the current thread transparently, by exposing a current `Session` through the Hibernate `SessionFactory`. Thus, Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction environment.
- **Integrated transaction management.** You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the `@Transactional` annotation or by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases, transaction semantics and exception handling (rollback and so on) are handled for you. As discussed in Resource and Transaction Management, you can also swap various

transaction managers, without affecting your ORM-related code. For example, you can swap between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. Additionally, JDBC-related code can fully integrate transactionally with the code you use to do ORM. This is useful for data access that is not suitable for ORM (such as batch processing and BLOB streaming) but that still needs to share common transactions with ORM operations.

For more comprehensive ORM support, including support for alternative database technologies such as MongoDB, you might want to check out the Spring Data suite of projects. If you are a JPA user, the Getting Started Accessing Data with JPA guide from https://spring.io provides a great introduction.

4.5.2. General ORM Integration Considerations

This section highlights considerations that apply to all ORM technologies. The Hibernate section provides more details and also show these features and configurations in a concrete context.

The major goal of Spring's ORM integration is clear application layering (with any data access and transaction technology) and for loose coupling of application objects — no more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. The goal is to have one simple and consistent approach to wiring up application objects, keeping them as reusable and free from container dependencies as possible. All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that need not be Spring-aware. In a typical Spring application, many important objects are JavaBeans: data access templates, data access objects, transaction managers, business services that use the data access objects and transaction managers, web view resolvers, web controllers that use the business services, and so on.

Resource and Transaction Management

Typical business applications are cluttered with repetitive resource management code. Many projects try to invent their own solutions, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates simple solutions for proper resource handling, namely IoC through templating in the case of JDBC and applying AOP interceptors for the ORM technologies.

The infrastructure provides proper resource handling and appropriate conversion of specific API exceptions to an unchecked infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct JDBC, the `JdbcTemplate` class mentioned in a previous section provides connection handling and proper conversion of `SQLException` to the `DataAccessException` hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. For ORM technologies, see the next section for how to get the same exception translation benefits.

When it comes to transaction management, the `JdbcTemplate` class hooks in to the Spring transaction support and supports both JTA and JDBC transactions, through respective Spring transaction managers. For the supported ORM technologies, Spring offers Hibernate and JPA support through the Hibernate and JPA transaction managers as well as JTA support. For details on

transaction support, see the Transaction Management chapter.

Exception Translation

When you use Hibernate or JPA in a DAO, you must decide how to handle the persistence technology's native exception classes. The DAO throws a subclass of a `HibernateException` or `PersistenceException`, depending on the technology. These exceptions are all runtime exceptions and do not have to be declared or caught. You may also have to deal with `IllegalArgumentException` and `IllegalStateException`. This means that callers can only treat exceptions as being generally fatal, unless they want to depend on the persistence technology's own exception structure. Catching specific causes (such as an optimistic locking failure) is not possible without tying the caller to the implementation strategy. This trade-off might be acceptable to applications that are strongly ORM-based or do not need any special exception treatment (or both). However, Spring lets exception translation be applied transparently through the `@Repository` annotation. The following examples (one for Java configuration and one for XML configuration) show how to do so:

Java

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}
```

Kotlin

```
@Repository
class ProductDaoImpl : ProductDao {

    // class body here...

}
```

```
<beans>

    <!-- Exception translation bean post processor -->
    <bean
class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor
"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

The postprocessor automatically looks for all exception translators (implementations of the `PersistenceExceptionTranslator` interface) and advises all beans marked with the `@Repository` annotation so that the discovered translators can intercept and apply the appropriate translation

on the thrown exceptions.

In summary, you can implement DAOs based on the plain persistence technology's API and annotations while still benefiting from Spring-managed transactions, dependency injection, and transparent exception conversion (if desired) to Spring's custom exception hierarchies.

4.5.3. Hibernate

We start with a coverage of Hibernate 5 in a Spring environment, using it to demonstrate the approach that Spring takes towards integrating OR mappers. This section covers many issues in detail and shows different variations of DAO implementations and transaction demarcation. Most of these patterns can be directly translated to all other supported ORM tools. The later sections in this chapter then cover the other ORM technologies and show brief examples.

As of Spring Framework 5.3, Spring requires Hibernate ORM 5.2+ for Spring's `HibernateJpaVendorAdapter` as well as for a native Hibernate `SessionFactory` setup. It is strongly recommended to go with Hibernate ORM 5.4 for a newly started application. For use with `HibernateJpaVendorAdapter`, Hibernate Search needs to be upgraded to 5.11.6.

`SessionFactory` Setup in a Spring Container

To avoid tying application objects to hard-coded resource lookups, you can define resources (such as a JDBC `DataSource` or a Hibernate `SessionFactory`) as beans in the Spring container. Application objects that need to access resources receive references to such predefined instances through bean references, as illustrated in the DAO definition in the next section.

The following excerpt from an XML application context definition shows how to set up a JDBC `DataSource` and a Hibernate `SessionFactory` on top of it:

```
<beans>

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>

    <bean id="mySessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.HSQLDialect
            </value>
        </property>
    </bean>

</beans>
```

Switching from a local Jakarta Commons DBCP `BasicDataSource` to a JNDI-located `DataSource` (usually managed by an application server) is only a matter of configuration, as the following example shows:

```
<beans>
    <jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>
</beans>
```

You can also access a JNDI-located `SessionFactory`, using Spring's `JndiObjectFactoryBean` / `<jee:jndi-lookup>` to retrieve and expose it. However, that is typically not common outside of an EJB context.

Spring also provides a `LocalSessionFactoryBuilder` variant, seamlessly integrating with `@Bean` style configuration and programmatic setup (no `FactoryBean` involved).

Both `LocalSessionFactoryBean` and `LocalSessionFactoryBuilder` support background bootstrapping, with Hibernate initialization running in parallel to the application bootstrap thread on a given bootstrap executor (such as a `SimpleAsyncTaskExecutor`). On `LocalSessionFactoryBean`, this is available through the `bootstrapExecutor` property. On the programmatic `LocalSessionFactoryBuilder`, there is an overloaded `buildSessionFactory` method that takes a bootstrap executor argument.

As of Spring Framework 5.1, such a native Hibernate setup can also expose a JPA `EntityManagerFactory` for standard JPA interaction next to native Hibernate access. See Native Hibernate Setup for JPA for details.

Implementing DAOs Based on the Plain Hibernate API

Hibernate has a feature called contextual sessions, wherein Hibernate itself manages one current `Session` per transaction. This is roughly equivalent to Spring's synchronization of one Hibernate `Session` per transaction. A corresponding DAO implementation resembles the following example, based on the plain Hibernate API:

Java

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list();
    }
}
```

Kotlin

```
class ProductDaoImpl(private val sessionFactory: SessionFactory) : ProductDao {

    fun loadProductsByCategory(category: String): Collection<*> {
        return sessionFactory.currentSession
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
                .list()
    }
}
```

This style is similar to that of the Hibernate reference documentation and examples, except for holding the `SessionFactory` in an instance variable. We strongly recommend such an instance-based setup over the old-school `static HibernateUtil` class from Hibernate's CaveatEmptor sample application. (In general, do not keep any resources in `static` variables unless absolutely necessary.)

The preceding DAO example follows the dependency injection pattern. It fits nicely into a Spring IoC container, as it would if coded against Spring's `HibernateTemplate`. You can also set up such a DAO in plain Java (for example, in unit tests). To do so, instantiate it and call `setSessionFactory(..)` with the desired factory reference. As a Spring bean definition, the DAO would resemble the following:

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

</beans>
```

The main advantage of this DAO style is that it depends on Hibernate API only. No import of any Spring class is required. This is appealing from a non-invasiveness perspective and may feel more natural to Hibernate developers.

However, the DAO throws plain `HibernateException` (which is unchecked, so it does not have to be declared or caught), which means that callers can treat exceptions only as being generally fatal — unless they want to depend on Hibernate's own exception hierarchy. Catching specific causes (such as an optimistic locking failure) is not possible without tying the caller to the implementation strategy. This trade off might be acceptable to applications that are strongly Hibernate-based, do not need any special exception treatment, or both.

Fortunately, Spring's `LocalSessionFactoryBean` supports Hibernate's `SessionFactory.getCurrentSession()` method for any Spring transaction strategy, returning the current Spring-managed transactional `Session`, even with `HibernateTransactionManager`. The standard behavior of that method remains to return the current `Session` associated with the ongoing JTA transaction, if any. This behavior applies regardless of whether you use Spring's `JtaTransactionManager`, EJB container managed transactions (CMTs), or JTA.

In summary, you can implement DAOs based on the plain Hibernate API, while still being able to participate in Spring-managed transactions.

Declarative Transaction Demarcation

We recommend that you use Spring's declarative transaction support, which lets you replace explicit transaction demarcation API calls in your Java code with an AOP transaction interceptor. You can configure this transaction interceptor in a Spring container by using either Java annotations or XML. This declarative transaction capability lets you keep business services free of repetitive transaction demarcation code and focus on adding business logic, which is the real value of your application.

Before you continue, we are strongly encourage you to read Declarative Transaction Management if you have not already done so.

You can annotate the service layer with `@Transactional` annotations and instruct the Spring container to find these annotations and provide transactional semantics for these annotated methods. The following example shows how to do so:

Java

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    @Transactional
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }

    @Transactional(readOnly = true)
    public List<Product> findAllProducts() {
        return this.productDao.findAllProducts();
    }
}
```

Kotlin

```
class ProductServiceImpl(private val productDao: ProductDao) : ProductService {

    @Transactional
    fun increasePriceOfAllProductsInCategory(category: String) {
        val productsToChange = productDao.loadProductsByCategory(category)
        // ...
    }

    @Transactional(readOnly = true)
    fun findAllProducts() = productDao.findAllProducts()
}
```

In the container, you need to set up the `PlatformTransactionManager` implementation (as a bean) and a `<tx:annotation-driven/>` entry, opting into `@Transactional` processing at runtime. The following example shows how to do so:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        https://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- SessionFactory, DataSource, etc. omitted -->

    <bean id="transactionManager"
            class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven/>

    <bean id="myProductService" class="product.SimpleProductService">
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>
```

Programmatic Transaction Demarcation

You can demarcate transactions in a higher level of the application, on top of lower-level data

access services that span any number of operations. Nor do restrictions exist on the implementation of the surrounding business service. It needs only a Spring `PlatformTransactionManager`. Again, the latter can come from anywhere, but preferably as a bean reference through a `setTransactionManager(..)` method. Also, the `productDAO` should be set by a `setProductDao(..)` method. The following pair of snippets show a transaction manager and a business service definition in a Spring application context and an example for a business method implementation:

```
<beans>

    <bean id="myTxManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="transactionManager" ref="myTxManager"/>
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>
```

Java

```
public class ProductServiceImpl implements ProductService {

    private TransactionTemplate transactionTemplate;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus status) {
                List productsToChange =
this.productDao.loadProductsByCategory(category);
                // do the price increase...
            }
        });
    }
}
```

Kotlin

```
class ProductServiceImpl(transactionManager: PlatformTransactionManager,
                         private val productDao: ProductDao) : ProductService {

    private val transactionTemplate = TransactionTemplate(transactionManager)

    fun increasePriceOfAllProductsInCategory(category: String) {
        transactionTemplate.execute {
            val productsToChange = productDao.loadProductsByCategory(category)
            // do the price increase...
        }
    }
}
```

Spring's `TransactionInterceptor` lets any checked application exception be thrown with the callback code, while `TransactionTemplate` is restricted to unchecked exceptions within the callback. `TransactionTemplate` triggers a rollback in case of an unchecked application exception or if the transaction is marked rollback-only by the application (by setting `TransactionStatus`). By default, `TransactionInterceptor` behaves the same way but allows configurable rollback policies per method.

Transaction Management Strategies

Both `TransactionTemplate` and `TransactionInterceptor` delegate the actual transaction handling to a `PlatformTransactionManager` instance (which can be a `HibernateTransactionManager` (for a single Hibernate `SessionFactory`) by using a `ThreadLocal Session` under the hood) or a `JtaTransactionManager` (delegating to the JTA subsystem of the container) for Hibernate applications. You can even use a custom `PlatformTransactionManager` implementation. Switching from native Hibernate transaction management to JTA (such as when facing distributed transaction requirements for certain deployments of your application) is only a matter of configuration. You can replace the Hibernate transaction manager with Spring's JTA transaction implementation. Both transaction demarcation and data access code work without changes, because they use the generic transaction management APIs.

For distributed transactions across multiple Hibernate session factories, you can combine `JtaTransactionManager` as a transaction strategy with multiple `LocalSessionFactoryBean` definitions. Each DAO then gets one specific `SessionFactory` reference passed into its corresponding bean property. If all underlying JDBC data sources are transactional container ones, a business service can demarcate transactions across any number of DAOs and any number of session factories without special regard, as long as it uses `JtaTransactionManager` as the strategy.

Both `HibernateTransactionManager` and `JtaTransactionManager` allow for proper JVM-level cache handling with Hibernate, without container-specific transaction manager lookup or a JCA connector (if you do not use EJB to initiate transactions).

`HibernateTransactionManager` can export the Hibernate JDBC `Connection` to plain JDBC access code for a specific `DataSource`. This ability allows for high-level transaction demarcation with mixed Hibernate and JDBC data access completely without JTA, provided you access only one database.

`HibernateTransactionManager` automatically exposes the Hibernate transaction as a JDBC transaction if you have set up the passed-in `SessionFactory` with a `DataSource` through the `dataSource` property of the `LocalSessionFactoryBean` class. Alternatively, you can specify explicitly the `DataSource` for which the transactions are supposed to be exposed through the `dataSource` property of the `HibernateTransactionManager` class.

Comparing Container-managed and Locally Defined Resources

You can switch between a container-managed JNDI `SessionFactory` and a locally defined one without having to change a single line of application code. Whether to keep resource definitions in the container or locally within the application is mainly a matter of the transaction strategy that you use. Compared to a Spring-defined local `SessionFactory`, a manually registered JNDI `SessionFactory` does not provide any benefits. Deploying a `SessionFactory` through Hibernate's JCA connector provides the added value of participating in the Jakarta EE server's management infrastructure, but does not add actual value beyond that.

Spring's transaction support is not bound to a container. When configured with any strategy other than JTA, transaction support also works in a stand-alone or test environment. Especially in the typical case of single-database transactions, Spring's single-resource local transaction support is a lightweight and powerful alternative to JTA. When you use local EJB stateless session beans to drive transactions, you depend both on an EJB container and on JTA, even if you access only a single database and use only stateless session beans to provide declarative transactions through container-managed transactions. Direct use of JTA programmatically also requires a Jakarta EE environment.

Spring-driven transactions can work as well with a locally defined Hibernate `SessionFactory` as they do with a local JDBC `DataSource`, provided they access a single database. Thus, you need only use Spring's JTA transaction strategy when you have distributed transaction requirements. A JCA connector requires container-specific deployment steps, and (obviously) JCA support in the first place. This configuration requires more work than deploying a simple web application with local resource definitions and Spring-driven transactions.

All things considered, if you do not use EJBs, stick with local `SessionFactory` setup and Spring's `HibernateTransactionManager` or `JtaTransactionManager`. You get all of the benefits, including proper transactional JVM-level caching and distributed transactions, without the inconvenience of container deployment. JNDI registration of a Hibernate `SessionFactory` through the JCA connector adds value only when used in conjunction with EJBs.

Spurious Application Server Warnings with Hibernate

In some JTA environments with very strict `XADataSource` implementations (currently some WebLogic Server and WebSphere versions), when Hibernate is configured without regard to the JTA transaction manager for that environment, spurious warnings or exceptions can show up in the application server log. These warnings or exceptions indicate that the connection being accessed is no longer valid or JDBC access is no longer valid, possibly because the transaction is no longer active. As an example, here is an actual exception from WebLogic:

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'. No
further JDBC access is allowed within this transaction.
```

Another common problem is a connection leak after JTA transactions, with Hibernate sessions (and potentially underlying JDBC connections) not getting closed properly.

You can resolve such issues by making Hibernate aware of the JTA transaction manager, to which it synchronizes (along with Spring). You have two options for doing this:

- Pass your Spring `JtaTransactionManager` bean to your Hibernate setup. The easiest way is a bean reference into the `jtaTransactionManager` property for your `LocalSessionFactoryBean` bean (see Hibernate Transaction Setup). Spring then makes the corresponding JTA strategies available to Hibernate.
- You may also configure Hibernate's JTA-related properties explicitly, in particular "hibernate.transaction.coordinator_class", "hibernate.connection.handling_mode" and potentially "hibernate.transaction.jta.platform" in your "hibernateProperties" on `LocalSessionFactoryBean` (see Hibernate's manual for details on those properties).

The remainder of this section describes the sequence of events that occur with and without Hibernate's awareness of the JTA `PlatformTransactionManager`.

When Hibernate is not configured with any awareness of the JTA transaction manager, the following events occur when a JTA transaction commits:

- The JTA transaction commits.
- Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so it is called back through an `afterCompletion` callback by the JTA transaction manager.
- Among other activities, this synchronization can trigger a callback by Spring to Hibernate, through Hibernate's `afterTransactionCompletion` callback (used to clear the Hibernate cache), followed by an explicit `close()` call on the Hibernate session, which causes Hibernate to attempt to `close()` the JDBC Connection.
- In some environments, this `Connection.close()` call then triggers the warning or error, as the application server no longer considers the `Connection` to be usable, because the transaction has already been committed.

When Hibernate is configured with awareness of the JTA transaction manager, the following events occur when a JTA transaction commits:

- The JTA transaction is ready to commit.
- Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so the transaction is called back through a `beforeCompletion` callback by the JTA transaction manager.
- Spring is aware that Hibernate itself is synchronized to the JTA transaction and behaves differently than in the previous scenario. In particular, it aligns with Hibernate's transactional resource management.
- The JTA transaction commits.

- Hibernate is synchronized to the JTA transaction, so the transaction is called back through an `afterCompletion` callback by the JTA transaction manager and can properly clear its cache.

4.5.4. JPA

The Spring JPA, available under the `org.springframework.orm.jpa` package, offers comprehensive support for the Java Persistence API in a manner similar to the integration with Hibernate while being aware of the underlying implementation in order to provide additional features.

Three Options for JPA Setup in a Spring Environment

The Spring JPA support offers three ways of setting up the JPA `EntityManagerFactory` that is used by the application to obtain an entity manager.

- Using `LocalEntityManagerFactoryBean`
- Obtaining an EntityManagerFactory from JNDI
- Using `LocalContainerEntityManagerFactoryBean`

Using `LocalEntityManagerFactoryBean`

You can use this option only in simple deployment environments such as stand-alone applications and integration tests.

The `LocalEntityManagerFactoryBean` creates an `EntityManagerFactory` suitable for simple deployment environments where the application uses only JPA for data access. The factory bean uses the JPA `PersistenceProvider` auto-detection mechanism (according to JPA's Java SE bootstrapping) and, in most cases, requires you to specify only the persistence unit name. The following XML example configures such a bean:

```
<beans>
    <bean id="myEmf"
class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
    </bean>
</beans>
```

This form of JPA deployment is the simplest and the most limited. You cannot refer to an existing JDBC `DataSource` bean definition, and no support for global transactions exists. Furthermore, weaving (byte-code transformation) of persistent classes is provider-specific, often requiring a specific JVM agent to be specified on startup. This option is sufficient only for stand-alone applications and test environments, for which the JPA specification is designed.

Obtaining an EntityManagerFactory from JNDI

You can use this option when deploying to a Jakarta EE server. Check your server's documentation on how to deploy a custom JPA provider into your server, allowing for a different provider than the server's default.

Obtaining an `EntityManagerFactory` from JNDI (for example in a Jakarta EE environment), is a matter of changing the XML configuration, as the following example shows:

```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

This action assumes standard Jakarta EE bootstrapping. The Jakarta EE server auto-detects persistence units (in effect, `META-INF/persistence.xml` files in application jars) and `persistence-unit-ref` entries in the Jakarta EE deployment descriptor (for example, `web.xml`) and defines environment naming context locations for those persistence units.

In such a scenario, the entire persistence unit deployment, including the weaving (byte-code transformation) of persistent classes, is up to the Jakarta EE server. The JDBC `DataSource` is defined through a JNDI location in the `META-INF/persistence.xml` file. `EntityManager` transactions are integrated with the server's JTA subsystem. Spring merely uses the obtained `EntityManagerFactory`, passing it on to application objects through dependency injection and managing transactions for the persistence unit (typically through `JtaTransactionManager`).

If you use multiple persistence units in the same application, the bean names of such JNDI-retrieved persistence units should match the persistence unit names that the application uses to refer to them (for example, in `@PersistenceUnit` and `@PersistenceContext` annotations).

Using `LocalContainerEntityManagerFactoryBean`

You can use this option for full JPA capabilities in a Spring-based application environment. This includes web containers such as Tomcat, stand-alone applications, and integration tests with sophisticated persistence requirements.

If you want to specifically configure a Hibernate setup, an immediate alternative is to set up a native Hibernate `LocalSessionFactoryBean` instead of a plain JPA `LocalContainerEntityManagerFactoryBean`, letting it interact with JPA access code as well as native Hibernate access code. See Native Hibernate setup for JPA interaction for details.

The `LocalContainerEntityManagerFactoryBean` gives full control over `EntityManagerFactory` configuration and is appropriate for environments where fine-grained customization is required. The `LocalContainerEntityManagerFactoryBean` creates a `PersistenceUnitInfo` instance based on the `persistence.xml` file, the supplied `dataSourceLookup` strategy, and the specified `loadTimeWeaver`. It is, thus, possible to work with custom data sources outside of JNDI and to control the weaving process. The following example shows a typical bean definition for a `LocalContainerEntityManagerFactoryBean`:

```
<beans>
    <bean id="myEmf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="someDataSource"/>
        <property name="loadTimeWeaver">
            <bean
class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
    </bean>
</beans>
```

The following example shows a typical `persistence.xml` file:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
    <persistence-unit name="myUnit" transaction-type="RESOURCE_LOCAL">
        <mapping-file>META-INF/orm.xml</mapping-file>
        <exclude-unlisted-classes/>
    </persistence-unit>
</persistence>
```

The `<exclude-unlisted-classes/>` shortcut indicates that no scanning for annotated entity classes is supposed to occur. An explicit 'true' value (`<exclude-unlisted-classes>true</exclude-unlisted-classes/>`) also means no scan. `<exclude-unlisted-classes>false</exclude-unlisted-classes/>` does trigger a scan. However, we recommend omitting the `exclude-unlisted-classes` element if you want entity class scanning to occur.

Using the `LocalContainerEntityManagerFactoryBean` is the most powerful JPA setup option, allowing for flexible local configuration within the application. It supports links to an existing JDBC `DataSource`, supports both local and global transactions, and so on. However, it also imposes requirements on the runtime environment, such as the availability of a weaving-capable class loader if the persistence provider demands byte-code transformation.

This option may conflict with the built-in JPA capabilities of a Jakarta EE server. In a full Jakarta EE environment, consider obtaining your `EntityManagerFactory` from JNDI. Alternatively, specify a custom `persistenceXmlLocation` on your `LocalContainerEntityManagerFactoryBean` definition (for example, META-INF/my-persistence.xml) and include only a descriptor with that name in your application jar files. Because the Jakarta EE server looks only for default `META-INF/persistence.xml` files, it ignores such custom persistence units and, hence, avoids conflicts with a Spring-driven JPA setup upfront. (This applies to Resin 3.1, for example.)

When is load-time weaving required?

Not all JPA providers require a JVM agent. Hibernate is an example of one that does not. If your provider does not require an agent or you have other alternatives, such as applying enhancements at build time through a custom compiler or an Ant task, you should not use the load-time weaver.

The `LoadTimeWeaver` interface is a Spring-provided class that lets JPA `ClassTransformer` instances be plugged in a specific manner, depending on whether the environment is a web container or application server. Hooking `ClassTransformers` through an agent is typically not efficient. The agents work against the entire virtual machine and inspect every class that is loaded, which is usually undesirable in a production server environment.

Spring provides a number of `LoadTimeWeaver` implementations for various environments, letting `ClassTransformer` instances be applied only for each class loader and not for each VM.

See the Spring configuration in the AOP chapter for more insight regarding the `LoadTimeWeaver` implementations and their setup, either generic or customized to various platforms (such as Tomcat, JBoss and WebSphere).

As described in Spring configuration, you can configure a context-wide `LoadTimeWeaver` by using the `@EnableLoadTimeWeaving` annotation or the `context:load-time-weaver` XML element. Such a global weaver is automatically picked up by all JPA `LocalContainerEntityManagerFactoryBean` instances. The following example shows the preferred way of setting up a load-time weaver, delivering auto-detection of the platform (e.g. Tomcat's weaving-capable class loader or Spring's JVM agent) and automatic propagation of the weaver to all weaver-aware beans:

```
<context:load-time-weaver/>
<bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
</bean>
```

However, you can, if needed, manually specify a dedicated weaver through the `loadTimeWeaver` property, as the following example shows:

```
<bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="loadTimeWeaver">
        <bean
class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
    </property>
</bean>
```

No matter how the LTW is configured, by using this technique, JPA applications relying on instrumentation can run in the target platform (for example, Tomcat) without needing an agent.

This is especially important when the hosting applications rely on different JPA implementations, because the JPA transformers are applied only at the class-loader level and are, thus, isolated from each other.

Dealing with Multiple Persistence Units

For applications that rely on multiple persistence units locations (stored in various JARS in the classpath, for example), Spring offers the `PersistenceUnitManager` to act as a central repository and to avoid the persistence units discovery process, which can be expensive. The default implementation lets multiple locations be specified. These locations are parsed and later retrieved through the persistence unit name. (By default, the classpath is searched for `META-INF/persistence.xml` files.) The following example configures multiple locations:

```
<bean id="pum"
class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
    <property name="persistenceXmlLocations">
        <list>
            <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
            <value>classpath:/my/package/**/custom-persistence.xml</value>
            <value>classpath*:META-INF/persistence.xml</value>
        </list>
    </property>
    <property name="dataSources">
        <map>
            <entry key="localDataSource" value-ref="local-db"/>
            <entry key="remoteDataSource" value-ref="remote-db"/>
        </map>
    </property>
    <!-- if no datasource is specified, use this one -->
    <property name="defaultDataSource" ref="remoteDataSource"/>
</bean>

<bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitManager" ref="pum"/>
    <property name="persistenceUnitName" value="myCustomUnit"/>
</bean>
```

The default implementation allows customization of the `PersistenceUnitInfo` instances (before they are fed to the JPA provider) either declaratively (through its properties, which affect all hosted units) or programmatically (through the `PersistenceUnitPostProcessor`, which allows persistence unit selection). If no `PersistenceUnitManager` is specified, one is created and used internally by `LocalContainerEntityManagerFactoryBean`.

Background Bootstrapping

`LocalContainerEntityManagerFactoryBean` supports background bootstrapping through the `bootstrapExecutor` property, as the following example shows:

```
<bean id="emf"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="bootstrapExecutor">
        <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>
    </property>
</bean>
```

The actual JPA provider bootstrapping is handed off to the specified executor and then, running in parallel, to the application bootstrap thread. The exposed `EntityManagerFactory` proxy can be injected into other application components and is even able to respond to `EntityManagerFactoryInfo` configuration inspection. However, once the actual JPA provider is being accessed by other components (for example, calling `createEntityManager`), those calls block until the background bootstrapping has completed. In particular, when you use Spring Data JPA, make sure to set up deferred bootstrapping for its repositories as well.

Implementing DAOs Based on JPA: `EntityManagerFactory` and `EntityManager`

Although `EntityManagerFactory` instances are thread-safe, `EntityManager` instances are not. The injected JPA `EntityManager` behaves like an `EntityManager` fetched from an application server's JNDI environment, as defined by the JPA specification. It delegates all calls to the current transactional `EntityManager`, if any. Otherwise, it falls back to a newly created `EntityManager` per operation, in effect making its usage thread-safe.

It is possible to write code against the plain JPA without any Spring dependencies, by using an injected `EntityManagerFactory` or `EntityManager`. Spring can understand the `@PersistenceUnit` and `@PersistenceContext` annotations both at the field and the method level if a `PersistenceAnnotationBeanPostProcessor` is enabled. The following example shows a plain JPA DAO implementation that uses the `@PersistenceUnit` annotation:

Java

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
            Query query = em.createQuery("from Product as p where p.category = ?1");
            query.setParameter(1, category);
            return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

Kotlin

```
class ProductDaoImpl : ProductDao {

    private lateinit var emf: EntityManagerFactory

    @PersistenceUnit
    fun setEntityManagerFactory(emf: EntityManagerFactory) {
        this.emf = emf
    }

    fun loadProductsByCategory(category: String): Collection<*> {
        val em = this.emf.createEntityManager()
        val query = em.createQuery("from Product as p where p.category = ?1");
        query.setParameter(1, category);
        return query.resultList;
    }
}
```

The preceding DAO has no dependency on Spring and still fits nicely into a Spring application context. Moreover, the DAO takes advantage of annotations to require the injection of the default `EntityManagerFactory`, as the following example bean definition shows:

```
<beans>

    <!-- bean post-processor for JPA annotations -->
    <bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

As an alternative to explicitly defining a `PersistenceAnnotationBeanPostProcessor`, consider using the Spring `context:annotation-config` XML element in your application context configuration. Doing so automatically registers all Spring standard post-processors for annotation-based configuration, including `CommonAnnotationBeanPostProcessor` and so on.

Consider the following example:

```
<beans>

    <!-- post-processors for all standard config annotations -->
    <context:annotation-config/>

    <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

The main problem with such a DAO is that it always creates a new `EntityManager` through the factory. You can avoid this by requesting a transactional `EntityManager` (also called a “shared EntityManager” because it is a shared, thread-safe proxy for the actual transactional EntityManager) to be injected instead of the factory. The following example shows how to do so:

Java

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
        Query query = em.createQuery("from Product as p where p.category =
:category");
        query.setParameter("category", category);
        return query.getResultList();
    }
}
```

Kotlin

```
class ProductDaoImpl : ProductDao {

    @PersistenceContext
    private lateinit var em: EntityManager

    fun loadProductsByCategory(category: String): Collection<*> {
        val query = em.createQuery("from Product as p where p.category = :category")
        query.setParameter("category", category)
        return query.resultList
    }
}
```

The `@PersistenceContext` annotation has an optional attribute called `type`, which defaults to `PersistenceContextType.TRANSACTION`. You can use this default to receive a shared `EntityManager` proxy. The alternative, `PersistenceContextType.EXTENDED`, is a completely different affair. This results in a so-called extended `EntityManager`, which is not thread-safe and, hence, must not be used in a concurrently accessed component, such as a Spring-managed singleton bean. Extended `EntityManager` instances are only supposed to be used in stateful components that, for example, reside in a session, with the lifecycle of the `EntityManager` not tied to a current transaction but rather being completely up to the application.

Method- and field-level Injection

You can apply annotations that indicate dependency injections (such as `@PersistenceUnit` and `@PersistenceContext`) on field or methods inside a class — hence the expressions "method-level injection" and "field-level injection". Field-level annotations are concise and easier to use while method-level annotations allow for further processing of the injected dependency. In both cases, the member visibility (public, protected, or private) does not matter.

What about class-level annotations?

On the Jakarta EE platform, they are used for dependency declaration and not for resource injection.

The injected `EntityManager` is Spring-managed (aware of the ongoing transaction). Even though the new DAO implementation uses method-level injection of an `EntityManager` instead of an `EntityManagerFactory`, no change is required in the application context XML, due to annotation usage.

The main advantage of this DAO style is that it depends only on the Java Persistence API. No import of any Spring class is required. Moreover, as the JPA annotations are understood, the injections are applied automatically by the Spring container. This is appealing from a non-invasiveness perspective and can feel more natural to JPA developers.

Spring-driven JPA transactions

We strongly encourage you to read Declarative Transaction Management, if you have not already done so, to get more detailed coverage of Spring's declarative transaction support.

The recommended strategy for JPA is local transactions through JPA's native transaction support. Spring's `JpaTransactionManager` provides many capabilities known from local JDBC transactions (such as transaction-specific isolation levels and resource-level read-only optimizations) against any regular JDBC connection pool (no XA requirement).

Spring JPA also lets a configured `JpaTransactionManager` expose a JPA transaction to JDBC access code that accesses the same `DataSource`, provided that the registered `JpaDialect` supports retrieval of the underlying JDBC `Connection`. Spring provides dialects for the EclipseLink and Hibernate JPA implementations. See the next section for details on the `JpaDialect` mechanism.

As an immediate alternative, Spring's native `HibernateTransactionManager` is capable of interacting with JPA access code, adapting to several Hibernate specifics and providing JDBC interaction. This makes particular sense in combination with `LocalSessionFactoryBean` setup. See Native Hibernate Setup for JPA Interaction for details.

Understanding `JpaDialect` and `JpaVendorAdapter`

As an advanced feature, `JpaTransactionManager` and subclasses of `AbstractEntityManagerFactoryBean` allow a custom `JpaDialect` to be passed into the `jpaDialect` bean property. A `JpaDialect` implementation can enable the following advanced features supported by Spring, usually in a vendor-specific manner:

- Applying specific transaction semantics (such as custom isolation level or transaction timeout)
- Retrieving the transactional JDBC `Connection` (for exposure to JDBC-based DAOs)
- Advanced translation of `PersistenceExceptions` to Spring `DataAccessExceptions`

This is particularly valuable for special transaction semantics and for advanced translation of exception. The default implementation (`DefaultJpaDialect`) does not provide any special abilities and, if the features listed earlier are required, you have to specify the appropriate dialect.

As an even broader provider adaptation facility primarily for Spring's full-featured `LocalContainerEntityManagerFactoryBean` setup, `JpaVendorAdapter` combines the capabilities of `JpaDialect` with other provider-specific defaults. Specifying a `HibernateJpaVendorAdapter` or `EclipseLinkJpaVendorAdapter` is the most convenient way of auto-configuring an `EntityManagerFactory` setup for Hibernate or EclipseLink, respectively. Note that those provider adapters are primarily designed for use with Spring-driven transaction management (that is, for use with `JpaTransactionManager`).

See the `JpaDialect` and `JpaVendorAdapter` javadoc for more details of its operations and how they are used within Spring's JPA support.

Setting up JPA with JTA Transaction Management

As an alternative to `JpaTransactionManager`, Spring also allows for multi-resource transaction coordination through JTA, either in a Jakarta EE environment or with a stand-alone transaction coordinator, such as Atomikos. Aside from choosing Spring's `JtaTransactionManager` instead of `JpaTransactionManager`, you need to take few further steps:

- The underlying JDBC connection pools need to be XA-capable and be integrated with your transaction coordinator. This is usually straightforward in a Jakarta EE environment, exposing a different kind of `DataSource` through JNDI. See your application server documentation for details. Analogously, a standalone transaction coordinator usually comes with special XA-integrated `DataSource` variants. Again, check its documentation.
- The JPA `EntityManagerFactory` setup needs to be configured for JTA. This is provider-specific, typically through special properties to be specified as `jpaProperties` on `LocalContainerEntityManagerFactoryBean`. In the case of Hibernate, these properties are even version-specific. See your Hibernate documentation for details.
- Spring's `HibernateJpaVendorAdapter` enforces certain Spring-oriented defaults, such as the connection release mode, `on-close`, which matches Hibernate's own default in Hibernate 5.0 but not any more in Hibernate 5.1+. For a JTA setup, make sure to declare your persistence unit transaction type as "JTA". Alternatively, set Hibernate 5.2's `hibernate.connection.handling_mode` property to `DELAYED_ACQUISITION_AND_RELEASE_AFTER_STATEMENT` to restore Hibernate's own default. See Spurious Application Server Warnings with Hibernate for related notes.
- Alternatively, consider obtaining the `EntityManagerFactory` from your application server itself (that is, through a JNDI lookup instead of a locally declared `LocalContainerEntityManagerFactoryBean`). A server-provided `EntityManagerFactory` might require special definitions in your server configuration (making the deployment less portable) but is set up for the server's JTA environment.

Native Hibernate Setup and Native Hibernate Transactions for JPA Interaction

A native `LocalSessionFactoryBean` setup in combination with `HibernateTransactionManager` allows for interaction with `@PersistenceContext` and other JPA access code. A Hibernate `SessionFactory` natively implements JPA's `EntityManagerFactory` interface now and a Hibernate `Session` handle natively is a JPA `EntityManager`. Spring's JPA support facilities automatically detect native Hibernate sessions.

Such native Hibernate setup can, therefore, serve as a replacement for a standard JPA `LocalContainerEntityManagerFactoryBean` and `JpaTransactionManager` combination in many scenarios, allowing for interaction with `SessionFactory.getCurrentSession()` (and also `HibernateTemplate`) next to `@PersistenceContext EntityManager` within the same local transaction. Such a setup also provides stronger Hibernate integration and more configuration flexibility, because it is not constrained by JPA bootstrap contracts.

You do not need `HibernateJpaVendorAdapter` configuration in such a scenario, since Spring's native Hibernate setup provides even more features (for example, custom Hibernate Integrator setup, Hibernate 5.3 bean container integration, and stronger optimizations for read-only transactions). Last but not least, you can also express native Hibernate setup through `LocalSessionFactoryBuilder`, seamlessly integrating with `@Bean` style configuration (no `FactoryBean` involved).

`LocalSessionFactoryBean` and `LocalSessionFactoryBuilder` support background bootstrapping, just as the JPA `LocalContainerEntityManagerFactoryBean` does. See Background Bootstrapping for an introduction.

On `LocalSessionFactoryBean`, this is available through the `bootstrapExecutor` property. On the programmatic `LocalSessionFactoryBuilder`, an overloaded `buildSessionFactory` method takes a bootstrap executor argument.

4.6. Marshalling XML by Using Object-XML Mappers

4.6.1. Introduction

This chapter, describes Spring's Object-XML Mapping support. Object-XML Mapping (O-X mapping for short) is the act of converting an XML document to and from an object. This conversion process is also known as XML Marshalling, or XML Serialization. This chapter uses these terms interchangeably.

Within the field of O-X mapping, a marshaller is responsible for serializing an object (graph) to XML. In similar fashion, an unmarshaller deserializes the XML to an object graph. This XML can take the form of a DOM document, an input or output stream, or a SAX handler.

Some of the benefits of using Spring for your O/X mapping needs are:

- Ease of configuration
- Consistent Interfaces
- Consistent Exception Hierarchy

Ease of configuration

Spring's bean factory makes it easy to configure marshallers, without needing to construct JAXB context, JiBX binding factories, and so on. You can configure the marshallers as you would any other bean in your application context. Additionally, XML namespace-based configuration is available for a number of marshallers, making the configuration even simpler.

Consistent Interfaces

Spring's O-X mapping operates through two global interfaces: `Marshaller` and `Unmarshaller`. These abstractions let you switch O-X mapping frameworks with relative ease, with little or no change required on the classes that do the marshalling. This approach has the additional benefit of making it possible to do XML marshalling with a mix-and-match approach (for example, some marshalling performed using JAXB and some by XStream) in a non-intrusive fashion, letting you use the strength of each technology.

Consistent Exception Hierarchy

Spring provides a conversion from exceptions from the underlying O-X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. These runtime exceptions wrap the original exception so that no information is lost.

4.6.2. `Marshaller` and `Unmarshaller`

As stated in the introduction, a marshaller serializes an object to XML, and an unmarshaller deserializes XML stream to an object. This section describes the two Spring interfaces used for this purpose.

Understanding `Marshaller`

Spring abstracts all marshalling operations behind the `org.springframework.oxm.Marshaller` interface, the main method of which follows:

```
public interface Marshaller {

    /**
     * Marshal the object graph with the given root into the provided Result.
     */
    void marshal(Object graph, Result result) throws XmlMappingException, IOException;
}
```

The `Marshaller` interface has one main method, which marshals the given object to a given `javax.xml.transform.Result`. The result is a tagging interface that basically represents an XML output abstraction. Concrete implementations wrap various XML representations, as the following table indicates:

Result implementation	Wraps XML representation
`DOMResult`	`org.w3c.dom.Node`
`SAXResult`	`org.xml.sax.ContentHandler`
`StreamResult`	`java.io.File`, `java.io.OutputStream`, or `java.io.Writer`

Although the `marshal()` method accepts a plain object as its first parameter, most `Marshaller` implementations cannot handle arbitrary objects. Instead, an object class must be mapped in a mapping file, be marked with an annotation, be registered with the marshaller, or have a common base class. Refer to the later sections in this chapter to determine how your O-X technology manages this.

Understanding `Unmarshaller`

Similar to the `Marshaller`, we have the `org.springframework.oxm.Unmarshaller` interface, which the following listing shows:

```
public interface Unmarshaller {

    /**
     * Unmarshal the given provided Source into an object graph.
     */
    Object unmarshal(Source source) throws XmlMappingException, IOException;
}
```

This interface also has one method, which reads from the given `javax.xml.transform.Source` (an XML input abstraction) and returns the object read. As with `Result`, `Source` is a tagging interface that has three concrete implementations. Each wraps a different XML representation, as the following table indicates:

Source implementation	Wraps XML representation
`DOMSource`	`org.w3c.dom.Node`
`SAXSource`	`org.xml.sax.InputSource`, and `org.xml.sax.XMLReader`
`StreamSource`	`java.io.File`, `java.io.InputStream`, or `java.io.Reader`

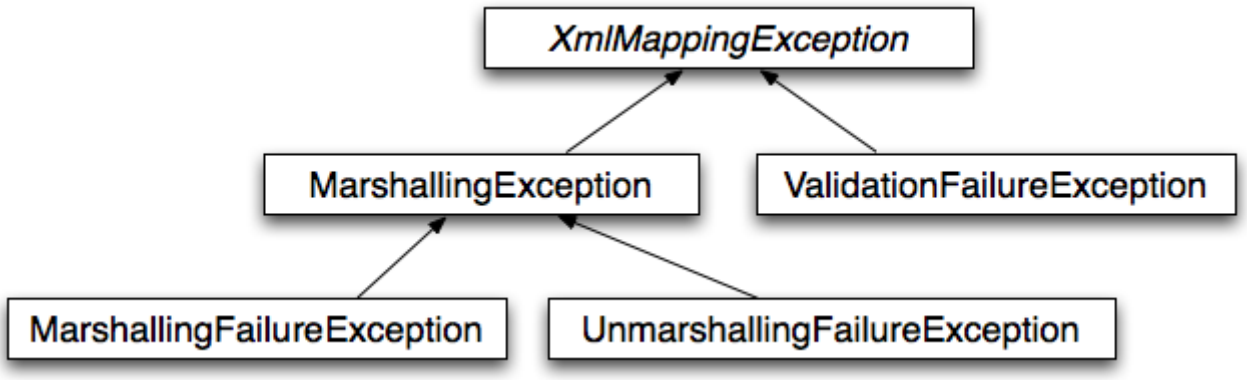
Even though there are two separate marshalling interfaces (`Marshaller` and `Unmarshaller`), all implementations in Spring-WS implement both in one class. This means that you can wire up one marshaller class and refer to it both as a marshaller and as an unmarshaller in your `applicationContext.xml`.

Understanding `XmlMappingException`

Spring converts exceptions from the underlying O-X mapping tool to its own exception hierarchy with the `XmlMappingException` as the root exception. These runtime exceptions wrap the original exception so that no information will be lost.

Additionally, the `MarshallingFailureException` and `UnmarshallingFailureException` provide a distinction between marshalling and unmarshalling operations, even though the underlying O-X mapping tool does not do so.

The O-X Mapping exception hierarchy is shown in the following figure:



4.6.3. Using `Marshaller` and `Unmarshaller`

You can use Spring's OXM for a wide variety of situations. In the following example, we use it to marshal the settings of a Spring-managed application as an XML file. In the following example, we use a simple JavaBean to represent the settings:

Java

```
public class Settings {

    private boolean fooEnabled;

    public boolean isFooEnabled() {
        return fooEnabled;
    }

    public void setFooEnabled(boolean fooEnabled) {
        this.fooEnabled = fooEnabled;
    }
}
```

Kotlin

```
class Settings {
    var isFooEnabled: Boolean = false
}
```

The application class uses this bean to store its settings. Besides a main method, the class has two methods: `saveSettings()` saves the settings bean to a file named `settings.xml`, and `loadSettings()` loads these settings again. The following `main()` method constructs a Spring application context and calls these two methods:

Java

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;

public class Application {

    private static final String FILE_NAME = "settings.xml";
    private Settings settings = new Settings();
    private Marshaller marshaller;
    private Unmarshaller unmarshaller;

    public void setMarshaller(Marshaller marshaller) {
        this.marshaller = marshaller;
    }

    public void setUnmarshaller(Unmarshaller unmarshaller) {
        this.unmarshaller = unmarshaller;
    }

    public void saveSettings() throws IOException {
        try (FileOutputStream os = new FileOutputStream(FILE_NAME)) {
            this.marshaller.marshal(settings, new StreamResult(os));
        }
    }

    public void loadSettings() throws IOException {
        try (FileInputStream is = new FileInputStream(FILE_NAME)) {
            this.settings = (Settings) this.unmarshaller.unmarshal(new
StreamSource(is));
        }
    }

    public static void main(String[] args) throws IOException {
        ApplicationContext appContext =
                new ClassPathXmlApplicationContext("applicationContext.xml");
        Application application = (Application) appContext.getBean("application");
        application.saveSettings();
        application.loadSettings();
    }
}
```

Kotlin

```
class Application {

    lateinit var marshaller: Marshaller

    lateinit var unmarshaller: Unmarshaller

    fun saveSettings() {
        FileOutputStream(FILE_NAME).use { outputStream -> marshaller.marshal(settings,
StreamResult(outputStream)) }
    }

    fun loadSettings() {
        FileInputStream(FILE_NAME).use { inputStream -> settings =
unmarshaller.unmarshal(StreamSource(inputStream)) as Settings }
    }
}

private const val FILE_NAME = "settings.xml"

fun main(args: Array<String>) {
    val appContext = ClassPathXmlApplicationContext("applicationContext.xml")
    val application = appContext.getBean("application") as Application
    application.saveSettings()
    application.loadSettings()
}
```

The `Application` requires both a `marshaller` and an `unmarshaller` property to be set. We can do so by using the following `applicationContext.xml`:

```
<beans>
    <bean id="application" class="Application">
        <property name="marshaller" ref="xstreamMarshaller" />
        <property name="unmarshaller" ref="xstreamMarshaller" />
    </bean>
    <bean id="xstreamMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller"/>
</beans>
```

This application context uses XStream, but we could have used any of the other marshaller instances described later in this chapter. Note that, by default, XStream does not require any further configuration, so the bean definition is rather simple. Also note that the `XStreamMarshaller` implements both `Marshaller` and `Unmarshaller`, so we can refer to the `xstreamMarshaller` bean in both the `marshaller` and `unmarshaller` property of the application.

This sample application produces the following `settings.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings foo-enabled="false"/>
```

4.6.4. XML Configuration Namespace

You can configure marshallers more concisely by using tags from the OXM namespace. To make these tags available, you must first reference the appropriate schema in the preamble of the XML configuration file. The following example shows how to do so:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:oxm="http://www.springframework.org/schema/oxm" ①
xsi:schemaLocation="http://www.springframework.org/schema/beans
  https://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/oxm
https://www.springframework.org/schema/oxm/spring-oxm.xsd"> ②
```

① Reference the `oxm` schema.

② Specify the `oxm` schema location.

The schema makes the following elements available:

- `jaxb2-marshaller`
- `jibx-marshaller`

Each tag is explained in its respective marshaller's section. As an example, though, the configuration of a JAXB2 marshaller might resemble the following:

```
<oxm:jaxb2-marshaller id="marshaller"
contextPath="org.springframework.ws.samples.airline.schema"/>
```

4.6.5. JAXB

The JAXB binding compiler translates a W3C XML Schema into one or more Java classes, a `jaxb.properties` file, and possibly some resource files. JAXB also offers a way to generate a schema from annotated Java classes.

Spring supports the JAXB 2.0 API as XML marshalling strategies, following the `Marshaller` and `Unmarshaller` interfaces described in `Marshaller` and `Unmarshaller`. The corresponding integration classes reside in the `org.springframework.oxm.jaxb` package.

Using `Jaxb2Marshaller`

The `Jaxb2Marshaller` class implements both of Spring's `Marshaller` and `Unmarshaller` interfaces. It requires a context path to operate. You can set the context path by setting the `contextPath` property.

The context path is a list of colon-separated Java package names that contain schema derived classes. It also offers a `classesToBeBound` property, which allows you to set an array of classes to be supported by the marshaller. Schema validation is performed by specifying one or more schema resources to the bean, as the following example shows:

```
<beans>
    <bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <list>
                <value>org.springframework.oxm.jaxb.Flight</value>
                <value>org.springframework.oxm.jaxb.Flights</value>
            </list>
        </property>
        <property name="schema" value="classpath:org/springframework/oxm/schema.xsd"/>
    </bean>

    ...

</beans>
```

XML Configuration Namespace

The `jaxb2-marshaller` element configures a `org.springframework.oxm.jaxb.Jaxb2Marshaller`, as the following example shows:

```
<oxm:jaxb2-marshaller id="marshaller"
contextPath="org.springframework.ws.samples.airline.schema"/>
```

Alternatively, you can provide the list of classes to bind to the marshaller by using the `class-to-be-bound` child element:

```
<oxm:jaxb2-marshaller id="marshaller">
    <oxm:class-to-be-bound
name="org.springframework.ws.samples.airline.schema.Airport"/>
    <oxm:class-to-be-bound
name="org.springframework.ws.samples.airline.schema.Flight"/>
    ...
</oxm:jaxb2-marshaller>
```

The following table describes the available attributes:

Attribute	Description	Required
`id`	The ID of the marshaller	No
`contextPath`	The JAXB Context path	No

4.6.6. JiBX

The JiBX framework offers a solution similar to that which Hibernate provides for ORM: A binding definition defines the rules for how your Java objects are converted to or from XML. After preparing the binding and compiling the classes, a JiBX binding compiler enhances the class files and adds code to handle converting instances of the classes from or to XML.

For more information on JiBX, see the JiBX web site. The Spring integration classes reside in the `org.springframework.oxm.jibx` package.

Using `JibxMarshaller`

The `JibxMarshaller` class implements both the `Marshaller` and `Unmarshaller` interface. To operate, it requires the name of the class to marshal in, which you can set using the `targetClass` property. Optionally, you can set the binding name by setting the `bindingName` property. In the following example, we bind the `Flights` class:

```
<beans>
    <bean id="jibxFlightsMarshaller"
class="org.springframework.oxm.jibx.JibxMarshaller">
        <property name="targetClass">org.springframework.oxm.jibx.Flights</property>
    </bean>
    ...
</beans>
```

A `JibxMarshaller` is configured for a single class. If you want to marshal multiple classes, you have to configure multiple `JibxMarshaller` instances with different `targetClass` property values.

XML Configuration Namespace

The `jibx-marshaller` tag configures a `org.springframework.oxm.jibx.JibxMarshaller`, as the following example shows:

```
<oxm:jibx-marshaller id="marshaller" target-
class="org.springframework.ws.samples.airline.schema.Flight"/>
```

The following table describes the available attributes:

Attribute	Description	Required
`id`	The ID of the marshaller	No
`target-class`	The target class for this marshaller	Yes
`bindingName`	The binding name used by this marshaller	No

4.6.7. XStream

XStream is a simple library to serialize objects to XML and back again. It does not require any mapping and generates clean XML.

For more information on XStream, see the XStream web site. The Spring integration classes reside in the `org.springframework.oxm.xstream` package.

Using `XStreamMarshaller`

The `XStreamMarshaller` does not require any configuration and can be configured in an application context directly. To further customize the XML, you can set an alias map, which consists of string aliases mapped to classes, as the following example shows:

```
<beans>
    <bean id="xstreamMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller">
        <property name="aliases">
            <props>
                <prop key="Flight">org.springframework.oxm.xstream.Flight</prop>
            </props>
        </property>
    </bean>
    ...
</beans>
```

By default, XStream lets arbitrary classes be unmarshalled, which can lead to unsafe Java serialization effects. As such, we do not recommend using the `XStreamMarshaller` to unmarshal XML from external sources (that is, the Web), as this can result in security vulnerabilities.

If you choose to use the `XStreamMarshaller` to unmarshal XML from an external source, set the `supportedClasses` property on the `XStreamMarshaller`, as the following example shows:

```
<bean id="xstreamMarshaller"
class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="supportedClasses"
value="org.springframework.oxm.xstream.Flight"/>
    ...
</bean>
```

Doing so ensures that only the registered classes are eligible for unmarshalling.

Additionally, you can register custom converters to make sure that only your supported classes can be unmarshalled. You might want to add a `CatchAllConverter` as the last converter in the list, in addition to converters that explicitly support the domain classes that should be supported. As a result, default XStream converters with lower priorities and possible security vulnerabilities do not get invoked.

Note that XStream is an XML serialization library, not a data binding library. Therefore, it has limited namespace support. As a result, it is rather unsuitable for usage within Web Services.

4.7. Appendix

4.7.1. XML Schemas

This part of the appendix lists XML schemas for data access, including the following:

- The `tx` Schema
- The `jdbc` Schema

The `tx` Schema

The `tx` tags deal with configuring all of those beans in Spring's comprehensive support for transactions. These tags are covered in the chapter entitled Transaction Management.

We strongly encourage you to look at the `'spring-tx.xsd'` file that ships with the Spring distribution. This file contains the XML Schema for Spring's transaction configuration and covers all of the various elements in the `tx` namespace, including attribute defaults and similar information. This file is documented inline, and, thus, the information is not repeated here in the interests of adhering to the DRY (Don't Repeat Yourself) principle.

In the interest of completeness, to use the elements in the `tx` schema, you need to have the following preamble at the top of your Spring XML configuration file. The text in the following snippet references the correct schema so that the tags in the `tx` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx" ①
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
https://www.springframework.org/schema/tx/spring-tx.xsd ②
        http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- bean definitions here -->

</beans>
```

① Declare usage of the `tx` namespace.

② Specify the location (with other schema locations).

Often, when you use the elements in the `tx` namespace, you are also using the elements from the `aop` namespace (since the declarative transaction support in Spring is implemented by using AOP). The preceding XML snippet contains the relevant lines needed to reference the `aop` schema so that the elements in the `aop` namespace are available to you.

The `jdbc` Schema

The `jdbc` elements let you quickly configure an embedded database or initialize an existing data source. These elements are documented in Embedded Database Support and Initializing a DataSource, respectively.

To use the elements in the `jdbc` schema, you need to have the following preamble at the top of your Spring XML configuration file. The text in the following snippet references the correct schema so that the elements in the `jdbc` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc" ①
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jdbc
https://www.springframework.org/schema/jdbc/spring-jdbc.xsd"> ②

    <!-- bean definitions here -->

</beans>
```

① Declare usage of the `jdbc` namespace.

② Specify the location (with other schema locations).

Chapter 5. Web on Servlet Stack

This part of the documentation covers support for Servlet-stack web applications built on the Servlet API and deployed to Servlet containers. Individual chapters include Spring MVC, View Technologies, CORS Support, and WebSocket Support. For reactive-stack web applications, see Web on Reactive Stack.

5.1. Spring Web MVC

Spring Web MVC is the original web framework built on the Servlet API and has been included in the Spring Framework from the very beginning. The formal name, "Spring Web MVC," comes from the name of its source module (`spring-webmvc`), but it is more commonly known as "Spring MVC".

Parallel to Spring Web MVC, Spring Framework 5.0 introduced a reactive-stack web framework whose name, "Spring WebFlux," is also based on its source module (`spring-webflux`). This chapter covers Spring Web MVC. The next chapter covers Spring WebFlux.

For baseline information and compatibility with Servlet container and Jakarta EE version ranges, see the Spring Framework Wiki.

5.1.1. DispatcherServlet

WebFlux

Spring MVC, as many other web frameworks, is designed around the front controller pattern where a central `Servlet`, the `DispatcherServlet`, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components. This model is flexible and supports diverse workflows.

The `DispatcherServlet`, as any `Servlet`, needs to be declared and mapped according to the Servlet specification by using Java configuration or in `web.xml`. In turn, the `DispatcherServlet` uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, and more.

The following example of the Java configuration registers and initializes the `DispatcherServlet`, which is auto-detected by the Servlet container (see Servlet Config):

Java

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) {

        // Load Spring web application configuration
        AnnotationConfigWebApplicationContext context = new
AnnotationConfigWebApplicationContext();
        context.register(AppConfig.class);

        // Create and register the DispatcherServlet
        DispatcherServlet servlet = new DispatcherServlet(context);
        ServletRegistration.Dynamic registration = servletContext.addServlet("app",
servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/app/*");
    }
}
```

Kotlin

```
class MyWebApplicationInitializer : WebApplicationInitializer {

    override fun onStartup(servletContext: ServletContext) {

        // Load Spring web application configuration
        val context = AnnotationConfigWebApplicationContext()
        context.register(AppConfig::class.java)

        // Create and register the DispatcherServlet
        val servlet = DispatcherServlet(context)
        val registration = servletContext.addServlet("app", servlet)
        registration.setLoadOnStartup(1)
        registration.addMapping("/app/*")
    }
}
```

In addition to using the ServletContext API directly, you can also extend `AbstractAnnotationConfigDispatcherServletInitializer` and override specific methods (see the example under Context Hierarchy).

For programmatic use cases, a `GenericWebApplicationContext` can be used as an alternative to `AnnotationConfigWebApplicationContext`. See the `GenericWebApplicationContext` javadoc for details.

The following example of `web.xml` configuration registers and initializes the `DispatcherServlet`:

```
<web-app>

    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/app-context.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>app</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>app</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>

</web-app>
```

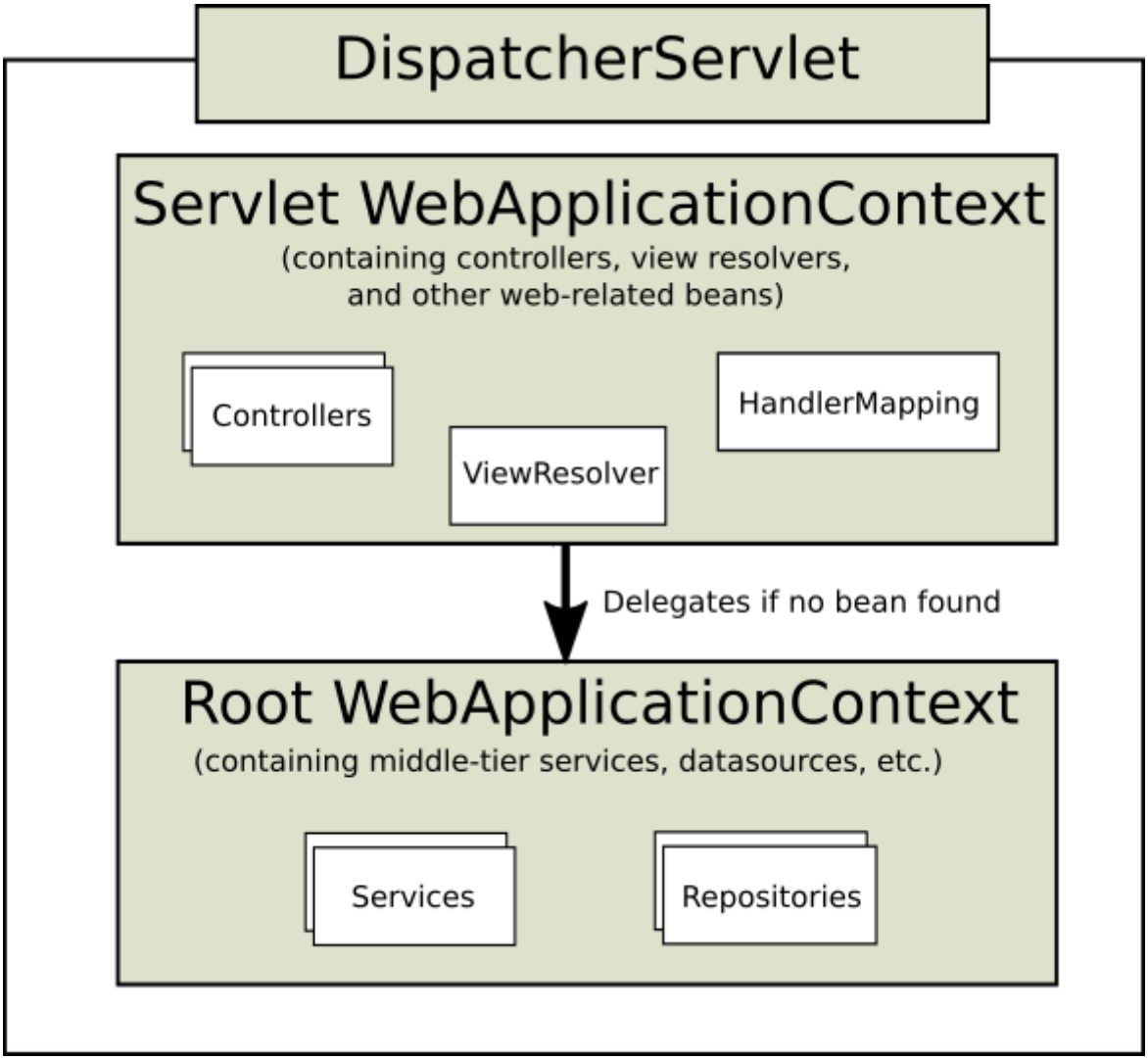
Spring Boot follows a different initialization sequence. Rather than hooking into the lifecycle of the Servlet container, Spring Boot uses Spring configuration to bootstrap itself and the embedded Servlet container. `Filter` and `Servlet` declarations are detected in Spring configuration and registered with the Servlet container. For more details, see the Spring Boot documentation.

Context Hierarchy

`DispatcherServlet` expects a `WebApplicationContext` (an extension of a plain `ApplicationContext`) for its own configuration. `WebApplicationContext` has a link to the `ServletContext` and the `Servlet` with which it is associated. It is also bound to the `ServletContext` such that applications can use static methods on `RequestContextUtils` to look up the `WebApplicationContext` if they need access to it.

For many applications, having a single `WebApplicationContext` is simple and suffices. It is also possible to have a context hierarchy where one root `WebApplicationContext` is shared across multiple `DispatcherServlet` (or other `Servlet`) instances, each with its own child `WebApplicationContext` configuration. See Additional Capabilities of the `ApplicationContext` for more on the context hierarchy feature.

The root `WebApplicationContext` typically contains infrastructure beans, such as data repositories and business services that need to be shared across multiple `Servlet` instances. Those beans are effectively inherited and can be overridden (that is, re-declared) in the Servlet-specific child `WebApplicationContext`, which typically contains beans local to the given `Servlet`. The following image shows this relationship:



The following example configures a `WebApplicationContext` hierarchy:

Java

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] { RootConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { App1Config.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/app1/*" };
    }
}
```

Kotlin

```
class MyWebAppInitializer : AbstractAnnotationConfigDispatcherServletInitializer() {

    override fun getRootConfigClasses(): Array<Class<*>> {
        return arrayOf(RootConfig::class.java)
    }

    override fun getServletConfigClasses(): Array<Class<*>> {
        return arrayOf(App1Config::class.java)
    }

    override fun getServletMappings(): Array<String> {
        return arrayOf("/app1/*")
    }
}
```

If an application context hierarchy is not required, applications can return all configuration through `getRootConfigClasses()` and `null` from `getServletConfigClasses()`.

The following example shows the `web.xml` equivalent:

```
<web-app>

    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/root-context.xml</param-value>
    </context-param>

    <servlet>
        <servlet-name>app1</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/app1-context.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>app1</servlet-name>
        <url-pattern>/app1/*</url-pattern>
    </servlet-mapping>

</web-app>
```

If an application context hierarchy is not required, applications may configure a "root" context only and leave the `contextConfigLocation` Servlet parameter empty.

Special Bean Types

WebFlux

The `DispatcherServlet` delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed `Object` instances that implement framework contracts. Those usually come with built-in contracts, but you can customize their properties and extend or replace them.

The following table lists the special beans detected by the `DispatcherServlet`:

Bean type	Explanation
`HandlerMapping`	Map a request to a handler along with a list of interceptors for pre- and post-processing. The mapping is based on some criteria, the details of which vary by `HandlerMapping` implementation. The two main `HandlerMapping` implementations are `RequestMappingHandlerMapping` (which supports `@RequestMapping` annotated methods) and `SimpleUrlHandlerMapping` (which maintains explicit registrations of URI path patterns to handlers).
`HandlerAdapter`	Help the `DispatcherServlet` to invoke a handler mapped to a request, regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a `HandlerAdapter` is to shield the `DispatcherServlet` from such details.
`HandlerExceptionResolver`	Strategy to resolve exceptions, possibly mapping them to handlers, to HTML error views, or other targets. See Exceptions.
`ViewResolver`	Resolve logical `String`-based view names returned from a handler to an actual `View` with which to render to the response. See View Resolution and View Technologies.
`LocaleResolver`, LocaleContextResolver	Resolve the `Locale` a client is using and possibly their time zone, in order to be able to offer internationalized views. See Locale.
`ThemeResolver`	Resolve themes your web application can use — for example, to offer personalized layouts. See Themes.
`MultipartResolver`	Abstraction for parsing a multi-part request (for example, browser form file upload) with the help of some multipart parsing library. See Multipart Resolver.
`FlashMapManager`	Store and retrieve the “input” and the “output” `FlashMap` that can be used to pass attributes from one request to another, usually across a redirect. See Flash Attributes.

Web MVC Config

WebFlux

Applications can declare the infrastructure beans listed in Special Bean Types that are required to process requests. The `DispatcherServlet` checks the `WebApplicationContext` for each special bean. If there are no matching bean types, it falls back on the default types listed in `DispatcherServlet.properties`.

In most cases, the MVC Config is the best starting point. It declares the required beans in either Java or XML and provides a higher-level configuration callback API to customize it.

Spring Boot relies on the MVC Java configuration to configure Spring MVC and provides many extra convenient options.

Servlet Config

In a Servlet environment, you have the option of configuring the Servlet container programmatically as an alternative or in combination with a `web.xml` file. The following example registers a `DispatcherServlet`:

Java

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic registration = container.addServlet("dispatcher",
new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```

Kotlin

```
import org.springframework.web.WebApplicationInitializer

class MyWebApplicationInitializer : WebApplicationInitializer {

    override fun onStartup(container: ServletContext) {
        val appContext = XmlWebApplicationContext()
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml")

        val registration = container.addServlet("dispatcher",
DispatcherServlet(appContext))
        registration.setLoadOnStartup(1)
        registration.addMapping("/")
    }
}
```

`WebApplicationInitializer` is an interface provided by Spring MVC that ensures your implementation is detected and automatically used to initialize any Servlet 3 container. An abstract base class implementation of `WebApplicationInitializer` named `AbstractDispatcherServletInitializer` makes it even easier to register the `DispatcherServlet` by overriding methods to specify the servlet mapping and the location of the `DispatcherServlet` configuration.

This is recommended for applications that use Java-based Spring configuration, as the following example shows:

Java

```
public class MyWebAppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Kotlin

```
class MyWebAppInitializer : AbstractAnnotationConfigDispatcherServletInitializer() {

    override fun getRootConfigClasses(): Array<Class<*>>? {
        return null
    }

    override fun getServletConfigClasses(): Array<Class<*>>? {
        return arrayOf(MyWebConfig::class.java)
    }

    override fun getServletMappings(): Array<String> {
        return arrayOf("/")
    }
}
```

If you use XML-based Spring configuration, you should extend directly from `AbstractDispatcherServletInitializer`, as the following example shows:

Java

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Kotlin

```
class MyWebAppInitializer : AbstractDispatcherServletInitializer() {

    override fun createRootApplicationContext(): WebApplicationContext? {
        return null
    }

    override fun createServletApplicationContext(): WebApplicationContext {
        return XmlWebApplicationContext().apply {
            setConfigLocation("/WEB-INF/spring/dispatcher-config.xml")
        }
    }

    override fun getServletMappings(): Array<String> {
        return arrayOf("/")
    }
}
```

`AbstractDispatcherServletInitializer` also provides a convenient way to add `Filter` instances and have them be automatically mapped to the `DispatcherServlet`, as the following example shows:

Java

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] {
            new HiddenHttpMethodFilter(), new CharacterEncodingFilter() };
    }
}
```

Kotlin

```
class MyWebAppInitializer : AbstractDispatcherServletInitializer() {

    // ...

    override fun getServletFilters(): Array<Filter> {
        return arrayOf(HiddenHttpMethodFilter(), CharacterEncodingFilter())
    }
}
```

Each filter is added with a default name based on its concrete type and automatically mapped to the `DispatcherServlet`.

The `isAsyncSupported` protected method of `AbstractDispatcherServletInitializer` provides a single place to enable async support on the `DispatcherServlet` and all filters mapped to it. By default, this flag is set to `true`.

Finally, if you need to further customize the `DispatcherServlet` itself, you can override the `createDispatcherServlet` method.

Processing

WebFlux

The `DispatcherServlet` processes requests as follows:

- The `WebApplicationContext` is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE` key.
- The locale resolver is bound to the request to let elements in the process resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need the locale resolver.
- The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.

- If you specify a multipart file resolver, the request is inspected for multiparts. If multiparts are found, the request is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the process. See Multipart Resolver for further information about multipart handling.
- An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is run to prepare a model for rendering. Alternatively, for annotated controllers, the response can be rendered (within the `HandlerAdapter`) instead of returning a view.
- If a model is returned, the view is rendered. If no model is returned (maybe due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

The `HandlerExceptionResolver` beans declared in the `WebApplicationContext` are used to resolve exceptions thrown during request processing. Those exception resolvers allow customizing the logic to address exceptions. See Exceptions for more details.

For HTTP caching support, handlers can use the `checkNotModified` methods of `WebRequest`, along with further options for annotated controllers as described in HTTP Caching for Controllers.

You can customize individual `DispatcherServlet` instances by adding Servlet initialization parameters (`init-param` elements) to the Servlet declaration in the `web.xml` file. The following table lists the supported parameters:

Table 21. DispatcherServlet initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `ConfigurableWebApplicationContext`, to be instantiated and locally configured by this Servlet. By default, `XmlWebApplicationContext` is used.
`contextConfigLocation`	String that is passed to the context instance (specified by `contextClass`) to indicate where contexts can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. In the case of multiple context locations with beans that are defined twice, the latest location takes precedence.
`namespace`	Namespace of the `WebApplicationContext`. Defaults to `[servlet-name]-servlet`.

Parameter	Explanation
`throwExceptionIfNoHandlerFound`	Whether to throw a `NoHandlerFoundException` when no handler was found for a request. The exception can then be caught with a `HandlerExceptionResolver` (for example, by using an `@ExceptionHandler` controller method) and handled as any others. By default, this is set to `false`, in which case the `DispatcherServlet` sets the response status to 404 (NOT_FOUND) without raising an exception. Note that, if default servlet handling is also configured, unresolved requests are always forwarded to the default servlet and a 404 is never raised.

Path Matching

The Servlet API exposes the full request path as `requestURI` and further sub-divides it into `contextPath`, `servletPath`, and `pathInfo` whose values vary depending on how a Servlet is mapped. From these inputs, Spring MVC needs to determine the lookup path to use for mapping handlers, which should exclude the `contextPath` and any `servletMapping` prefix, if applicable.

The `servletPath` and `pathInfo` are decoded and that makes them impossible to compare directly to the full `requestURI` in order to derive the lookupPath and that makes it necessary to decode the `requestURI`. However this introduces its own issues because the path may contain encoded reserved characters such as `"/"` or `";"` that can in turn alter the structure of the path after they are decoded which can also lead to security issues. In addition, Servlet containers may normalize the `servletPath` to varying degrees which makes it further impossible to perform `startsWith` comparisons against the `requestURI`.

This is why it is best to avoid reliance on the `servletPath` which comes with the prefix-based `servletPath` mapping type. If the `DispatcherServlet` is mapped as the default Servlet with `"/"` or otherwise without a prefix with `"/*"` and the Servlet container is 4.0+ then Spring MVC is able to detect the Servlet mapping type and avoid use of the `servletPath` and `pathInfo` altogether. On a 3.1 Servlet container, assuming the same Servlet mapping types, the equivalent can be achieved by providing a `UrlPathHelper` with `alwaysUseFullPath=true` via Path Matching in the MVC config.

Fortunately the default Servlet mapping `"/"` is a good choice. However, there is still an issue in that the `requestURI` needs to be decoded to make it possible to compare to controller mappings. This is again undesirable because of the potential to decode reserved characters that alter the path structure. If such characters are not expected, then you can reject them (like the Spring Security HTTP firewall), or you can configure `UrlPathHelper` with `urlDecode=false` but controller mappings will need to match to the encoded path which may not always work well. Furthermore, sometimes the `DispatcherServlet` needs to share the URL space with another Servlet and may need to be mapped by prefix.

The above issues are addressed when using `PathPatternParser` and parsed patterns, as an alternative to String path matching with `AntPathMatcher`. The `PathPatternParser` has been available for use in Spring MVC from version 5.3, and is enabled by default from version 6.0. Unlike `AntPathMatcher` which needs either the lookup path decoded or the controller mapping encoded, a parsed `PathPattern` matches to a parsed representation of the path called `RequestPath`, one path segment at a time. This allows decoding and sanitizing path segment values individually without the risk of altering the structure of the path. Parsed `PathPattern` also supports the use of `servletPath` prefix mapping as long as a Servlet path mapping is used and the prefix is kept simple, i.e. it has no encoded characters. For pattern syntax details and comparison, see Pattern Comparison.

Interception

All `HandlerMapping` implementations support handler interceptors that are useful when you want to apply specific functionality to certain requests — for example, checking for a principal. Interceptors must implement `HandlerInterceptor` from the `org.springframework.web.servlet` package with three methods that should provide enough flexibility to do all kinds of pre-processing and post-processing:

- `preHandle(..)`: Before the actual handler is run
- `postHandle(..)`: After the handler is run
- `afterCompletion(..)`: After the complete request has finished

The `preHandle(..)` method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns `true`, the handler execution chain continues. When it returns false, the `DispatcherServlet` assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

See Interceptors in the section on MVC configuration for examples of how to configure interceptors. You can also register them directly by using setters on individual `HandlerMapping` implementations.

`postHandle` method is less useful with `@ResponseBody` and `ResponseEntity` methods for which the response is written and committed within the `HandlerAdapter` and before `postHandle`. That means it is too late to make any changes to the response, such as adding an extra header. For such scenarios, you can implement `ResponseBodyAdvice` and either declare it as an Controller Advice bean or configure it directly on `RequestMappingHandlerAdapter`.

Exceptions

WebFlux

If an exception occurs during request mapping or is thrown from a request handler (such as a `@Controller`), the `DispatcherServlet` delegates to a chain of `HandlerExceptionResolver` beans to resolve the exception and provide alternative handling, which is typically an error response.

The following table lists the available `HandlerExceptionResolver` implementations:

Table 22. HandlerExceptionResolver implementations

HandlerExceptionResolver	Description
SimpleMappingExceptionResolver	A mapping between exception class names and error view names. Useful for rendering error pages in a browser application.
DefaultHandlerExceptionResolver	Resolves exceptions raised by Spring MVC and maps them to HTTP status codes. See also alternative ResponseEntityExceptionHandler and Error Responses.
ResponseStatusExceptionResolver	Resolves exceptions with the @ResponseStatus annotation and maps them to HTTP status codes based on the value in the annotation.
ExceptionHandlerExceptionResolver	Resolves exceptions by invoking an @ExceptionHandler method in a @Controller or a @ControllerAdvice class. See @ExceptionHandler methods.

Chain of Resolvers

You can form an exception resolver chain by declaring multiple HandlerExceptionResolver beans in your Spring configuration and setting their order properties as needed. The higher the order property, the later the exception resolver is positioned.

The contract of HandlerExceptionResolver specifies that it can return:

- a ModelAndView that points to an error view.
- An empty ModelAndView if the exception was handled within the resolver.
- null if the exception remains unresolved, for subsequent resolvers to try, and, if the exception remains at the end, it is allowed to bubble up to the Servlet container.

The MVC Config automatically declares built-in resolvers for default Spring MVC exceptions, for @ResponseStatus annotated exceptions, and for support of @ExceptionHandler methods. You can customize that list or replace it.

Container Error Page

If an exception remains unresolved by any HandlerExceptionResolver and is, therefore, left to propagate or if the response status is set to an error status (that is, 4xx, 5xx), Servlet containers can render a default error page in HTML. To customize the default error page of the container, you can declare an error page mapping in web.xml. The following example shows how to do so:

```
<error-page>
    <location>/error</location>
</error-page>
```

Given the preceding example, when an exception bubbles up or the response has an error status, the Servlet container makes an ERROR dispatch within the container to the configured URL (for example, /error). This is then processed by the DispatcherServlet, possibly mapping it to a @Controller, which could be implemented to return an error view name with a model or to render a

JSON response, as the following example shows:

Java

```
@RestController
public class ErrorController {

    @RequestMapping(path = "/error")
    public Map<String, Object> handle(HttpServletRequest request) {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("status", request.getAttribute("jakarta.servlet.error.status_code"));
        map.put("reason", request.getAttribute("jakarta.servlet.error.message"));
        return map;
    }
}
```

Kotlin

```
@RestController
class ErrorController {

    @RequestMapping(path = ["/error"])
    fun handle(request: HttpServletRequest): Map<String, Any> {
        val map = HashMap<String, Any>()
        map["status"] = request.getAttribute("jakarta.servlet.error.status_code")
        map["reason"] = request.getAttribute("jakarta.servlet.error.message")
        return map
    }
}
```

The Servlet API does not provide a way to create error page mappings in Java. You can, however, use both a `WebApplicationInitializer` and a minimal `web.xml`.

View Resolution

WebFlux

Spring MVC defines the `ViewResolver` and `View` interfaces that let you render models in a browser without tying you to a specific view technology. `ViewResolver` provides a mapping between view names and actual views. `View` addresses the preparation of data before handing over to a specific view technology.

The following table provides more details on the `ViewResolver` hierarchy:

Table 23. ViewResolver implementations

ViewResolver	Description
`AbstractCachingViewResolver`	Subclasses of `AbstractCachingViewResolver` cache view instances that they resolve. Caching improves performance of certain view technologies. You can turn off the cache by setting the `cache` property to `false`. Furthermore, if you must refresh a certain view at runtime (for example, when a FreeMarker template is modified), you can use the `removeFromCache(String viewName, Locale loc)` method.
`UrlBasedViewResolver`	Simple implementation of the `ViewResolver` interface that effects the direct resolution of logical view names to URLs without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
`InternalResourceViewResolver`	Convenient subclass of `UrlBasedViewResolver` that supports `InternalResourceView` (in effect, Servlets and JSPs) and subclasses such as `JstlView` and `TilesView`. You can specify the view class for all views generated by this resolver by using `setViewClass(..)`. See the `UrlBasedViewResolver` javadoc for details.
`FreeMarkerViewResolver`	Convenient subclass of `UrlBasedViewResolver` that supports `FreeMarkerView` and custom subclasses of them.
`ContentNegotiatingViewResolver`	Implementation of the `ViewResolver` interface that resolves a view based on the request file name or `Accept` header. See Content Negotiation.
`BeanNameViewResolver`	Implementation of the `ViewResolver` interface that interprets a view name as a bean name in the current application context. This is a very flexible variant which allows for mixing and matching different view types based on distinct view names. Each such `View` can be defined as a bean e.g. in XML or in configuration classes.

Handling

WebFlux

You can chain view resolvers by declaring more than one resolver bean and, if necessary, by setting the `order` property to specify ordering. Remember, the higher the order property, the later the view resolver is positioned in the chain.

The contract of a `ViewResolver` specifies that it can return null to indicate that the view could not be found. However, in the case of JSPs and `InternalResourceViewResolver`, the only way to figure out if a JSP exists is to perform a dispatch through `RequestDispatcher`. Therefore, you must always configure an `InternalResourceViewResolver` to be last in the overall order of view resolvers.

Configuring view resolution is as simple as adding `ViewResolver` beans to your Spring configuration. The MVC Config provides a dedicated configuration API for View Resolvers and for adding logic-less View Controllers which are useful for HTML template rendering without controller logic.

Redirecting

WebFlux

The special `redirect:` prefix in a view name lets you perform a redirect. The `UrlBasedViewResolver` (and its subclasses) recognize this as an instruction that a redirect is needed. The rest of the view name is the redirect URL.

The net effect is the same as if the controller had returned a `RedirectView`, but now the controller itself can operate in terms of logical view names. A logical view name (such as `redirect:/myapp/some/resource`) redirects relative to the current Servlet context, while a name such as `redirect:https://myhost.com/some/arbitrary/path` redirects to an absolute URL.

Note that, if a controller method is annotated with the `@ResponseStatus`, the annotation value takes precedence over the response status set by `RedirectView`.

Forwarding

You can also use a special `forward:` prefix for view names that are ultimately resolved by `UrlBasedViewResolver` and subclasses. This creates an `InternalResourceView`, which does a `RequestDispatcher.forward()`. Therefore, this prefix is not useful with `InternalResourceViewResolver` and `InternalResourceView` (for JSPs), but it can be helpful if you use another view technology but still want to force a forward of a resource to be handled by the Servlet/JSP engine. Note that you may also chain multiple view resolvers, instead.

Content Negotiation

WebFlux

`ContentNegotiatingViewResolver` does not resolve views itself but rather delegates to other view resolvers and selects the view that resembles the representation requested by the client. The representation can be determined from the `Accept` header or from a query parameter (for example, `"/path?format=pdf"`).

The `ContentNegotiatingViewResolver` selects an appropriate `View` to handle the request by comparing the request media types with the media type (also known as `Content-Type`) supported by the `View` associated with each of its `ViewResolvers`. The first `View` in the list that has a compatible `Content-Type` returns the representation to the client. If a compatible view cannot be supplied by the `ViewResolver` chain, the list of views specified through the `DefaultViews` property is consulted. This latter option is appropriate for singleton `Views` that can render an appropriate representation of the current resource regardless of the logical view name. The `Accept` header can include wildcards (for example `text/*`), in which case a `View` whose `Content-Type` is `text/xml` is a compatible match.

See View Resolvers under MVC Config for configuration details.

Locale

Most parts of Spring's architecture support internationalization, as the Spring web MVC framework does. `DispatcherServlet` lets you automatically resolve messages by using the client's locale. This is done with `LocaleResolver` objects.

When a request comes in, the `DispatcherServlet` looks for a locale resolver and, if it finds one, it tries to use it to set the locale. By using the `RequestContext.getLocale()` method, you can always retrieve the locale that was resolved by the locale resolver.

In addition to automatic locale resolution, you can also attach an interceptor to the handler mapping (see Interception for more information on handler mapping interceptors) to change the locale under specific circumstances (for example, based on a parameter in the request).

Locale resolvers and interceptors are defined in the `org.springframework.web.servlet.i18n` package and are configured in your application context in the normal way. The following selection of locale resolvers is included in Spring.

- Time Zone
- Header Resolver
- Cookie Resolver
- Session Resolver
- Locale Interceptor

Time Zone

In addition to obtaining the client's locale, it is often useful to know its time zone. The `LocaleContextResolver` interface offers an extension to `LocaleResolver` that lets resolvers provide a richer `LocaleContext`, which may include time zone information.

When available, the user's `TimeZone` can be obtained by using the `RequestContext.getTimeZone()` method. Time zone information is automatically used by any Date/Time `Converter` and `Formatter` objects that are registered with Spring's `ConversionService`.

Header Resolver

This locale resolver inspects the `accept-language` header in the request that was sent by the client (for example, a web browser). Usually, this header field contains the locale of the client's operating system. Note that this resolver does not support time zone information.

Cookie Resolver

This locale resolver inspects a `Cookie` that might exist on the client to see if a `Locale` or `TimeZone` is specified. If so, it uses the specified details. By using the properties of this locale resolver, you can specify the name of the cookie as well as the maximum age. The following example defines a `CookieLocaleResolver`:

```
<bean id="localeResolver"
class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser
shuts down) -->
    <property name="cookieMaxAge" value="100000"/>

</bean>
```

The following table describes the properties `CookieLocaleResolver`:

Table 24. CookieLocaleResolver properties

Property	Default	Description
`cookieName`	classname + LOCALE	The name of the cookie
`cookieMaxAge`	Servlet container default	The maximum time a cookie persists on the client. If `-1` is specified, the cookie will not be persisted. It is available only until the client shuts down the browser.
`cookiePath`	/	Limits the visibility of the cookie to a certain part of your site. When `cookiePath` is specified, the cookie is visible only to that path and the paths below it.

Session Resolver

The `SessionLocaleResolver` lets you retrieve `Locale` and `TimeZone` from the session that might be associated with the user's request. In contrast to `CookieLocaleResolver`, this strategy stores locally chosen locale settings in the Servlet container's `HttpSession`. As a consequence, those settings are temporary for each session and are, therefore, lost when each session ends.

Note that there is no direct relationship with external session management mechanisms, such as the Spring Session project. This `SessionLocaleResolver` evaluates and modifies the corresponding `HttpSession` attributes against the current `HttpServletRequest`.

Locale Interceptor

You can enable changing of locales by adding the `LocaleChangeInterceptor` to one of the `HandlerMapping` definitions. It detects a parameter in the request and changes the locale accordingly, calling the `setLocale` method on the `LocaleResolver` in the dispatcher's application context. The next example shows that calls to all `*.view` resources that contain a parameter named `siteLanguage` now changes the locale. So, for example, a request for the URL, `https://www.sf.net/home.view?siteLanguage=nl`, changes the site language to Dutch. The following example shows how to intercept the locale:

```
<bean id="localeChangeInterceptor"
        class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver"
        class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

Themes

You can apply Spring Web MVC framework themes to set the overall look-and-feel of your application, thereby enhancing user experience. A theme is a collection of static resources, typically style sheets and images, that affect the visual style of the application.

as of 6.0 support for themes has been deprecated theme in favor of using CSS, and without any special support on the server side.

Defining a theme

To use themes in your web application, you must set up an implementation of the `org.springframework.ui.context.ThemeSource` interface. The `WebApplicationContext` interface extends `ThemeSource` but delegates its responsibilities to a dedicated implementation. By default, the delegate is an `org.springframework.ui.context.support.ResourceBundleThemeSource` implementation that loads properties files from the root of the classpath. To use a custom `ThemeSource` implementation or to configure the base name prefix of the `ResourceBundleThemeSource`, you can register a bean in the application context with the reserved name, `themeSource`. The web application context automatically detects a bean with that name and uses it.

When you use the `ResourceBundleThemeSource`, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme, as the following example shows:

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

The keys of the properties are the names that refer to the themed elements from view code. For a

JSP, you typically do this using the `spring:theme` custom tag, which is very similar to the `spring:message` tag. The following JSP fragment uses the theme defined in the previous example to customize the look and feel:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
    <head>
        <link rel="stylesheet" href="<spring:theme code='styleSheet'/>"
type="text/css"/>
    </head>
    <body style="background=<spring:theme code='background'/>">
        ...
    </body>
</html>
```

By default, the `ResourceBundleThemeSource` uses an empty base name prefix. As a result, the properties files are loaded from the root of the classpath. Thus, you would put the `cool.properties` theme definition in a directory at the root of the classpath (for example, in `/WEB-INF/classes`). The `ResourceBundleThemeSource` uses the standard Java resource bundle loading mechanism, allowing for full internationalization of themes. For example, we could have a `/WEB-INF/classes/cool_nl.properties` that references a special background image with Dutch text on it.

Resolving Themes

After you define themes, as described in the preceding section, you decide which theme to use. The `DispatcherServlet` looks for a bean named `themeResolver` to find out which `ThemeResolver` implementation to use. A theme resolver works in much the same way as a `LocaleResolver`. It detects the theme to use for a particular request and can also alter the request's theme. The following table describes the theme resolvers provided by Spring:

Table 25. ThemeResolver implementations

Class	Description
`FixedThemeResolver`	Selects a fixed theme, set by using the `defaultThemeName` property.
`SessionThemeResolver`	The theme is maintained in the user's HTTP session. It needs to be set only once for each session but is not persisted between sessions.
`CookieThemeResolver`	The selected theme is stored in a cookie on the client.

Spring also provides a `ThemeChangeInterceptor` that lets theme changes on every request with a simple request parameter.

Multipart Resolver

WebFlux

`MultipartResolver` from the `org.springframework.web.multipart` package is a strategy for parsing multipart requests including file uploads. There is one implementation based on Commons FileUpload and another based on Servlet multipart request parsing.

To enable multipart handling, you need to declare a `MultipartResolver` bean in your `DispatcherServlet` Spring configuration with a name of `multipartResolver`. The `DispatcherServlet` detects it and applies it to the incoming request. When a POST with a content type of `multipart/form-data` is received, the resolver parses the content wraps the current `HttpServletRequest` as a `MultipartHttpServletRequest` to provide access to resolved files in addition to exposing parts as request parameters.

Apache Commons `FileUpload`

To use Apache Commons `FileUpload`, you can configure a bean of type `CommonsMultipartResolver` with a name of `multipartResolver`. You also need to have the `commons-fileupload` jar as a dependency on your classpath.

This resolver variant delegates to a local library within the application, providing maximum portability across Servlet containers. As an alternative, consider standard Servlet multipart resolution through the container's own parser as discussed below.

Commons FileUpload traditionally applies to POST requests only but accepts any `multipart/` content type. See the `CommonsMultipartResolver` javadoc for details and configuration options.

Servlet Multipart Parsing

Servlet multipart parsing needs to be enabled through Servlet container configuration. To do so:

- In Java, set a `MultipartConfigElement` on the Servlet registration.
- In `web.xml`, add a `"<multipart-config>"` section to the servlet declaration.

The following example shows how to set a `MultipartConfigElement` on the Servlet registration:

Java

```
public class AppInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    // ...

    @Override
    protected void customizeRegistration(ServletRegistration.Dynamic registration) {

        // Optionally also set maxFileSize, maxRequestSize, fileSizeThreshold
        registration.setMultipartConfig(new MultipartConfigElement("/tmp"));
    }

}
```

Kotlin

```
class AppInitializer : AbstractAnnotationConfigDispatcherServletInitializer() {

    // ...

    override fun customizeRegistration(registration: ServletRegistration.Dynamic) {

        // Optionally also set maxFileSize, maxRequestSize, fileSizeThreshold
        registration.setMultipartConfig(MultipartConfigElement("/tmp"))
    }

}
```

Once the Servlet multipart configuration is in place, you can add a bean of type `StandardServletMultipartResolver` with a name of `multipartResolver`.

This resolver variant uses your Servlet container's multipart parser as-is, potentially exposing the application to container implementation differences. By default, it will try to parse any `multipart/` content type with any HTTP method but this may not be supported across all Servlet containers. See the `StandardServletMultipartResolver` javadoc for details and configuration options.

Logging

WebFlux

DEBUG-level logging in Spring MVC is designed to be compact, minimal, and human-friendly. It focuses on high-value bits of information that are useful over and over again versus others that are useful only when debugging a specific issue.

TRACE-level logging generally follows the same principles as DEBUG (and, for example, also should not be a fire hose) but can be used for debugging any issue. In addition, some log messages may show a different level of detail at TRACE versus DEBUG.

Good logging comes from the experience of using the logs. If you spot anything that does not meet the stated goals, please let us know.

Sensitive Data

WebFlux

DEBUG and TRACE logging may log sensitive information. This is why request parameters and headers are masked by default and their logging in full must be enabled explicitly through the `enableLoggingRequestDetails` property on `DispatcherServlet`.

The following example shows how to do so by using Java configuration:

Java

```
public class MyInitializer
        extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return ... ;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return ... ;
    }

    @Override
    protected String[] getServletMappings() {
        return ... ;
    }

    @Override
    protected void customizeRegistration(ServletRegistration.Dynamic registration) {
        registration.setInitParameter("enableLoggingRequestDetails", "true");
    }

}
```

Kotlin

```
class MyInitializer : AbstractAnnotationConfigDispatcherServletInitializer() {

    override fun getRootConfigClasses(): Array<Class<*>>? {
        return ...
    }

    override fun getServletConfigClasses(): Array<Class<*>>? {
        return ...
    }

    override fun getServletMappings(): Array<String> {
        return ...
    }

    override fun customizeRegistration(registration: ServletRegistration.Dynamic) {
        registration.setInitParameter("enableLoggingRequestDetails", "true")
    }
}
```

5.1.2. Filters

WebFlux

The `spring-web` module provides some useful filters:

- Form Data
- Forwarded Headers
- Shallow ETag
- CORS

Form Data

Browsers can submit form data only through HTTP GET or HTTP POST but non-browser clients can also use HTTP PUT, PATCH, and DELETE. The Servlet API requires `ServletRequest.getParameter*()` methods to support form field access only for HTTP POST.

The `spring-web` module provides `FormContentFilter` to intercept HTTP PUT, PATCH, and DELETE requests with a content type of `application/x-www-form-urlencoded`, read the form data from the body of the request, and wrap the `ServletRequest` to make the form data available through the `ServletRequest.getParameter*()` family of methods.

Forwarded Headers

WebFlux

As a request goes through proxies (such as load balancers) the host, port, and scheme may change, and that makes it a challenge to create links that point to the correct host, port, and scheme from a client perspective.

RFC 7239 defines the `Forwarded` HTTP header that proxies can use to provide information about the original request. There are other non-standard headers, too, including `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto`, `X-Forwarded-Ssl`, and `X-Forwarded-Prefix`.

`ForwardedHeaderFilter` is a Servlet filter that modifies the request in order to a) change the host, port, and scheme based on `Forwarded` headers, and b) to remove those headers to eliminate further impact. The filter relies on wrapping the request, and therefore it must be ordered ahead of other filters, such as `RequestContextFilter`, that should work with the modified and not the original request.

There are security considerations for forwarded headers since an application cannot know if the headers were added by a proxy, as intended, or by a malicious client. This is why a proxy at the boundary of trust should be configured to remove untrusted `Forwarded` headers that come from the outside. You can also configure the `ForwardedHeaderFilter` with `removeOnly=true`, in which case it removes but does not use the headers.

In order to support asynchronous requests and error dispatches this filter should be mapped with `DispatcherType.ASYNC` and also `DispatcherType.ERROR`. If using Spring Framework's `AbstractAnnotationConfigDispatcherServletInitializer` (see Servlet Config) all filters are

automatically registered for all dispatch types. However if registering the filter via `web.xml` or in Spring Boot via a `FilterRegistrationBean` be sure to include `DispatcherType.ASYNC` and `DispatcherType.ERROR` in addition to `DispatcherType.REQUEST`.

Shallow ETag

The `ShallowEtagHeaderFilter` filter creates a "shallow" ETag by caching the content written to the response and computing an MD5 hash from it. The next time a client sends, it does the same, but it also compares the computed value against the `If-None-Match` request header and, if the two are equal, returns a 304 (NOT_MODIFIED).

This strategy saves network bandwidth but not CPU, as the full response must be computed for each request. Other strategies at the controller level, described earlier, can avoid the computation. See HTTP Caching.

This filter has a `writeWeakETag` parameter that configures the filter to write weak ETags similar to the following: `W/"02a2d595e6ed9a0b24f027f2b63b134d6"` (as defined in RFC 7232 Section 2.3).

In order to support asynchronous requests this filter must be mapped with `DispatcherType.ASYNC` so that the filter can delay and successfully generate an ETag to the end of the last async dispatch. If using Spring Framework's `AbstractAnnotationConfigDispatcherServletInitializer` (see Servlet Config) all filters are automatically registered for all dispatch types. However if registering the filter via `web.xml` or in Spring Boot via a `FilterRegistrationBean` be sure to include `DispatcherType.ASYNC`.

CORS

WebFlux

Spring MVC provides fine-grained support for CORS configuration through annotations on controllers. However, when used with Spring Security, we advise relying on the built-in `CorsFilter` that must be ordered ahead of Spring Security's chain of filters.

See the sections on CORS and the CORS Filter for more details.

5.1.3. Annotated Controllers

WebFlux

Spring MVC provides an annotation-based programming model where `@Controller` and `@RestController` components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces. The following example shows a controller defined by annotations:

Java

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    public String handle(Model model) {
        model.addAttribute("message", "Hello World!");
        return "index";
    }
}
```

Kotlin

```
import org.springframework.ui.set

@Controller
class HelloController {

    @GetMapping("/hello")
    fun handle(model: Model): String {
        model["message"] = "Hello World!"
        return "index"
    }
}
```

In the preceding example, the method accepts a `Model` and returns a view name as a `String`, but many other options exist and are explained later in this chapter.

Guides and tutorials on spring.io use the annotation-based programming model described in this section.

Declaration

WebFlux

You can define controller beans by using a standard Spring bean definition in the Servlet's `WebApplicationContext`. The `@Controller` stereotype allows for auto-detection, aligned with Spring general support for detecting `@Component` classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

To enable auto-detection of such `@Controller` beans, you can add component scanning to your Java configuration, as the following example shows:

Java

```
@Configuration
@ComponentScan("org.example.web")
public class WebConfig {

    // ...
}
```

Kotlin

```
@Configuration
@ComponentScan("org.example.web")
class WebConfig {

    // ...
}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example.web"/>

    <!-- ... -->

</beans>
```

`@RestController` is a composed annotation that is itself meta-annotated with `@Controller` and `@ResponseBody` to indicate a controller whose every method inherits the type-level `@ResponseBody` annotation and, therefore, writes directly to the response body versus view resolution and rendering with an HTML template.

AOP Proxies

WebFlux

In some cases, you may need to decorate a controller with an AOP proxy at runtime. One example is if you choose to have `@Transactional` annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is automatically the

case with such annotations directly on the controller.

If the controller implements an interface, and needs AOP proxying, you may need to explicitly configure class-based proxying. For example, with `@EnableTransactionManagement` you can change to `@EnableTransactionManagement(proxyTargetClass = true)`, and with `<tx:annotation-driven/>` you can change to `<tx:annotation-driven proxy-target-class="true"/>`.

Keep in mind that as of 6.0, with interface proxying, Spring MVC no longer detects controllers based solely on a type-level `@RequestMapping` annotation on the interface. Please, enable class based proxying, or otherwise the interface must also have an `@Controller` annotation.

Request Mapping

WebFlux

You can use the `@RequestMapping` annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

There are also HTTP method specific shortcut variants of `@RequestMapping`:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

The shortcuts are Custom Annotations that are provided because, arguably, most controller methods should be mapped to a specific HTTP method versus using `@RequestMapping`, which, by default, matches to all HTTP methods. A `@RequestMapping` is still needed at the class level to express shared mappings.

The following example has type and method level mappings:

Java

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

Kotlin

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    fun getPerson(@PathVariable id: Long): Person {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    fun add(@RequestBody person: Person) {
        // ...
    }
}
```

URI patterns

WebFlux

`@RequestMapping` methods can be mapped using URL patterns. There are two alternatives:

- `PathPattern` — a pre-parsed pattern matched against the URL path also pre-parsed as `PathContainer`. Designed for web use, this solution deals effectively with encoding and path parameters, and matches efficiently.
- `AntPathMatcher` — match String patterns against a String path. This is the original solution also used in Spring configuration to select resources on the classpath, on the filesystem, and other locations. It is less efficient and the String path input is a challenge for dealing effectively with encoding and other issues with URLs.

`PathPattern` is the recommended solution for web applications and it is the only choice in Spring WebFlux. It was enabled for use in Spring MVC from version 5.3 and is enabled by default from version 6.0. See MVC config for customizations of path matching options.

`PathPattern` supports the same pattern syntax as `AntPathMatcher`. In addition, it also supports the capturing pattern, e.g. `{*spring}`, for matching 0 or more path segments at the end of a path. `PathPattern` also restricts the use of `**` for matching multiple path segments such that it's only allowed at the end of a pattern. This eliminates many cases of ambiguity when choosing the best matching pattern for a given request. For full pattern syntax please refer to PathPattern and AntPathMatcher.

Some example patterns:

- `"/resources/ima?e.png"` - match one character in a path segment
- `"/resources/*.png"` - match zero or more characters in a path segment
- `"/resources/**"` - match multiple path segments
- `"/projects/{project}/versions"` - match a path segment and capture it as a variable
- `"/projects/{project:[a-z]+}/versions"` - match and capture a variable with a regex

Captured URI variables can be accessed with `@PathVariable`. For example:

Java

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

Kotlin

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
    // ...
}
```

You can declare URI variables at the class and method levels, as the following example shows:

Java

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class OwnerController {

    @GetMapping("/pets/{petId}")
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

Kotlin

```
@Controller
@RequestMapping("/owners/{ownerId}")
class OwnerController {

    @GetMapping("/pets/{petId}")
    fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
        // ...
    }
}
```

URI variables are automatically converted to the appropriate type, or `TypeMismatchException` is raised. Simple types (`int`, `long`, `Date`, and so on) are supported by default and you can register support for any other data type. See Type Conversion and `DataBinder`.

You can explicitly name URI variables (for example, `@PathVariable("customId")`), but you can leave that detail out if the names are the same and your code is compiled with the `-parameters` compiler flag.

The syntax `{varName:regex}` declares a URI variable with a regular expression that has syntax of `{varName:regex}`. For example, given URL `"/spring-web-3.0.5.jar"`, the following method extracts the name, version, and file extension:

Java

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
public void handle(@PathVariable String name, @PathVariable String version,
@PathVariable String ext) {
    // ...
}
```

Kotlin

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
fun handle(@PathVariable name: String, @PathVariable version: String, @PathVariable
ext: String) {
    // ...
}
```

URI path patterns can also have embedded `${…}` placeholders that are resolved on startup by using `PropertySourcesPlaceholderConfigurer` against local, system, environment, and other property sources. You can use this, for example, to parameterize a base URL based on some external configuration.

Pattern Comparison

WebFlux

When multiple patterns match a URL, the best match must be selected. This is done with one of the following depending on whether use of parsed `PathPattern` is enabled for use or not:

- `PathPattern.SPECIFICITY_COMPARATOR`
- `AntPathMatcher.getPatternComparator(String path)`

Both help to sort patterns with more specific ones on top. A pattern is less specific if it has a lower count of URI variables (counted as 1), single wildcards (counted as 1), and double wildcards (counted as 2). Given an equal score, the longer pattern is chosen. Given the same score and length, the pattern with more URI variables than wildcards is chosen.

The default mapping pattern (`/**`) is excluded from scoring and always sorted last. Also, prefix patterns (such as `/public/**`) are considered less specific than other pattern that do not have double wildcards.

For the full details, follow the above links to the pattern Comparators.

Suffix Match

Starting in 5.3, by default Spring MVC no longer performs `.*` suffix pattern matching where a controller mapped to `/person` is also implicitly mapped to `/person.*`. As a consequence path extensions are no longer used to interpret the requested content type for the response — for example, `/person.pdf`, `/person.xml`, and so on.

Using file extensions in this way was necessary when browsers used to send `Accept` headers that were hard to interpret consistently. At present, that is no longer a necessity and using the `Accept` header should be the preferred choice.

Over time, the use of file name extensions has proven problematic in a variety of ways. It can cause ambiguity when overlain with the use of URI variables, path parameters, and URI encoding. Reasoning about URL-based authorization and security (see next section for more details) also becomes more difficult.

To completely disable the use of path extensions in versions prior to 5.3, set the following:

- `useSuffixPatternMatching(false)`, see PathMatchConfigurer
- `favorPathExtension(false)`, see ContentNegotiationConfigurer

Having a way to request content types other than through the `"Accept"` header can still be useful, e.g. when typing a URL in a browser. A safe alternative to path extensions is to use the query parameter strategy. If you must use file extensions, consider restricting them to a list of explicitly registered extensions through the `mediaTypes` property of ContentNegotiationConfigurer.

Suffix Match and RFD

A reflected file download (RFD) attack is similar to XSS in that it relies on request input (for example, a query parameter and a URI variable) being reflected in the response. However, instead of inserting JavaScript into HTML, an RFD attack relies on the browser switching to perform a download and treating the response as an executable script when double-clicked later.

In Spring MVC, `@ResponseBody` and `ResponseEntity` methods are at risk, because they can render different content types, which clients can request through URL path extensions. Disabling suffix pattern matching and using path extensions for content negotiation lower the risk but are not sufficient to prevent RFD attacks.

To prevent RFD attacks, prior to rendering the response body, Spring MVC adds a `Content-Disposition:inline;filename=f.txt` header to suggest a fixed and safe download file. This is done only if the URL path contains a file extension that is neither allowed as safe nor explicitly registered for content negotiation. However, it can potentially have side effects when URLs are typed directly into a browser.

Many common path extensions are allowed as safe by default. Applications with custom `HttpMessageConverter` implementations can explicitly register file extensions for content negotiation to avoid having a `Content-Disposition` header added for those extensions. See Content Types.

See CVE-2015-5211 for additional recommendations related to RFD.

Consumable Media Types

WebFlux

You can narrow the request mapping based on the `Content-Type` of the request, as the following example shows:

Java

```
@PostMapping(path = "/pets", consumes = "application/json") ①
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

① Using a `consumes` attribute to narrow the mapping by the content type.

Kotlin

```
@PostMapping("/pets", consumes = ["application/json"]) ①
fun addPet(@RequestBody pet: Pet) {
    // ...
}
```

① Using a `consumes` attribute to narrow the mapping by the content type.

The `consumes` attribute also supports negation expressions — for example, `!text/plain` means any content type other than `text/plain`.

You can declare a shared `consumes` attribute at the class level. Unlike most other request-mapping attributes, however, when used at the class level, a method-level `consumes` attribute overrides rather than extends the class-level declaration.

`MediaType` provides constants for commonly used media types, such as `APPLICATION_JSON_VALUE` and `APPLICATION_XML_VALUE`.

Producible Media Types

WebFlux

You can narrow the request mapping based on the `Accept` request header and the list of content types that a controller method produces, as the following example shows:

Java

```
@GetMapping(path = "/pets/{petId}", produces = "application/json") ①
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

① Using a `produces` attribute to narrow the mapping by the content type.

Kotlin

```
@GetMapping("/pets/{petId}", produces = ["application/json"]) ①
@ResponseBody
fun getPet(@PathVariable petId: String): Pet {
    // ...
}
```

① Using a `produces` attribute to narrow the mapping by the content type.

The media type can specify a character set. Negated expressions are supported — for example, `!text/plain` means any content type other than "text/plain".

You can declare a shared `produces` attribute at the class level. Unlike most other request-mapping attributes, however, when used at the class level, a method-level `produces` attribute overrides rather

than extends the class-level declaration.

`MediaType` provides constants for commonly used media types, such as `APPLICATION_JSON_VALUE` and `APPLICATION_XML_VALUE`.

Parameters, headers

WebFlux

You can narrow request mappings based on request parameter conditions. You can test for the presence of a request parameter (`myParam`), for the absence of one (`!myParam`), or for a specific value (`myParam=myValue`). The following example shows how to test for a specific value:

Java

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue") ①
public void findPet(@PathVariable String petId) {
    // ...
}
```

① Testing whether `myParam` equals `myValue`.

Kotlin

```
@GetMapping("/pets/{petId}", params = ["myParam=myValue"]) ①
fun findPet(@PathVariable petId: String) {
    // ...
}
```

① Testing whether `myParam` equals `myValue`.

You can also use the same with request header conditions, as the following example shows:

Java

```
@GetMapping(path = "/pets", headers = "myHeader=myValue") ①
public void findPet(@PathVariable String petId) {
    // ...
}
```

① Testing whether `myHeader` equals `myValue`.

Kotlin

```
@GetMapping("/pets", headers = ["myHeader=myValue"]) ①
fun findPet(@PathVariable petId: String) {
    // ...
}
```

① Testing whether `myHeader` equals `myValue`.

You can match `Content-Type` and `Accept` with the headers condition, but it is better to use consumes and produces instead.

HTTP HEAD, OPTIONS

WebFlux

`@GetMapping` (and `@RequestMapping(method=HttpMethod.GET)`) support HTTP HEAD transparently for request mapping. Controller methods do not need to change. A response wrapper, applied in `jakarta.servlet.http.HttpServlet`, ensures a `Content-Length` header is set to the number of bytes written (without actually writing to the response).

`@GetMapping` (and `@RequestMapping(method=HttpMethod.GET)`) are implicitly mapped to and support HTTP HEAD. An HTTP HEAD request is processed as if it were HTTP GET except that, instead of writing the body, the number of bytes are counted and the `Content-Length` header is set.

By default, HTTP OPTIONS is handled by setting the `Allow` response header to the list of HTTP methods listed in all `@RequestMapping` methods that have matching URL patterns.

For a `@RequestMapping` without HTTP method declarations, the `Allow` header is set to `GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS`. Controller methods should always declare the supported HTTP methods (for example, by using the HTTP method specific variants: `@GetMapping`, `@PostMapping`, and others).

You can explicitly map the `@RequestMapping` method to HTTP HEAD and HTTP OPTIONS, but that is not necessary in the common case.

Custom Annotations

WebFlux

Spring MVC supports the use of composed annotations for request mapping. Those are annotations that are themselves meta-annotated with `@RequestMapping` and composed to redeclare a subset (or all) of the `@RequestMapping` attributes with a narrower, more specific purpose.

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping` are examples of composed annotations. They are provided because, arguably, most controller methods should be mapped to a specific HTTP method versus using `@RequestMapping`, which, by default, matches to all HTTP methods. If you need an example of composed annotations, look at how those are declared.

Spring MVC also supports custom request-mapping attributes with custom request-matching logic. This is a more advanced option that requires subclassing `RequestMappingHandlerMapping` and overriding the `getCustomMethodCondition` method, where you can check the custom attribute and return your own `RequestCondition`.

Explicit Registrations

WebFlux

You can programmatically register handler methods, which you can use for dynamic registrations or for advanced cases, such as different instances of the same handler under different URLs. The

following example registers a handler method:

Java

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler
handler) ①
            throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
                .paths("/user/{id}").methods(RequestMethod.GET).build(); ②

        Method method = UserHandler.class.getMethod("getUser", Long.class); ③

        mapping.registerMapping(info, handler, method); ④
    }
}
```

① Inject the target handler and the handler mapping for controllers.

② Prepare the request mapping meta data.

③ Get the handler method.

④ Add the registration.

Kotlin

```
@Configuration
class MyConfig {

    @Autowired
    fun setHandlerMapping(mapping: RequestMappingHandlerMapping, handler: UserHandler)
{ ①
        val info =
RequestMappingInfo.paths("/user/{id}").methods(RequestMethod.GET).build() ②
        val method = UserHandler::class.java.getMethod("getUser", Long::class.java) ③
        mapping.registerMapping(info, handler, method) ④
    }
}
```

① Inject the target handler and the handler mapping for controllers.

② Prepare the request mapping meta data.

③ Get the handler method.

④ Add the registration.

Handler Methods

WebFlux

`@RequestMapping` handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

Method Arguments

WebFlux

The next table describes the supported controller method arguments. Reactive types are not supported for any arguments.

JDK 8's `java.util.Optional` is supported as a method argument in combination with annotations that have a `required` attribute (for example, `@RequestParam`, `@RequestHeader`, and others) and is equivalent to `required=false`.

Controller method argument	Description
`WebRequest`, `NativeWebRequest`	Generic access to request parameters and request and session attributes, without direct use of the Servlet API.
`jakarta.servlet.ServletRequest`, `jakarta.servlet.ServletResponse`	Choose any specific request or response type — for example, `ServletRequest`, `HttpServletRequest`, or Spring's `MultipartRequest`, `MultipartHttpServletRequest`.
`jakarta.servlet.http.HttpSession`	Enforces the presence of a session. As a consequence, such an argument is never `null`. Note that session access is not thread-safe. Consider setting the `RequestMappingHandlerAdapter` instance's `synchronizeOnSession` flag to `true` if multiple requests are allowed to concurrently access a session.
`jakarta.servlet.http.PushBuilder`	Servlet 4.0 push builder API for programmatic HTTP/2 resource pushes. Note that, per the Servlet specification, the injected `PushBuilder` instance can be null if the client does not support that HTTP/2 feature.
`java.security.Principal`	Currently authenticated user — possibly a specific `Principal` implementation class if known. Note that this argument is not resolved eagerly, if it is annotated in order to allow a custom resolver to resolve it before falling back on default resolution via `HttpServletRequest#getUserPrincipal`. For example, the Spring Security `Authentication` implements `Principal` and would be injected as such via `HttpServletRequest#getUserPrincipal`, unless it is also annotated with `@AuthenticationPrincipal` in which case it is resolved by a custom Spring Security resolver through `Authentication#getPrincipal`.
`HttpMethod`	The HTTP method of the request.

Controller method argument	Description
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available (in effect, the configured `LocaleResolver` or `LocaleContextResolver`).
`java.util.TimeZone` + `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`java.io.InputStream`, `java.io.Reader`	For access to the raw request body as exposed by the Servlet API.
`java.io.OutputStream`, `java.io.Writer`	For access to the raw response body as exposed by the Servlet API.
`@PathVariable`	For access to URI template variables. See URI patterns.
`@MatrixVariable`	For access to name-value pairs in URI path segments. See Matrix Variables.
`@RequestParam`	For access to the Servlet request parameters, including multipart files. Parameter values are converted to the declared method argument type. See `@RequestParam` as well as Multipart. Note that use of `@RequestParam` is optional for simple parameter values. See “Any other argument”, at the end of this table.
`@RequestHeader`	For access to request headers. Header values are converted to the declared method argument type. See `@RequestHeader`.
`@CookieValue`	For access to cookies. Cookies values are converted to the declared method argument type. See `@CookieValue`.
`@RequestBody`	For access to the HTTP request body. Body content is converted to the declared method argument type by using `HttpMessageConverter` implementations. See `@RequestBody`.
`HttpEntity<B>`	For access to request headers and body. The body is converted with an `HttpMessageConverter`. See HttpEntity.
`@RequestPart`	For access to a part in a `multipart/form-data` request, converting the part’s body with an `HttpMessageConverter`. See Multipart.
`java.util.Map`, `org.springframework.ui.Model`, `org.springframework.ui.ModelMap`	For access to the model that is used in HTML controllers and exposed to templates as part of view rendering.
`RedirectAttributes`	Specify attributes to use in case of a redirect (that is, to be appended to the query string) and flash attributes to be stored temporarily until the request after redirect. See Redirect Attributes and Flash Attributes.

Controller method argument	Description
`@ModelAttribute`	For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See `@ModelAttribute` as well as Model and `DataBinder`. Note that use of `@ModelAttribute` is optional (for example, to set its attributes). See “Any other argument” at the end of this table.
`Errors`, `BindingResult`	For access to errors from validation and data binding for a command object (that is, a `@ModelAttribute` argument) or errors from the validation of a `@RequestBody` or `@RequestPart` arguments. You must declare an `Errors`, or `BindingResult` argument immediately after the validated method argument.
`SessionStatus` + class-level `@SessionAttributes`	For marking form processing complete, which triggers cleanup of session attributes declared through a class-level `@SessionAttributes` annotation. See `@SessionAttributes` for more details.
`UriComponentsBuilder`	For preparing a URL relative to the current request’s host, port, scheme, context path, and the literal part of the servlet mapping. See URI Links.
`@SessionAttribute`	For access to any session attribute, in contrast to model attributes stored in the session as a result of a class-level `@SessionAttributes` declaration. See `@SessionAttribute` for more details.
`@RequestAttribute`	For access to request attributes. See `@RequestAttribute` for more details.
Any other argument	If a method argument is not matched to any of the earlier values in this table and it is a simple type (as determined by BeanUtils#isSimpleProperty), it is resolved as a `@RequestParam`. Otherwise, it is resolved as a `@ModelAttribute`.

Return Values

WebFlux

The next table describes the supported controller method return values. Reactive types are supported for all return values.

Controller method return value	Description
`@ResponseBody`	The return value is converted through `HttpMessageConverter` implementations and written to the response. See `@ResponseBody`.
`HttpEntity<B>`, `ResponseEntity<B>`	The return value that specifies the full response (including HTTP headers and body) is to be converted through `HttpMessageConverter` implementations and written to the response. See ResponseEntity.

Controller method return value	Description
`HttpHeaders`	For returning a response with headers and no body.
`ErrorResponse`	To render an RFC 7807 error response with details in the body, see Error Responses
`ProblemDetail`	To render an RFC 7807 error response with details in the body, see Error Responses
`String`	A view name to be resolved with `ViewResolver` implementations and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (see Explicit Registrations).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (see Explicit Registrations).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model, with the view name implicitly determined through a `RequestToViewNameTranslator`.
`@ModelAttribute`	An attribute to be added to the model, with the view name implicitly determined through a `RequestToViewNameTranslator`. Note that `@ModelAttribute` is optional. See "Any other return value" at the end of this table.
`ModelAndView` object	The view and model attributes to use and, optionally, a response status.
`void`	A method with a `void` return type (or `null` return value) is considered to have fully handled the response if it also has a `ServletResponse`, an `OutputStream` argument, or an `@ResponseStatus` annotation. The same is also true if the controller has made a positive `ETag` or `lastModified` timestamp check (see Controllers for details). If none of the above is true, a `void` return type can also indicate “no response body” for REST controllers or a default view name selection for HTML controllers.
`DeferredResult<V>`	Produce any of the preceding return values asynchronously from any thread — for example, as a result of some event or callback. See Asynchronous Requests and `DeferredResult`.
`Callable<V>`	Produce any of the above return values asynchronously in a Spring MVC-managed thread. See Asynchronous Requests and `Callable`.

Controller method return value	Description
`ListenableFuture<V>`, `java.util.concurrent.CompletionStage<V>`, `java.util.concurrent.CompletableFuture<V>`	Alternative to `DeferredResult`, as a convenience (for example, when an underlying service returns one of those).
`ResponseBodyEmitter`, `SseEmitter`	Emit a stream of objects asynchronously to be written to the response with `HttpMessageConverter` implementations. Also supported as the body of a `ResponseEntity`. See Asynchronous Requests and HTTP Streaming.
`StreamingResponseBody`	Write to the response `OutputStream` asynchronously. Also supported as the body of a `ResponseEntity`. See Asynchronous Requests and HTTP Streaming.
Reactor and other reactive types registered via `ReactiveAdapterRegistry`	A single value type, e.g. `Mono`, is comparable to returning `DeferredResult`. A multi-value type, e.g. `Flux`, may be treated as a stream depending on the requested media type, e.g. "text/event-stream", "application/json+stream", or otherwise is collected to a List and rendered as a single value. See Asynchronous Requests and Reactive Types.
Other return values	If a return value remains unresolved in any other way, it is treated as a model attribute, unless it is a simple type as determined by BeanUtils#isSimpleProperty, in which case it remains unresolved.

Type Conversion

WebFlux

Some annotated controller method arguments that represent `String`-based request input (such as `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, and `@CookieValue`) can require type conversion if the argument is declared as something other than `String`.

For such cases, type conversion is automatically applied based on the configured converters. By default, simple types (`int`, `long`, `Date`, and others) are supported. You can customize type conversion through a `WebDataBinder` (see `DataBinder`) or by registering `Formatters` with the `FormattingConversionService`. See Spring Field Formatting.

A practical issue in type conversion is the treatment of an empty String source value. Such a value is treated as missing if it becomes `null` as a result of type conversion. This can be the case for `Long`, `UUID`, and other target types. If you want to allow `null` to be injected, either use the `required` flag on the argument annotation, or declare the argument as `@Nullable`.

As of 5.3, non-null arguments will be enforced even after type conversion. If your handler method intends to accept a null value as well, either declare your argument as `@Nullable` or mark it as `required=false` in the corresponding `@RequestParam`, etc. annotation. This is a best practice and the recommended solution for regressions encountered in a 5.3 upgrade.

Alternatively, you may specifically handle e.g. the resulting `MissingPathVariableException` in the case of a required `@PathVariable`. A null value after conversion will be treated like an empty original value, so the corresponding `Missing···Exception` variants will be thrown.

Matrix Variables

WebFlux

RFC 3986 discusses name-value pairs in path segments. In Spring MVC, we refer to those as "matrix variables" based on an "old post" by Tim Berners-Lee, but they can be also be referred to as URI path parameters.

Matrix variables can appear in any path segment, with each variable separated by a semicolon and multiple values separated by comma (for example, `/cars;color=red,green;year=2012`). Multiple values can also be specified through repeated variable names (for example, `color=red;color=green;color=blue`).

If a URL is expected to contain matrix variables, the request mapping for a controller method must use a URI variable to mask that variable content and ensure the request can be matched successfully independent of matrix variable order and presence. The following example uses a matrix variable:

Java

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
}
```

Kotlin

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
fun findPet(@PathVariable petId: String, @MatrixVariable q: Int) {

    // petId == 42
    // q == 11
}
```

Given that all path segments may contain matrix variables, you may sometimes need to disambiguate which path variable the matrix variable is expected to be in. The following example shows how to do so:

Java

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable(name="q", pathVar="ownerId") int q1,
        @MatrixVariable(name="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22
}
```

Kotlin

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(
        @MatrixVariable(name = "q", pathVar = "ownerId") q1: Int,
        @MatrixVariable(name = "q", pathVar = "petId") q2: Int) {

    // q1 == 11
    // q2 == 22
}
```

A matrix variable may be defined as optional and a default value specified, as the following example shows:

Java

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1
}
```

Kotlin

```
// GET /pets/42

@GetMapping("/pets/{petId}")
fun findPet(@MatrixVariable(required = false, defaultValue = "1") q: Int) {

    // q == 1
}
```

To get all matrix variables, you can use a `MultiValueMap`, as the following example shows:

Java

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable MultiValueMap<String, String> matrixVars,
        @MatrixVariable(pathVar="petId") MultiValueMap<String, String> petMatrixVars)
{

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 22, "s" : 23]
}
```

Kotlin

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(
        @MatrixVariable matrixVars: MultiValueMap<String, String>,
        @MatrixVariable(pathVar="petId") petMatrixVars: MultiValueMap<String, String>)
{

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 22, "s" : 23]
}
```

Note that you need to enable the use of matrix variables. In the MVC Java configuration, you need to set a `UrlPathHelper` with `removeSemicolonContent=false` through Path Matching. In the MVC XML namespace, you can set `<mvc:annotation-driven enable-matrix-variables="true"/>`.

`@RequestParam`

WebFlux

You can use the `@RequestParam` annotation to bind Servlet request parameters (that is, query parameters or form data) to a method argument in a controller.

The following example shows how to do so:

Java

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) { ①
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...

}
```

① Using `@RequestParam` to bind `petId`.

Kotlin

```
import org.springframework.ui.set

@Controller
@RequestMapping("/pets")
class EditPetForm {

    // ...

    @GetMapping
    fun setupForm(@RequestParam("petId") petId: Int, model: Model): String { ①
        val pet = this.clinic.loadPet(petId);
        model["pet"] = pet
        return "petForm"
    }

    // ...

}
```

① Using `@RequestParam` to bind `petId`.

By default, method parameters that use this annotation are required, but you can specify that a method parameter is optional by setting the `@RequestParam` annotation's `required` flag to `false` or by declaring the argument with an `java.util.Optional` wrapper.

Type conversion is automatically applied if the target method parameter type is not `String`. See Type Conversion.

Declaring the argument type as an array or list allows for resolving multiple parameter values for the same parameter name.

When an `@RequestParam` annotation is declared as a `Map<String, String>` or `MultiValueMap<String, String>`, without a parameter name specified in the annotation, then the map is populated with the request parameter values for each given parameter name.

Note that use of `@RequestParam` is optional (for example, to set its attributes). By default, any argument that is a simple value type (as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver, is treated as if it were annotated with `@RequestParam`.

`@RequestHeader`

WebFlux

You can use the `@RequestHeader` annotation to bind a request header to a method argument in a controller.

Consider the following request, with headers:

```
Host                    localhost:8080
Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language         fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding         gzip,deflate
Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive              300
```

The following example gets the value of the `Accept-Encoding` and `Keep-Alive` headers:

Java

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding, ①
        @RequestHeader("Keep-Alive") long keepAlive) { ②
    //...
}
```

① Get the value of the `Accept-Encoding` header.

② Get the value of the `Keep-Alive` header.

Kotlin

```
@GetMapping("/demo")
fun handle(
        @RequestHeader("Accept-Encoding") encoding: String, ①
        @RequestHeader("Keep-Alive") keepAlive: Long) { ②
    //...
}
```

① Get the value of the `Accept-Encoding` header.

② Get the value of the `Keep-Alive` header.

If the target method parameter type is not `String`, type conversion is automatically applied. See Type Conversion.

When an `@RequestHeader` annotation is used on a `Map<String, String>`, `MultiValueMap<String, String>`, or `HttpHeaders` argument, the map is populated with all header values.

Built-in support is available for converting a comma-separated string into an array or collection of strings or other types known to the type conversion system. For example, a method parameter annotated with `@RequestHeader("Accept")` can be of type `String` but also `String[]` or `List<String>`.

`@CookieValue`

WebFlux

You can use the `@CookieValue` annotation to bind the value of an HTTP cookie to a method argument

in a controller.

Consider a request with the following cookie:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

The following example shows how to get the cookie value:

Java

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) { ①
    //...
}
```

① Get the value of the `JSESSIONID` cookie.

Kotlin

```
@GetMapping("/demo")
fun handle(@CookieValue("JSESSIONID") cookie: String) { ①
    //...
}
```

① Get the value of the `JSESSIONID` cookie.

If the target method parameter type is not `String`, type conversion is applied automatically. See Type Conversion.

`@ModelAttribute`

WebFlux

You can use the `@ModelAttribute` annotation on a method argument to access an attribute from the model or have it be instantiated if not present. The model attribute is also overlain with values from HTTP Servlet request parameters whose names match to field names. This is referred to as data binding, and it saves you from having to deal with parsing and converting individual query parameters and form fields. The following example shows how to do so:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { ①
    // method logic...
}
```

① Bind an instance of `Pet`.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute pet: Pet): String { ①
    // method logic...
}
```

① Bind an instance of `Pet`.

The `Pet` instance above is sourced in one of the following ways:

- Retrieved from the model where it may have been added by a @ModelAttribute method.
- Retrieved from the HTTP session if the model attribute was listed in the class-level `@SessionAttributes` annotation.
- Obtained through a `Converter` where the model attribute name matches the name of a request value such as a path variable or a request parameter (see next example).
- Instantiated using its default constructor.
- Instantiated through a "primary constructor" with arguments that match to Servlet request parameters. Argument names are determined through JavaBeans `@ConstructorProperties` or through runtime-retained parameter names in the bytecode.

One alternative to using a @ModelAttribute method to supply it or relying on the framework to create the model attribute, is to have a `Converter<String, T>` to provide the instance. This is applied when the model attribute name matches to the name of a request value such as a path variable or a request parameter, and there is a `Converter` from `String` to the model attribute type. In the following example, the model attribute name is `account` which matches the URI path variable `account`, and there is a registered `Converter<String, Account>` which could load the `Account` from a data store:

Java

```
@PutMapping("/accounts/{account}")
public String save(@ModelAttribute("account") Account account) { ①
    // ...
}
```

① Bind an instance of `Account` using an explicit attribute name.

Kotlin

```
@PutMapping("/accounts/{account}")
fun save(@ModelAttribute("account") account: Account): String { ①
    // ...
}
```

① Bind an instance of `Account` using an explicit attribute name.

After the model attribute instance is obtained, data binding is applied. The `WebDataBinder` class

matches Servlet request parameter names (query parameters and form fields) to field names on the target `Object`. Matching fields are populated after type conversion is applied, where necessary. For more on data binding (and validation), see Validation. For more on customizing data binding, see `DataBinder`.

Data binding can result in errors. By default, a `BindException` is raised. However, to check for such errors in the controller method, you can add a `BindingResult` argument immediately next to the `@ModelAttribute`, as the following example shows:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) { ①
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

① Adding a `BindingResult` next to the `@ModelAttribute`.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute("pet") pet: Pet, result: BindingResult): String { ①
    if (result.hasErrors()) {
        return "petForm"
    }
    // ...
}
```

① Adding a `BindingResult` next to the `@ModelAttribute`.

In some cases, you may want access to a model attribute without data binding. For such cases, you can inject the `Model` into the controller and access it directly or, alternatively, set `@ModelAttribute(binding=false)`, as the following example shows:

Java

```
@ModelAttribute
public AccountForm setUpForm() {
    return new AccountForm();
}

@ModelAttribute
public Account findAccount(@PathVariable String accountId) {
    return accountRepository.findOne(accountId);
}

@PostMapping("update")
public String update(@Valid AccountForm form, BindingResult result,
        @ModelAttribute(binding=false) Account account) { ①
    // ...
}
```

① Setting `@ModelAttribute(binding=false)`.

Kotlin

```
@ModelAttribute
fun setUpForm(): AccountForm {
    return AccountForm()
}

@ModelAttribute
fun findAccount(@PathVariable accountId: String): Account {
    return accountRepository.findOne(accountId)
}

@PostMapping("update")
fun update(@Valid form: AccountForm, result: BindingResult,
           @ModelAttribute(binding = false) account: Account): String { ①
    // ...
}
```

① Setting `@ModelAttribute(binding=false)`.

You can automatically apply validation after data binding by adding the `jakarta.validation.Valid` annotation or Spring's `@Validated` annotation (Bean Validation and Spring validation). The following example shows how to do so:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult
result) { ①
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

① Validate the `Pet` instance.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") pet: Pet, result: BindingResult):
String { ①
    if (result.hasErrors()) {
        return "petForm"
    }
    // ...
}
```

① Validate the `Pet` instance.

Note that using `@ModelAttribute` is optional (for example, to set its attributes). By default, any argument that is not a simple value type (as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver is treated as if it were annotated with `@ModelAttribute`.

`@SessionAttributes`

WebFlux

`@SessionAttributes` is used to store model attributes in the HTTP Servlet session between requests. It is a type-level annotation that declares the session attributes used by a specific controller. This typically lists the names of model attributes or types of model attributes that should be transparently stored in the session for subsequent requests to access.

The following example uses the `@SessionAttributes` annotation:

Java

```
@Controller
@SessionAttributes("pet") ①
public class EditPetForm {
    // ...
}
```

① Using the `@SessionAttributes` annotation.

Kotlin

```
@Controller
@SessionAttributes("pet") ①
class EditPetForm {
    // ...
}
```

① Using the `@SessionAttributes` annotation.

On the first request, when a model attribute with the name, `pet`, is added to the model, it is automatically promoted to and saved in the HTTP Servlet session. It remains there until another controller method uses a `SessionStatus` method argument to clear the storage, as the following example shows:

Java

```
@Controller
@SessionAttributes("pet") ①
public class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    public String handle(Pet pet, BindingResult errors, SessionStatus status) {
        if (errors.hasErrors) {
            // ...
        }
        status.setComplete(); ②
        // ...
    }
}
```

① Storing the `Pet` value in the Servlet session.

② Clearing the `Pet` value from the Servlet session.

Kotlin

```
@Controller
@SessionAttributes("pet") ①
class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    fun handle(pet: Pet, errors: BindingResult, status: SessionStatus): String {
        if (errors.hasErrors()) {
            // ...
        }
        status.setComplete() ②
        // ...
    }
}
```

① Storing the `Pet` value in the Servlet session.

② Clearing the `Pet` value from the Servlet session.

`@SessionAttribute`

WebFlux

If you need access to pre-existing session attributes that are managed globally (that is, outside the controller — for example, by a filter) and may or may not be present, you can use the `@SessionAttribute` annotation on a method parameter, as the following example shows:

Java

```
@RequestMapping("/")
public String handle(@SessionAttribute User user) { ①
    // ...
}
```

① Using a `@SessionAttribute` annotation.

Kotlin

```
@RequestMapping("/")
fun handle(@SessionAttribute user: User): String { ①
    // ...
}
```

① Using a `@SessionAttribute` annotation.

For use cases that require adding or removing session attributes, consider injecting `org.springframework.web.context.request.WebRequest` or `jakarta.servlet.http.HttpSession` into the controller method.

For temporary storage of model attributes in the session as part of a controller workflow, consider using `@SessionAttributes` as described in `@SessionAttributes`.

`@RequestAttribute`

WebFlux

Similar to `@SessionAttribute`, you can use the `@RequestAttribute` annotations to access pre-existing request attributes created earlier (for example, by a Servlet `Filter` or `HandlerInterceptor`):

Java

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) { ①
    // ...
}
```

① Using the `@RequestAttribute` annotation.

Kotlin

```
@GetMapping("/")
fun handle(@RequestAttribute client: Client): String { ①
    // ...
}
```

① Using the `@RequestAttribute` annotation.

Redirect Attributes

By default, all model attributes are considered to be exposed as URI template variables in the redirect URL. Of the remaining attributes, those that are primitive types or collections or arrays of primitive types are automatically appended as query parameters.

Appending primitive type attributes as query parameters can be the desired result if a model instance was prepared specifically for the redirect. However, in annotated controllers, the model can contain additional attributes added for rendering purposes (for example, drop-down field values). To avoid the possibility of having such attributes appear in the URL, a `@RequestMapping` method can declare an argument of type `RedirectAttributes` and use it to specify the exact attributes to make available to `RedirectView`. If the method does redirect, the content of `RedirectAttributes` is used. Otherwise, the content of the model is used.

The `RequestMappingHandlerAdapter` provides a flag called `ignoreDefaultModelOnRedirect`, which you can use to indicate that the content of the default `Model` should never be used if a controller method redirects. Instead, the controller method should declare an attribute of type `RedirectAttributes` or, if it does not do so, no attributes should be passed on to `RedirectView`. Both the MVC namespace and the MVC Java configuration keep this flag set to `false`, to maintain backwards compatibility. However, for new applications, we recommend setting it to `true`.

Note that URI template variables from the present request are automatically made available when expanding a redirect URL, and you don't need to explicitly add them through `Model` or

`RedirectAttributes`. The following example shows how to define a redirect:

Java

```
@PostMapping("/files/{path}")
public String upload(...) {
    // ...
    return "redirect:files/{path}";
}
```

Kotlin

```
@PostMapping("/files/{path}")
fun upload(...): String {
    // ...
    return "redirect:files/{path}"
}
```

Another way of passing data to the redirect target is by using flash attributes. Unlike other redirect attributes, flash attributes are saved in the HTTP session (and, hence, do not appear in the URL). See Flash Attributes for more information.

Flash Attributes

Flash attributes provide a way for one request to store attributes that are intended for use in another. This is most commonly needed when redirecting — for example, the Post-Redirect-Get pattern. Flash attributes are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and are removed immediately.

Spring MVC has two main abstractions in support of flash attributes. `FlashMap` is used to hold flash attributes, while `FlashMapManager` is used to store, retrieve, and manage `FlashMap` instances.

Flash attribute support is always “on” and does not need to be enabled explicitly. However, if not used, it never causes HTTP session creation. On each request, there is an “input” `FlashMap` with attributes passed from a previous request (if any) and an “output” `FlashMap` with attributes to save for a subsequent request. Both `FlashMap` instances are accessible from anywhere in Spring MVC through static methods in `RequestContextUtils`.

Annotated controllers typically do not need to work with `FlashMap` directly. Instead, a `@RequestMapping` method can accept an argument of type `RedirectAttributes` and use it to add flash attributes for a redirect scenario. Flash attributes added through `RedirectAttributes` are automatically propagated to the “output” FlashMap. Similarly, after the redirect, attributes from the “input” `FlashMap` are automatically added to the `Model` of the controller that serves the target URL.

Matching requests to flash attributes

The concept of flash attributes exists in many other web frameworks and has proven to sometimes be exposed to concurrency issues. This is because, by definition, flash attributes are to be stored until the next request. However the very “next” request may not be the intended recipient but another asynchronous request (for example, polling or resource requests), in which case the flash attributes are removed too early.

To reduce the possibility of such issues, `RedirectView` automatically “stamps” `FlashMap` instances with the path and query parameters of the target redirect URL. In turn, the default `FlashMapManager` matches that information to incoming requests when it looks up the “input” `FlashMap`.

This does not entirely eliminate the possibility of a concurrency issue but reduces it greatly with information that is already available in the redirect URL. Therefore, we recommend that you use flash attributes mainly for redirect scenarios.

Multipart

WebFlux

After a `MultipartResolver` has been enabled, the content of POST requests with `multipart/form-data` is parsed and accessible as regular request parameters. The following example accesses one regular form field and one uploaded file:

Java

```
@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(@RequestParam("name") String name,
            @RequestParam("file") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}
```

Kotlin

```
@Controller
class FileUploadController {

    @PostMapping("/form")
    fun handleFormUpload(@RequestParam("name") name: String,
                         @RequestParam("file") file: MultipartFile): String {

        if (!file.isEmpty) {
            val bytes = file.bytes
            // store the bytes somewhere
            return "redirect:uploadSuccess"
        }
        return "redirect:uploadFailure"
    }
}
```

Declaring the argument type as a `List<MultipartFile>` allows for resolving multiple files for the same parameter name.

When the `@RequestParam` annotation is declared as a `Map<String, MultipartFile>` or `MultiValueMap<String, MultipartFile>`, without a parameter name specified in the annotation, then the map is populated with the multipart files for each given parameter name.

With Servlet multipart parsing, you may also declare `jakarta.servlet.http.Part` instead of Spring's `MultipartFile`, as a method argument or collection value type.

You can also use multipart content as part of data binding to a command object. For example, the form field and file from the preceding example could be fields on a form object, as the following example shows:

Java

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...
}

@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(MyForm form, BindingResult errors) {
        if (!form.getFile().isEmpty()) {
            byte[] bytes = form.getFile().getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }
        return "redirect:uploadFailure";
    }
}
```

Kotlin

```
class MyForm(val name: String, val file: MultipartFile, ...)

@Controller
class FileUploadController {

    @PostMapping("/form")
    fun handleFormUpload(form: MyForm, errors: BindingResult): String {
        if (!form.file.isEmpty) {
            val bytes = form.file.bytes
            // store the bytes somewhere
            return "redirect:uploadSuccess"
        }
        return "redirect:uploadFailure"
    }
}
```

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. The following example shows a file with JSON:

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
    "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

You can access the "meta-data" part with `@RequestParam` as a `String` but you'll probably want it deserialized from JSON (similar to `@RequestBody`). Use the `@RequestPart` annotation to access a multipart after converting it with an HttpMessageConverter:

Java

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") MetaData metadata,
        @RequestPart("file-data") MultipartFile file) {
    // ...
}
```

Kotlin

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") metadata: MetaData,
        @RequestPart("file-data") file: MultipartFile): String {
    // ...
}
```

You can use `@RequestPart` in combination with `jakarta.validation.Valid` or use Spring's `@Validated` annotation, both of which cause Standard Bean Validation to be applied. By default, validation errors cause a `MethodArgumentNotValidException`, which is turned into a 400 (BAD_REQUEST) response. Alternatively, you can handle validation errors locally within the controller through an `Errors` or `BindingResult` argument, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@Valid @RequestPart("meta-data") MetaData metadata,
        BindingResult result) {
    // ...
}
```

Kotlin

```
@PostMapping("/")
fun handle(@Valid @RequestPart("meta-data") metadata: MetaData,
        result: BindingResult): String {
    // ...
}
```

`@RequestBody`

WebFlux

You can use the `@RequestBody` annotation to have the request body read and deserialized into an `Object` through an `HttpMessageConverter`. The following example uses a `@RequestBody` argument:

Java

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@RequestBody account: Account) {
    // ...
}
```

You can use the Message Converters option of the MVC Config to configure or customize message conversion.

You can use `@RequestBody` in combination with `jakarta.validation.Valid` or Spring's `@Validated` annotation, both of which cause Standard Bean Validation to be applied. By default, validation errors cause a `MethodArgumentNotValidException`, which is turned into a 400 (BAD_REQUEST) response. Alternatively, you can handle validation errors locally within the controller through an `Errors` or `BindingResult` argument, as the following example shows:

Java

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Account account, BindingResult result) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@Valid @RequestBody account: Account, result: BindingResult) {
    // ...
}
```

HttpEntity

WebFlux

`HttpEntity` is more or less identical to using `@RequestBody` but is based on a container object that exposes request headers and body. The following listing shows an example:

Java

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(entity: HttpEntity<Account>) {
    // ...
}
```

`@ResponseBody`

WebFlux

You can use the `@ResponseBody` annotation on a method to have the return serialized to the response body through an HttpMessageConverter. The following listing shows an example:

Java

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

Kotlin

```
@GetMapping("/accounts/{id}")
@ResponseBody
fun handle(): Account {
    // ...
}
```

`@ResponseBody` is also supported at the class level, in which case it is inherited by all controller methods. This is the effect of `@RestController`, which is nothing more than a meta-annotation marked with `@Controller` and `@ResponseBody`.

You can use `@ResponseBody` with reactive types. See Asynchronous Requests and Reactive Types for more details.

You can use the Message Converters option of the MVC Config to configure or customize message conversion.

You can combine `@ResponseBody` methods with JSON serialization views. See Jackson JSON for details.

ResponseEntity

WebFlux

`ResponseEntity` is like `@ResponseBody` but with status and headers. For example:

Java

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).body(body);
}
```

Kotlin

```
@GetMapping("/something")
fun handle(): ResponseEntity<String> {
    val body = ...
    val etag = ...
    return ResponseEntity.ok().eTag(etag).build(body)
}
```

Spring MVC supports using a single value reactive type to produce the `ResponseEntity` asynchronously, and/or single and multi-value reactive types for the body. This allows the following types of async responses:

- `ResponseEntity<Mono<T>>` or `ResponseEntity<Flux<T>>` make the response status and headers

known immediately while the body is provided asynchronously at a later point. Use `Mono` if the body consists of 0..1 values or `Flux` if it can produce multiple values.

- `Mono<ResponseEntity<T>>` provides all three — response status, headers, and body, asynchronously at a later point. This allows the response status and headers to vary depending on the outcome of asynchronous request handling.

Jackson JSON

Spring offers support for the Jackson JSON library.

JSON Views

WebFlux

Spring MVC provides built-in support for Jackson's Serialization Views, which allow rendering only a subset of all fields in an `Object`. To use it with `@ResponseBody` or `ResponseEntity` controller methods, you can use Jackson's `@JsonView` annotation to activate a serialization view class, as the following example shows:

Java

```
@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }
}

public class User {

    public interface WithoutPasswordView {};
    public interface WithPasswordView extends WithoutPasswordView {};

    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @JsonView(WithoutPasswordView.class)
    public String getUsername() {
        return this.username;
    }

    @JsonView(WithPasswordView.class)
    public String getPassword() {
        return this.password;
    }
}
```

Kotlin

```
@RestController
class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView::class)
    fun getUser() = User("eric", "7!jd#h23")
}

class User(
        @JsonView(WithoutPasswordView::class) val username: String,
        @JsonView(WithPasswordView::class) val password: String) {

    interface WithoutPasswordView
    interface WithPasswordView : WithoutPasswordView
}
```

`@JsonView` allows an array of view classes, but you can specify only one per controller method. If you need to activate multiple views, you can use a composite interface.

If you want to do the above programmatically, instead of declaring an `@JsonView` annotation, wrap the return value with `MappingJacksonValue` and use it to supply the serialization view:

Java

```
@RestController
public class UserController {

    @GetMapping("/user")
    public MappingJacksonValue getUser() {
        User user = new User("eric", "7!jd#h23");
        MappingJacksonValue value = new MappingJacksonValue(user);
        value.setSerializationView(User.WithoutPasswordView.class);
        return value;
    }
}
```

Kotlin

```
@RestController
class UserController {

    @GetMapping("/user")
    fun getUser(): MappingJacksonValue {
        val value = MappingJacksonValue(User("eric", "7!jd#h23"))
        value.serializationView = User.WithoutPasswordView::class.java
        return value
    }
}
```

For controllers that rely on view resolution, you can add the serialization view class to the model, as the following example shows:

Java

```
@Controller
public class UserController extends AbstractController {

    @GetMapping("/user")
    public String getUser(Model model) {
        model.addAttribute("user", new User("eric", "7!jd#h23"));
        model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);
        return "userView";
    }
}
```

Kotlin

```
@Controller
class UserController : AbstractController() {

    @GetMapping("/user")
    fun getUser(model: Model): String {
        model["user"] = User("eric", "7!jd#h23")
        model[JsonView::class.qualifiedName] = User.WithoutPasswordView::class.java
        return "userView"
    }
}
```

Model

WebFlux

You can use the `@ModelAttribute` annotation:

- On a method argument in `@RequestMapping` methods to create or access an `Object` from the model

and to bind it to the request through a `WebDataBinder`.

- As a method-level annotation in `@Controller` or `@ControllerAdvice` classes that help to initialize the model prior to any `@RequestMapping` method invocation.
- On a `@RequestMapping` method to mark its return value is a model attribute.

This section discusses `@ModelAttribute` methods — the second item in the preceding list. A controller can have any number of `@ModelAttribute` methods. All such methods are invoked before `@RequestMapping` methods in the same controller. A `@ModelAttribute` method can also be shared across controllers through `@ControllerAdvice`. See the section on Controller Advice for more details.

`@ModelAttribute` methods have flexible method signatures. They support many of the same arguments as `@RequestMapping` methods, except for `@ModelAttribute` itself or anything related to the request body.

The following example shows a `@ModelAttribute` method:

Java

```
@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountRepository.findAccount(number));
    // add more ...
}
```

Kotlin

```
@ModelAttribute
fun populateModel(@RequestParam number: String, model: Model) {
    model.addAttribute(accountRepository.findAccount(number))
    // add more ...
}
```

The following example adds only one attribute:

Java

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountRepository.findAccount(number);
}
```

Kotlin

```
@ModelAttribute
fun addAccount(@RequestParam number: String): Account {
    return accountRepository.findAccount(number)
}
```

When a name is not explicitly specified, a default name is chosen based on the `Object` type, as explained in the javadoc for `Conventions`. You can always assign an explicit name by using the overloaded `addAttribute` method or through the `name` attribute on `@ModelAttribute` (for a return value).

You can also use `@ModelAttribute` as a method-level annotation on `@RequestMapping` methods, in which case the return value of the `@RequestMapping` method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a `String` that would otherwise be interpreted as a view name. `@ModelAttribute` can also customize the model attribute name, as the following example shows:

Java

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
public Account handle() {
    // ...
    return account;
}
```

Kotlin

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
fun handle(): Account {
    // ...
    return account
}
```

`DataBinder`

WebFlux

`@Controller` or `@ControllerAdvice` classes can have `@InitBinder` methods that initialize instances of `WebDataBinder`, and those, in turn, can:

- Bind request parameters (that is, form or query data) to a model object.
- Convert String-based request values (such as request parameters, path variables, headers, cookies, and others) to the target type of controller method arguments.
- Format model object values as `String` values when rendering HTML forms.

`@InitBinder` methods can register controller-specific `java.beans.PropertyEditor` or Spring `Converter` and `Formatter` components. In addition, you can use the MVC config to register `Converter` and `Formatter` types in a globally shared `FormattingConversionService`.

`@InitBinder` methods support many of the same arguments that `@RequestMapping` methods do, except for `@ModelAttribute` (command object) arguments. Typically, they are declared with a `WebDataBinder` argument (for registrations) and a `void` return value. The following listing shows an example:

Java

```
@Controller
public class FormController {

    @InitBinder ①
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat,
false));
    }

    // ...
}
```

① Defining an `@InitBinder` method.

Kotlin

```
@Controller
class FormController {

    @InitBinder ①
    fun initBinder(binder: WebDataBinder) {
        val dateFormat = SimpleDateFormat("yyyy-MM-dd")
        dateFormat.isLenient = false
        binder.registerCustomEditor(Date::class.java, CustomDateEditor(dateFormat,
false))
    }

    // ...
}
```

① Defining an `@InitBinder` method.

Alternatively, when you use a `Formatter`-based setup through a shared `FormattingConversionService`, you can re-use the same approach and register controller-specific `Formatter` implementations, as the following example shows:

Java

```
@Controller
public class FormController {

    @InitBinder ①
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }

    // ...
}
```

① Defining an `@InitBinder` method on a custom formatter.

Kotlin

```
@Controller
class FormController {

    @InitBinder ①
    protected fun initBinder(binder: WebDataBinder) {
        binder.addCustomFormatter(DateFormatter("yyyy-MM-dd"))
    }

    // ...
}
```

① Defining an `@InitBinder` method on a custom formatter.

Model Design

WebFlux

In the context of web applications, *data binding* involves the binding of HTTP request parameters (that is, form data or query parameters) to properties in a model object and its nested objects.

Only `public` properties following the JavaBeans naming conventions are exposed for data binding — for example, `public String getFirstName()` and `public void setFirstName(String)` methods for a `firstName` property.

The model object, and its nested object graph, is also sometimes referred to as a *command object*, *form-backing object*, or *POJO* (Plain Old Java Object).

By default, Spring permits binding to all public properties in the model object graph. This means you need to carefully consider what public properties the model has, since a client could target any public property path, even some that are not expected to be targeted for a given use case.

For example, given an HTTP form data endpoint, a malicious client could supply values for properties that exist in the model object graph but are not part of the HTML form presented in the

browser. This could lead to data being set on the model object and any of its nested objects, that is not expected to be updated.

The recommended approach is to use a *dedicated model object* that exposes only properties that are relevant for the form submission. For example, on a form for changing a user's email address, the model object should declare a minimum set of properties such as in the following `ChangeEmailForm`.

```
public class ChangeEmailForm {

    private String oldEmailAddress;
    private String newEmailAddress;

    public void setOldEmailAddress(String oldEmailAddress) {
        this.oldEmailAddress = oldEmailAddress;
    }

    public String getOldEmailAddress() {
        return this.oldEmailAddress;
    }

    public void setNewEmailAddress(String newEmailAddress) {
        this.newEmailAddress = newEmailAddress;
    }

    public String getNewEmailAddress() {
        return this.newEmailAddress;
    }

}
```

If you cannot or do not want to use a *dedicated model object* for each data binding use case, you **must** limit the properties that are allowed for data binding. Ideally, you can achieve this by registering *allowed field patterns* via the `setAllowedFields()` method on `WebDataBinder`.

For example, to register allowed field patterns in your application, you can implement an `@InitBinder` method in a `@Controller` or `@ControllerAdvice` component as shown below:

```
@Controller
public class ChangeEmailController {

    @InitBinder
    void initBinder(WebDataBinder binder) {
        binder.setAllowedFields("oldEmailAddress", "newEmailAddress");
    }

    // @RequestMapping methods, etc.

}
```

In addition to registering allowed patterns, it is also possible to register *disallowed field patterns* via the `setDisallowedFields()` method in `DataBinder` and its subclasses. Please note, however, that an "allow list" is safer than a "deny list". Consequently, `setAllowedFields()` should be favored over `setDisallowedFields()`.

Note that matching against allowed field patterns is case-sensitive; whereas, matching against disallowed field patterns is case-insensitive. In addition, a field matching a disallowed pattern will not be accepted even if it also happens to match a pattern in the allowed list.

It is extremely important to properly configure allowed and disallowed field patterns when exposing your domain model directly for data binding purposes. Otherwise, it is a big security risk.

Furthermore, it is strongly recommended that you do **not** use types from your domain model such as JPA or Hibernate entities as the model object in data binding scenarios.

Exceptions

WebFlux

`@Controller` and `@ControllerAdvice` classes can have `@ExceptionHandler` methods to handle exceptions from controller methods, as the following example shows:

Java

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

Kotlin

```
@Controller
class SimpleController {

    // ...

    @ExceptionHandler
    fun handle(ex: IOException): ResponseEntity<String> {
        // ...
    }
}
```

The exception may match against a top-level exception being propagated (e.g. a direct `IOException` being thrown) or against a nested cause within a wrapper exception (e.g. an `IOException` wrapped inside an `IllegalStateException`). As of 5.3, this can match at arbitrary cause levels, whereas previously only an immediate cause was considered.

For matching exception types, preferably declare the target exception as a method argument, as the preceding example shows. When multiple exception methods match, a root exception match is generally preferred to a cause exception match. More specifically, the `ExceptionDepthComparator` is used to sort exceptions based on their depth from the thrown exception type.

Alternatively, the annotation declaration may narrow the exception types to match, as the following example shows:

Java

```
@ExceptionHandler({FileSystemException.class, RemoteException.class})
public ResponseEntity<String> handle(IOException ex) {
    // ...
}
```

Kotlin

```
@ExceptionHandler(FileSystemException::class, RemoteException::class)
fun handle(ex: IOException): ResponseEntity<String> {
    // ...
}
```

You can even use a list of specific exception types with a very generic argument signature, as the following example shows:

Java

```
@ExceptionHandler({FileSystemException.class, RemoteException.class})
public ResponseEntity<String> handle(Exception ex) {
    // ...
}
```

Kotlin

```
@ExceptionHandler(FileSystemException::class, RemoteException::class)
fun handle(ex: Exception): ResponseEntity<String> {
    // ...
}
```

The distinction between root and cause exception matching can be surprising.

In the `IOException` variant shown earlier, the method is typically called with the actual `FileSystemException` or `RemoteException` instance as the argument, since both of them extend from `IOException`. However, if any such matching exception is propagated within a wrapper exception which is itself an `IOException`, the passed-in exception instance is that wrapper exception.

The behavior is even simpler in the `handle(Exception)` variant. This is always invoked with the wrapper exception in a wrapping scenario, with the actually matching exception to be found through `ex.getCause()` in that case. The passed-in exception is the actual `FileSystemException` or `RemoteException` instance only when these are thrown as top-level exceptions.

We generally recommend that you be as specific as possible in the argument signature, reducing the potential for mismatches between root and cause exception types. Consider breaking a multi-matching method into individual `@ExceptionHandler` methods, each matching a single specific exception type through its signature.

In a multi-`@ControllerAdvice` arrangement, we recommend declaring your primary root exception mappings on a `@ControllerAdvice` prioritized with a corresponding order. While a root exception match is preferred to a cause, this is defined among the methods of a given controller or `@ControllerAdvice` class. This means a cause match on a higher-priority `@ControllerAdvice` bean is preferred to any match (for example, root) on a lower-priority `@ControllerAdvice` bean.

Last but not least, an `@ExceptionHandler` method implementation can choose to back out of dealing with a given exception instance by rethrowing it in its original form. This is useful in scenarios where you are interested only in root-level matches or in matches within a specific context that cannot be statically determined. A rethrown exception is propagated through the remaining resolution chain, as though the given `@ExceptionHandler` method would not have matched in the first place.

Support for `@ExceptionHandler` methods in Spring MVC is built on the `DispatcherServlet` level, HandlerExceptionResolver mechanism.

Method Arguments

WebFlux

`@ExceptionHandler` methods support the following arguments:

Method argument	Description
Exception type	For access to the raised exception.
`HandlerMethod`	For access to the controller method that raised the exception.
`WebRequest`, `NativeWebRequest`	Generic access to request parameters and request and session attributes without direct use of the Servlet API.

Method argument	Description
`jakarta.servlet.ServletRequest`, `jakarta.servlet.ServletResponse`	Choose any specific request or response type (for example, `ServletRequest` or `HttpServletRequest` or Spring's `MultipartRequest` or `MultipartHttpServletRequest`).
`jakarta.servlet.http.HttpSession`	Enforces the presence of a session. As a consequence, such an argument is never `null`. Note that session access is not thread-safe. Consider setting the `RequestMappingHandlerAdapter` instance's `synchronizeOnSession` flag to `true` if multiple requests are allowed to access a session concurrently.
`java.security.Principal`	Currently authenticated user — possibly a specific `Principal` implementation class if known.
`HttpMethod`	The HTTP method of the request.
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available — in effect, the configured `LocaleResolver` or `LocaleContextResolver`.
`java.util.TimeZone`, `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`java.io.OutputStream`, `java.io.Writer`	For access to the raw response body, as exposed by the Servlet API.
`java.util.Map`, `org.springframework.ui.Model`, `org.springframework.ui.ModelMap`	For access to the model for an error response. Always empty.
`RedirectAttributes`	Specify attributes to use in case of a redirect — (that is to be appended to the query string) and flash attributes to be stored temporarily until the request after the redirect. See Redirect Attributes and Flash Attributes.
`@SessionAttribute`	For access to any session attribute, in contrast to model attributes stored in the session as a result of a class-level `@SessionAttributes` declaration. See `@SessionAttribute` for more details.
`@RequestAttribute`	For access to request attributes. See `@RequestAttribute` for more details.

Return Values

WebFlux

`@ExceptionHandler` methods support the following return values:

Return value	Description
`@ResponseBody`	The return value is converted through `HttpMessageConverter` instances and written to the response. See `@ResponseBody`.

Return value	Description
`HttpEntity<B>`, `ResponseEntity<B>`	The return value specifies that the full response (including the HTTP headers and the body) be converted through `HttpMessageConverter` instances and written to the response. See ResponseEntity.
`ErrorResponse`	To render an RFC 7807 error response with details in the body, see Error Responses
`ProblemDetail`	To render an RFC 7807 error response with details in the body, see Error Responses
`String`	A view name to be resolved with `ViewResolver` implementations and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method may also programmatically enrich the model by declaring a `Model` argument (descried earlier).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model with the view name implicitly determined through a `RequestToViewNameTranslator`.
`@ModelAttribute`	An attribute to be added to the model with the view name implicitly determined through a `RequestToViewNameTranslator`. Note that `@ModelAttribute` is optional. See “Any other return value” at the end of this table.
`ModelAndView` object	The view and model attributes to use and, optionally, a response status.
`void`	A method with a `void` return type (or `null` return value) is considered to have fully handled the response if it also has a `ServletResponse` an `OutputStream` argument, or a `@ResponseStatus` annotation. The same is also true if the controller has made a positive `ETag` or `lastModified` timestamp check (see Controllers for details). If none of the above is true, a `void` return type can also indicate “no response body” for REST controllers or default view name selection for HTML controllers.
Any other return value	If a return value is not matched to any of the above and is not a simple type (as determined by BeanUtils#isSimpleProperty), by default, it is treated as a model attribute to be added to the model. If it is a simple type, it remains unresolved.

Controller Advice

WebFlux

`@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` methods apply only to the `@Controller` class, or class hierarchy, in which they are declared. If, instead, they are declared in an `@ControllerAdvice` or `@RestControllerAdvice` class, then they apply to any controller. Moreover, as of 5.3, `@ExceptionHandler` methods in `@ControllerAdvice` can be used to handle exceptions from any `@Controller` or any other handler.

`@ControllerAdvice` is meta-annotated with `@Component` and therefore can be registered as a Spring bean through component scanning. `@RestControllerAdvice` is meta-annotated with `@ControllerAdvice` and `@ResponseBody`, and that means `@ExceptionHandler` methods will have their return value rendered via response body message conversion, rather than via HTML views.

On startup, `RequestMappingHandlerMapping` and `ExceptionHandlerExceptionResolver` detect controller advice beans and apply them at runtime. Global `@ExceptionHandler` methods, from an `@ControllerAdvice`, are applied *after* local ones, from the `@Controller`. By contrast, global `@ModelAttribute` and `@InitBinder` methods are applied *before* local ones.

The `@ControllerAdvice` annotation has attributes that let you narrow the set of controllers and handlers that they apply to. For example:

Java

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class,
AbstractController.class})
public class ExampleAdvice3 {}
```

Kotlin

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = [RestController::class])
class ExampleAdvice1

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
class ExampleAdvice2

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = [ControllerInterface::class,
AbstractController::class])
class ExampleAdvice3
```

The selectors in the preceding example are evaluated at runtime and may negatively impact performance if used extensively. See the `@ControllerAdvice` javadoc for more details.

5.1.4. Functional Endpoints

WebFlux

Spring Web MVC includes WebMvc.fn, a lightweight functional programming model in which functions are used to route and handle requests and contracts are designed for immutability. It is an alternative to the annotation-based programming model but otherwise runs on the same DispatcherServlet.

Overview

WebFlux

In WebMvc.fn, an HTTP request is handled with a `HandlerFunction`: a function that takes `ServerRequest` and returns a `ServerResponse`. Both the request and the response object have immutable contracts that offer JDK 8-friendly access to the HTTP request and response. `HandlerFunction` is the equivalent of the body of a `@RequestMapping` method in the annotation-based programming model.

Incoming requests are routed to a handler function with a `RouterFunction`: a function that takes `ServerRequest` and returns an optional `HandlerFunction` (i.e. `Optional<HandlerFunction>`). When the router function matches, a handler function is returned; otherwise an empty Optional. `RouterFunction` is the equivalent of a `@RequestMapping` annotation, but with the major difference that router functions provide not just data, but also behavior.

`RouterFunctions.route()` provides a router builder that facilitates the creation of routers, as the following example shows:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.servlet.function.RequestPredicates.*;
import static org.springframework.web.servlet.function.RouterFunctions.route;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson)
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople)
    .POST("/person", handler::createPerson)
    .build();


public class PersonHandler {

    // ...

    public ServerResponse listPeople(ServerRequest request) {
        // ...
    }

    public ServerResponse createPerson(ServerRequest request) {
        // ...
    }

    public ServerResponse getPerson(ServerRequest request) {
        // ...
    }
}
```

Kotlin

```
import org.springframework.web.servlet.function.router

val repository: PersonRepository = ...
val handler = PersonHandler(repository)

val route = router { ①
    accept(APPLICATION_JSON).nest {
        GET("/person/{id}", handler::getPerson)
        GET("/person", handler::listPeople)
    }
    POST("/person", handler::createPerson)
}


class PersonHandler(private val repository: PersonRepository) {

    // ...

    fun listPeople(request: ServerRequest): ServerResponse {
        // ...
    }

    fun createPerson(request: ServerRequest): ServerResponse {
        // ...
    }

    fun getPerson(request: ServerRequest): ServerResponse {
        // ...
    }
}
```

① Create router using the router DSL.

If you register the `RouterFunction` as a bean, for instance by exposing it in a `@Configuration` class, it will be auto-detected by the servlet, as explained in Running a Server.

HandlerFunction

WebFlux

`ServerRequest` and `ServerResponse` are immutable interfaces that offer JDK 8-friendly access to the HTTP request and response, including headers, body, method, and status code.

ServerRequest

`ServerRequest` provides access to the HTTP method, URI, headers, and query parameters, while access to the body is provided through the `body` methods.

The following example extracts the request body to a `String`:

Java

```
String string = request.body(String.class);
```

Kotlin

```
val string = request.body<String>()
```

The following example extracts the body to a `List<Person>`, where `Person` objects are decoded from a serialized form, such as JSON or XML:

Java

```
List<Person> people = request.body(new ParameterizedTypeReference<List<Person>>() {});
```

Kotlin

```
val people = request.body<Person>()
```

The following example shows how to access parameters:

Java

```
MultiValueMap<String, String> params = request.params();
```

Kotlin

```
val map = request.params()
```

ServerResponse

`ServerResponse` provides access to the HTTP response and, since it is immutable, you can use a `build` method to create it. You can use the builder to set the response status, to add response headers, or to provide a body. The following example creates a 200 (OK) response with JSON content:

Java

```
Person person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person);
```

Kotlin

```
val person: Person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person)
```

The following example shows how to build a 201 (CREATED) response with a `Location` header and

no body:

Java

```
URI location = ...
ServerResponse.created(location).build();
```

Kotlin

```
val location: URI = ...
ServerResponse.created(location).build()
```

You can also use an asynchronous result as the body, in the form of a `CompletableFuture`, `Publisher`, or any other type supported by the `ReactiveAdapterRegistry`. For instance:

Java

```
Mono<Person> person = webClient.get().retrieve().bodyToMono(Person.class);
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person);
```

Kotlin

```
val person = webClient.get().retrieve().awaitBody<Person>()
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person)
```

If not just the body, but also the status or headers are based on an asynchronous type, you can use the static `async` method on `ServerResponse`, which accepts `CompletableFuture<ServerResponse>`, `Publisher<ServerResponse>`, or any other asynchronous type supported by the `ReactiveAdapterRegistry`. For instance:

Java

```
Mono<ServerResponse> asyncResponse =
webClient.get().retrieve().bodyToMono(Person.class)
  .map(p -> ServerResponse.ok().header("Name", p.name()).body(p));
ServerResponse.async(asyncResponse);
```

Server-Sent Events can be provided via the static `sse` method on `ServerResponse`. The builder provided by that method allows you to send Strings, or other objects as JSON. For example:

Java

```
public RouterFunction<ServerResponse> sse() {
    return route(GET("/sse"), request -> ServerResponse.sse(sseBuilder -> {
                // Save the sseBuilder object somewhere..
            }));
}

// In some other thread, sending a String
sseBuilder.send("Hello world");

// Or an object, which will be transformed into JSON
Person person = ...
sseBuilder.send(person);

// Customize the event by using the other methods
sseBuilder.id("42")
        .event("sse event")
        .data(person);

// and done at some point
sseBuilder.complete();
```

Kotlin

```
fun sse(): RouterFunction<ServerResponse> = router {
    GET("/sse") { request -> ServerResponse.sse { sseBuilder ->
        // Save the sseBuilder object somewhere..
    }
}

// In some other thread, sending a String
sseBuilder.send("Hello world")

// Or an object, which will be transformed into JSON
val person = ...
sseBuilder.send(person)

// Customize the event by using the other methods
sseBuilder.id("42")
        .event("sse event")
        .data(person)

// and done at some point
sseBuilder.complete()
```

Handler Classes

We can write a handler function as a lambda, as the following example shows:

Java

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().body("Hello World");
```

Kotlin

```
val helloWorld: (ServerRequest) -> ServerResponse =
  { ServerResponse.ok().body("Hello World") }
```

That is convenient, but in an application we need multiple functions, and multiple inline lambda's can get messy. Therefore, it is useful to group related handler functions together into a handler class, which has a similar role as `@Controller` in an annotation-based application. For example, the following class exposes a reactive `Person` repository:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;

public class PersonHandler {

    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public ServerResponse listPeople(ServerRequest request) { ①
        List<Person> people = repository.allPeople();
        return ok().contentType(APPLICATION_JSON).body(people);
    }

    public ServerResponse createPerson(ServerRequest request) throws Exception { ②
        Person person = request.body(Person.class);
        repository.savePerson(person);
        return ok().build();
    }

    public ServerResponse getPerson(ServerRequest request) { ③
        int personId = Integer.parseInt(request.pathVariable("id"));
        Person person = repository.getPerson(personId);
        if (person != null) {
            return ok().contentType(APPLICATION_JSON).body(person);
        }
        else {
            return ServerResponse.notFound().build();
        }
    }

}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `createPerson` is a handler function that stores a new `Person` contained in the request body.

③ `getPerson` is a handler function that returns a single person, identified by the `id` path variable. We retrieve that `Person` from the repository and create a JSON response, if it is found. If it is not found, we return a 404 Not Found response.

Kotlin

```
class PersonHandler(private val repository: PersonRepository) {

    fun listPeople(request: ServerRequest): ServerResponse { ①
        val people: List<Person> = repository.allPeople()
        return ok().contentType(APPLICATION_JSON).body(people);
    }

    fun createPerson(request: ServerRequest): ServerResponse { ②
        val person = request.body<Person>()
        repository.savePerson(person)
        return ok().build()
    }

    fun getPerson(request: ServerRequest): ServerResponse { ③
        val personId = request.pathVariable("id").toInt()
        return repository.getPerson(personId)?.let {
ok().contentType(APPLICATION_JSON).body(it) }
                ?: ServerResponse.notFound().build()

    }
}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `createPerson` is a handler function that stores a new `Person` contained in the request body.

③ `getPerson` is a handler function that returns a single person, identified by the `id` path variable. We retrieve that `Person` from the repository and create a JSON response, if it is found. If it is not found, we return a 404 Not Found response.

Validation

A functional endpoint can use Spring's validation facilities to apply validation to the request body. For example, given a custom Spring Validator implementation for a `Person`:

Java

```
public class PersonHandler {

    private final Validator validator = new PersonValidator(); ①

    // ...

    public ServerResponse createPerson(ServerRequest request) {
        Person person = request.body(Person.class);
        validate(person); ②
        repository.savePerson(person);
        return ok().build();
    }

    private void validate(Person person) {
        Errors errors = new BeanPropertyBindingResult(person, "person");
        validator.validate(person, errors);
        if (errors.hasErrors()) {
            throw new ServerWebInputException(errors.toString()); ③
        }
    }
}
```

① Create `Validator` instance.

② Apply validation.

③ Raise exception for a 400 response.

Kotlin

```
class PersonHandler(private val repository: PersonRepository) {

    private val validator = PersonValidator() ①

    // ...

    fun createPerson(request: ServerRequest): ServerResponse {
        val person = request.body<Person>()
        validate(person) ②
        repository.savePerson(person)
        return ok().build()
    }

    private fun validate(person: Person) {
        val errors: Errors = BeanPropertyBindingResult(person, "person")
        validator.validate(person, errors)
        if (errors.hasErrors()) {
            throw ServerWebInputException(errors.toString()) ③
        }
    }
}
```

① Create `Validator` instance.

② Apply validation.

③ Raise exception for a 400 response.

Handlers can also use the standard bean validation API (JSR-303) by creating and injecting a global `Validator` instance based on `LocalValidatorFactoryBean`. See Spring Validation.

`RouterFunction`

WebFlux

Router functions are used to route the requests to the corresponding `HandlerFunction`. Typically, you do not write router functions yourself, but rather use a method on the `RouterFunctions` utility class to create one. `RouterFunctions.route()` (no parameters) provides you with a fluent builder for creating a router function, whereas `RouterFunctions.route(RequestPredicate, HandlerFunction)` offers a direct way to create a router.

Generally, it is recommended to use the `route()` builder, as it provides convenient short-cuts for typical mapping scenarios without requiring hard-to-discover static imports. For instance, the router function builder offers the method `GET(String, HandlerFunction)` to create a mapping for GET requests; and `POST(String, HandlerFunction)` for POSTs.

Besides HTTP method-based mapping, the route builder offers a way to introduce additional predicates when mapping to requests. For each HTTP method there is an overloaded variant that takes a `RequestPredicate` as a parameter, through which additional constraints can be expressed.

Predicates

You can write your own `RequestPredicate`, but the `RequestPredicates` utility class offers commonly used implementations, based on the request path, HTTP method, content-type, and so on. The following example uses a request predicate to create a constraint based on the `Accept` header:

Java

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN),
        request -> ServerResponse.ok().body("Hello World")).build();
```

Kotlin

```
import org.springframework.web.servlet.function.router

val route = router {
    GET("/hello-world", accept(TEXT_PLAIN)) {
        ServerResponse.ok().body("Hello World")
    }
}
```

You can compose multiple request predicates together by using:

- `RequestPredicate.and(RequestPredicate)` — both must match.
- `RequestPredicate.or(RequestPredicate)` — either can match.

Many of the predicates from `RequestPredicates` are composed. For example, `RequestPredicates.GET(String)` is composed from `RequestPredicates.method(HttpMethod)` and `RequestPredicates.path(String)`. The example shown above also uses two request predicates, as the builder uses `RequestPredicates.GET` internally, and composes that with the `accept` predicate.

Routes

Router functions are evaluated in order: if the first route does not match, the second is evaluated, and so on. Therefore, it makes sense to declare more specific routes before general ones. This is also important when registering router functions as Spring beans, as will be described later. Note that this behavior is different from the annotation-based programming model, where the "most specific" controller method is picked automatically.

When using the router function builder, all defined routes are composed into one `RouterFunction` that is returned from `build()`. There are also other ways to compose multiple router functions together:

- `add(RouterFunction)` on the `RouterFunctions.route()` builder
- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)` — shortcut for `RouterFunction.and()` with nested `RouterFunctions.route()`.

The following example shows the composition of four routes:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.servlet.function.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> otherRoute = ...

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ①
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople) ②
    .POST("/person", handler::createPerson) ③
    .add(otherRoute) ④
    .build();
```

① `GET /person/{id}` with an `Accept` header that matches JSON is routed to `PersonHandler.getPerson`

② `GET /person` with an `Accept` header that matches JSON is routed to `PersonHandler.listPeople`

③ `POST /person` with no additional predicates is mapped to `PersonHandler.createPerson`, and

④ `otherRoute` is a router function that is created elsewhere, and added to the route built.

Kotlin

```
import org.springframework.http.MediaType.APPLICATION_JSON
import org.springframework.web.servlet.function.router

val repository: PersonRepository = ...
val handler = PersonHandler(repository);

val otherRoute = router {  }

val route = router {
    GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ①
    GET("/person", accept(APPLICATION_JSON), handler::listPeople) ②
    POST("/person", handler::createPerson) ③
}.and(otherRoute) ④
```

① `GET /person/{id}` with an `Accept` header that matches JSON is routed to `PersonHandler.getPerson`

② `GET /person` with an `Accept` header that matches JSON is routed to `PersonHandler.listPeople`

③ `POST /person` with no additional predicates is mapped to `PersonHandler.createPerson`, and

④ `otherRoute` is a router function that is created elsewhere, and added to the route built.

Nested Routes

It is common for a group of router functions to have a shared predicate, for instance a shared path.

In the example above, the shared predicate would be a path predicate that matches `/person`, used by three of the routes. When using annotations, you would remove this duplication by using a type-level `@RequestMapping` annotation that maps to `/person`. In WebMvc.fn, path predicates can be shared through the `path` method on the router function builder. For instance, the last few lines of the example above can be improved in the following way by using nested routes:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", builder -> builder ①
        .GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        .GET(accept(APPLICATION_JSON), handler::listPeople)
        .POST(handler::createPerson))
    .build();
```

① Note that second parameter of `path` is a consumer that takes the router builder.

Kotlin

```
import org.springframework.web.servlet.function.router

val route = router {
    "/person".nest { ①
        GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        GET(accept(APPLICATION_JSON), handler::listPeople)
        POST(handler::createPerson)
    }
}
```

① Using `nest` DSL.

Though path-based nesting is the most common, you can nest on any kind of predicate by using the `nest` method on the builder. The above still contains some duplication in the form of the shared `Accept`-header predicate. We can further improve by using the `nest` method together with `accept`:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople))
        .POST(handler::createPerson))
    .build();
```

Kotlin

```
import org.springframework.web.servlet.function.router

val route = router {
    "/person".nest {
        accept(APPLICATION_JSON).nest {
            GET("/{id}", handler::getPerson)
            GET("", handler::listPeople)
            POST(handler::createPerson)
        }
    }
}
```

Running a Server

WebFlux

You typically run router functions in a `DispatcherHandler`-based setup through the MVC Config, which uses Spring configuration to declare the components required to process requests. The MVC Java configuration declares the following infrastructure components to support functional endpoints:

- `RouterFunctionMapping`: Detects one or more `RouterFunction<?>` beans in the Spring configuration, orders them, combines them through `RouterFunction.andOther`, and routes requests to the resulting composed `RouterFunction`.
- `HandlerFunctionAdapter`: Simple adapter that lets `DispatcherHandler` invoke a `HandlerFunction` that was mapped to a request.

The preceding components let functional endpoints fit within the `DispatcherServlet` request processing lifecycle and also (potentially) run side by side with annotated controllers, if any are declared. It is also how functional endpoints are enabled by the Spring Boot Web starter.

The following example shows a WebFlux Java configuration:

Java

```
@Configuration
@EnableMvc
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public RouterFunction<?> routerFunctionA() {
        // ...
    }

    @Bean
    public RouterFunction<?> routerFunctionB() {
        // ...
    }

    // ...

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        // configure message conversion...
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        // configure CORS...
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // configure view resolution for HTML rendering...
    }
}
```

Kotlin

```
@Configuration
@EnableMvc
class WebConfig : WebMvcConfigurer {

    @Bean
    fun routerFunctionA(): RouterFunction<*> {
        // ...
    }

    @Bean
    fun routerFunctionB(): RouterFunction<*> {
        // ...
    }

    // ...

    override fun configureMessageConverters(converters: List<HttpMessageConverter<*>>)
{
        // configure message conversion...
    }

    override fun addCorsMappings(registry: CorsRegistry) {
        // configure CORS...
    }

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        // configure view resolution for HTML rendering...
    }
}
```

Filtering Handler Functions

WebFlux

You can filter handler functions by using the `before`, `after`, or `filter` methods on the routing function builder. With annotations, you can achieve similar functionality by using `@ControllerAdvice`, a `ServletFilter`, or both. The filter will apply to all routes that are built by the builder. This means that filters defined in nested routes do not apply to "top-level" routes. For instance, consider the following example:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople)
            .before(request -> ServerRequest.from(request) ①
                .header("X-RequestHeader", "Value")
                .build()))
        .POST(handler::createPerson))
    .after((request, response) -> logResponse(response)) ②
    .build();
```

① The `before` filter that adds a custom request header is only applied to the two GET routes.

② The `after` filter that logs the response is applied to all routes, including the nested ones.

Kotlin

```
import org.springframework.web.servlet.function.router

val route = router {
    "/person".nest {
        GET("/{id}", handler::getPerson)
        GET(handler::listPeople)
        before { ①
            ServerRequest.from(it)
                    .header("X-RequestHeader", "Value").build()
        }
    }
    POST(handler::createPerson)
    after { _, response -> ②
        logResponse(response)
    }
}
```

① The `before` filter that adds a custom request header is only applied to the two GET routes.

② The `after` filter that logs the response is applied to all routes, including the nested ones.

The `filter` method on the router builder takes a `HandlerFilterFunction`: a function that takes a `ServerRequest` and `HandlerFunction` and returns a `ServerResponse`. The handler function parameter represents the next element in the chain. This is typically the handler that is routed to, but it can also be another filter if multiple are applied.

Now we can add a simple security filter to our route, assuming that we have a `SecurityManager` that can determine whether a particular path is allowed. The following example shows how to do so:

Java

```
SecurityManager securityManager = ...

RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople))
        .POST(handler::createPerson))
    .filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    })
    .build();
```

Kotlin

```
import org.springframework.web.servlet.function.router

val securityManager: SecurityManager = ...

val route = router {
    ("/person" and accept(APPLICATION_JSON)).nest {
        GET("/{id}", handler::getPerson)
        GET("", handler::listPeople)
        POST(handler::createPerson)
        filter { request, next ->
            if (securityManager.allowAccessTo(request.path())) {
                next(request)
            }
            else {
                status(UNAUTHORIZED).build();
            }
        }
    }
}
```

The preceding example demonstrates that invoking the `next.handle(ServerRequest)` is optional. We only let the handler function be run when access is allowed.

Besides using the `filter` method on the router function builder, it is possible to apply a filter to an existing router function via `RouterFunction.filter(HandlerFilterFunction)`.

CORS support for functional endpoints is provided through a dedicated `CorsFilter`.

5.1.5. URI Links

WebFlux

This section describes various options available in the Spring Framework to work with URI's.

UriComponents

Spring MVC and Spring WebFlux

`UriComponentsBuilder` helps to build URI's from URI templates with variables, as the following example shows:

Java

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")  ①
        .queryParam("q", "{q}")  ②
        .encode() ③
        .build(); ④

URI uri = uriComponents.expand("Westin", "123").toUri();  ⑤
```

① Static factory method with a URI template.

② Add or replace URI components.

③ Request to have the URI template and URI variables encoded.

④ Build a `UriComponents`.

⑤ Expand variables and obtain the `URI`.

Kotlin

```
val uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")  ①
        .queryParam("q", "{q}")  ②
        .encode() ③
        .build() ④

val uri = uriComponents.expand("Westin", "123").toUri()  ⑤
```

① Static factory method with a URI template.

② Add or replace URI components.

③ Request to have the URI template and URI variables encoded.

④ Build a `UriComponents`.

⑤ Expand variables and obtain the `URI`.

The preceding example can be consolidated into one chain and shortened with `buildAndExpand`, as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri();
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri()
```

You can shorten it further by going directly to a URI (which implies encoding), as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123")
```

You can shorten it further still with a full URI template, as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123");
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123")
```

UriBuilder

Spring MVC and Spring WebFlux

`UriComponentsBuilder` implements `UriBuilder`. You can create a `UriBuilder`, in turn, with a `UriBuilderFactory`. Together, `UriBuilderFactory` and `UriBuilder` provide a pluggable mechanism to build URIs from URI templates, based on shared configuration, such as a base URL, encoding preferences, and other details.

You can configure `RestTemplate` and `WebClient` with a `UriBuilderFactory` to customize the preparation of URIs. `DefaultUriBuilderFactory` is a default implementation of `UriBuilderFactory` that uses `UriComponentsBuilder` internally and exposes shared configuration options.

The following example shows how to configure a `RestTemplate`:

Java

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

Kotlin

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode

val baseUrl = "https://example.org"
val factory = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val restTemplate = RestTemplate()
restTemplate.uriTemplateHandler = factory
```

The following example configures a `WebClient`:

Java

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

Kotlin

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode

val baseUrl = "https://example.org"
val factory = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val client = WebClient.builder().uriBuilderFactory(factory).build()
```

In addition, you can also use `DefaultUriBuilderFactory` directly. It is similar to using `UriComponentsBuilder` but, instead of static factory methods, it is an actual instance that holds configuration and preferences, as the following example shows:

Java

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

Kotlin

```
val baseUrl = "https://example.com"
val uriBuilderFactory = DefaultUriBuilderFactory(baseUrl)

val uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123")
```

URI Encoding

Spring MVC and Spring WebFlux

`UriComponentsBuilder` exposes encoding options at two levels:

- UriComponentsBuilder#encode(): Pre-encodes the URI template first and then strictly encodes

URI variables when expanded.

- UriComponents#encode(): Encodes URI components *after* URI variables are expanded.

Both options replace non-ASCII and illegal characters with escaped octets. However, the first option also replaces characters with reserved meaning that appear in URI variables.

Consider ";", which is legal in a path but has reserved meaning. The first option replaces ";" with "%3B" in URI variables but not in the URI template. By contrast, the second option never replaces ";", since it is a legal character in a path.

For most cases, the first option is likely to give the expected result, because it treats URI variables as opaque data to be fully encoded, while the second option is useful if URI variables do intentionally contain reserved characters. The second option is also useful when not expanding URI variables at all since that will also encode anything that incidentally looks like a URI variable.

The following example uses the first option:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri();

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri()

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

You can shorten the preceding example by going directly to the URI (which implies encoding), as the following example shows:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar");
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar")
```

You can shorten it further still with a full URI template, as the following example shows:

Java

```
URI uri = UriComponentsBuilder.fromUriString("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar");
```

Kotlin

```
val uri = UriComponentsBuilder.fromUriString("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar")
```

The `WebClient` and the `RestTemplate` expand and encode URI templates internally through the `UriBuilderFactory` strategy. Both can be configured with a custom strategy, as the following example shows:

Java

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// Customize the RestTemplate..
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..
WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

Kotlin

```
val baseUrl = "https://example.com"
val factory = DefaultUriBuilderFactory(baseUrl).apply {
    encodingMode = EncodingMode.TEMPLATE_AND_VALUES
}

// Customize the RestTemplate..
val restTemplate = RestTemplate().apply {
    uriTemplateHandler = factory
}

// Customize the WebClient..
val client = WebClient.builder().uriBuilderFactory(factory).build()
```

The `DefaultUriBuilderFactory` implementation uses `UriComponentsBuilder` internally to expand and encode URI templates. As a factory, it provides a single place to configure the approach to encoding, based on one of the below encoding modes:

- `TEMPLATE_AND_VALUES`: Uses `UriComponentsBuilder#encode()`, corresponding to the first option in the earlier list, to pre-encode the URI template and strictly encode URI variables when expanded.
- `VALUES_ONLY`: Does not encode the URI template and, instead, applies strict encoding to URI variables through `UriUtils#encodeUriVariables` prior to expanding them into the template.
- `URI_COMPONENT`: Uses `UriComponents#encode()`, corresponding to the second option in the earlier list, to encode URI component value *after* URI variables are expanded.
- `NONE`: No encoding is applied.

The `RestTemplate` is set to `EncodingMode.URI_COMPONENT` for historic reasons and for backwards compatibility. The `WebClient` relies on the default value in `DefaultUriBuilderFactory`, which was changed from `EncodingMode.URI_COMPONENT` in 5.0.x to `EncodingMode.TEMPLATE_AND_VALUES` in 5.1.

Relative Servlet Requests

You can use `ServletUriComponentsBuilder` to create URIs relative to the current request, as the following example shows:

Java

```
HttpServletRequest request = ...

// Re-uses scheme, host, port, path, and query string...

URI uri = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}")
        .build("123");
```

Kotlin

```
val request: HttpServletRequest = ...

// Re-uses scheme, host, port, path, and query string...

val uri = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}")
        .build("123")
```

You can create URIs relative to the context path, as the following example shows:

Java

```
HttpServletRequest request = ...

// Re-uses scheme, host, port, and context path...

URI uri = ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts")
        .build()
        .toUri();
```

Kotlin

```
val request: HttpServletRequest = ...

// Re-uses scheme, host, port, and context path...

val uri = ServletUriComponentsBuilder.fromContextPath(request)
        .path("/accounts")
        .build()
        .toUri()
```

You can create URIs relative to a Servlet (for example, `/main/*`), as the following example shows:

Java

```
HttpServletRequest request = ...

// Re-uses scheme, host, port, context path, and Servlet mapping prefix...

URI uri = ServletUriComponentsBuilder.fromServletMapping(request)
        .path("/accounts")
        .build()
        .toUri();
```

Kotlin

```
val request: HttpServletRequest = ...

// Re-uses scheme, host, port, context path, and Servlet mapping prefix...

val uri = ServletUriComponentsBuilder.fromServletMapping(request)
        .path("/accounts")
        .build()
        .toUri()
```

As of 5.1, `ServletUriComponentsBuilder` ignores information from the `Forwarded` and `X-Forwarded-*` headers, which specify the client-originated address. Consider using the `ForwardedHeaderFilter` to extract and use or to discard such headers.

Links to Controllers

Spring MVC provides a mechanism to prepare links to controller methods. For example, the following MVC controller allows for link creation:

Java

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @GetMapping("/bookings/{booking}")
    public ModelAndView getBooking(@PathVariable Long booking) {
        // ...
    }
}
```

Kotlin

```
@Controller
@RequestMapping("/hotels/{hotel}")
class BookingController {

    @GetMapping("/bookings/{booking}")
    fun getBooking(@PathVariable booking: Long): ModelAndView {
        // ...
    }
}
```

You can prepare a link by referring to the method by name, as the following example shows:

Java

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

Kotlin

```
val uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController::class.java, "getBooking",
21).buildAndExpand(42)

val uri = uriComponents.encode().toUri()
```

In the preceding example, we provide actual method argument values (in this case, the long value: `21`) to be used as a path variable and inserted into the URL. Furthermore, we provide the value, `42`, to fill in any remaining URI variables, such as the `hotel` variable inherited from the type-level request mapping. If the method had more arguments, we could supply null for arguments not needed for the URL. In general, only `@PathVariable` and `@RequestParam` arguments are relevant for constructing the URL.

There are additional ways to use `MvcUriComponentsBuilder`. For example, you can use a technique akin to mock testing through proxies to avoid referring to the controller method by name, as the following example shows (the example assumes static import of `MvcUriComponentsBuilder.on`):

Java

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

Kotlin

```
val uriComponents = MvcUriComponentsBuilder

.fromMethodCall(on(BookingController::class.java).getBooking(21)).buildAndExpand(42)

val uri = uriComponents.encode().toUri()
```

Controller method signatures are limited in their design when they are supposed to be usable for link creation with `fromMethodCall`. Aside from needing a proper parameter signature, there is a technical limitation on the return type (namely, generating a runtime proxy for link builder invocations), so the return type must not be `final`. In particular, the common `String` return type for view names does not work here. You should use `ModelAndView` or even plain `Object` (with a `String` return value) instead.

The earlier examples use static methods in `MvcUriComponentsBuilder`. Internally, they rely on `ServletUriComponentsBuilder` to prepare a base URL from the scheme, host, port, context path, and servlet path of the current request. This works well in most cases. However, sometimes, it can be insufficient. For example, you may be outside the context of a request (such as a batch process that prepares links) or perhaps you need to insert a path prefix (such as a locale prefix that was removed from the request path and needs to be re-inserted into links).

For such cases, you can use the static `fromXxx` overloaded methods that accept a `UriComponentsBuilder` to use a base URL. Alternatively, you can create an instance of `MvcUriComponentsBuilder` with a base URL and then use the instance-based `withXxx` methods. For example, the following listing uses `withMethodCall`:

Java

```
UriComponentsBuilder base =
ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");
MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

Kotlin

```
val base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en")
val builder = MvcUriComponentsBuilder.relativeTo(base)
builder.withMethodCall(on(BookingController::class.java).getBooking(21)).buildAndExpan
d(42)

val uri = uriComponents.encode().toUri()
```

As of 5.1, `MvcUriComponentsBuilder` ignores information from the `Forwarded` and `X-Forwarded-*` headers, which specify the client-originated address. Consider using the ForwardedHeaderFilter to extract and use or to discard such headers.

Links in Views

In views such as Thymeleaf, FreeMarker, or JSP, you can build links to annotated controllers by referring to the implicitly or explicitly assigned name for each request mapping.

Consider the following example:

Java

```
@RequestMapping("/people/{id}/addresses")
public class PersonAddressController {

    @RequestMapping("/{country}")
    public HttpEntity<PersonAddress> getAddress(@PathVariable String country) { ... }
}
```

Kotlin

```
@RequestMapping("/people/{id}/addresses")
class PersonAddressController {

    @RequestMapping("/{country}")
    fun getAddress(@PathVariable country: String): HttpEntity<PersonAddress> { ... }
}
```

Given the preceding controller, you can prepare a link from a JSP, as follows:

```
<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>
...
<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get
Address</a>
```

The preceding example relies on the `mvcUrl` function declared in the Spring tag library (that is, META-INF/spring.tld), but it is easy to define your own function or prepare a similar one for other templating technologies.

Here is how this works. On startup, every `@RequestMapping` is assigned a default name through `HandlerMethodMappingNamingStrategy`, whose default implementation uses the capital letters of the class and the method name (for example, the `getThing` method in `ThingController` becomes "TC#getThing"). If there is a name clash, you can use `@RequestMapping(name="..")` to assign an explicit name or implement your own `HandlerMethodMappingNamingStrategy`.

5.1.6. Asynchronous Requests

Compared to WebFlux

Spring MVC has an extensive integration with Servlet asynchronous request processing:

- `DeferredResult` and `Callable` return values in controller methods provide basic support for a single asynchronous return value.
- Controllers can stream multiple values, including SSE and raw data.
- Controllers can use reactive clients and return reactive types for response handling.

`DeferredResult`

Compared to WebFlux

Once the asynchronous request processing feature is enabled in the Servlet container, controller methods can wrap any supported controller method return value with `DeferredResult`, as the following example shows:

Java

```
@GetMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
    DeferredResult<String> deferredResult = new DeferredResult<String>();
    // Save the deferredResult somewhere..
    return deferredResult;
}

// From some other thread...
deferredResult.setResult(result);
```

Kotlin

```
@GetMapping("/quotes")
@ResponseBody
fun quotes(): DeferredResult<String> {
    val deferredResult = DeferredResult<String>()
    // Save the deferredResult somewhere..
    return deferredResult
}

// From some other thread...
deferredResult.setResult(result)
```

The controller can produce the return value asynchronously, from a different thread — for example, in response to an external event (JMS message), a scheduled task, or other event.

`Callable`

Compared to WebFlux

A controller can wrap any supported return value with `java.util.concurrent.Callable`, as the following example shows:

Java

```
@PostMapping
public Callable<String> processUpload(final MultipartFile file) {
    return () -> "someView";
}
```

Kotlin

```
@PostMapping
fun processUpload(file: MultipartFile) = Callable<String> {
    // ...
    "someView"
}
```

The return value can then be obtained by running the given task through the configured `TaskExecutor`.

Processing

Compared to WebFlux

Here is a very concise overview of Servlet asynchronous request processing:

- A `ServletRequest` can be put in asynchronous mode by calling `request.startAsync()`. The main effect of doing so is that the Servlet (as well as any filters) can exit, but the response remains open to let processing complete later.
- The call to `request.startAsync()` returns `AsyncContext`, which you can use for further control over asynchronous processing. For example, it provides the `dispatch` method, which is similar to a forward from the Servlet API, except that it lets an application resume request processing on a Servlet container thread.
- The `ServletRequest` provides access to the current `DispatcherType`, which you can use to distinguish between processing the initial request, an asynchronous dispatch, a forward, and other dispatcher types.

`DeferredResult` processing works as follows:

- The controller returns a `DeferredResult` and saves it in some in-memory queue or list where it can be accessed.
- Spring MVC calls `request.startAsync()`.
- Meanwhile, the `DispatcherServlet` and all configured filters exit the request processing thread, but the response remains open.
- The application sets the `DeferredResult` from some thread, and Spring MVC dispatches the request back to the Servlet container.
- The `DispatcherServlet` is invoked again, and processing resumes with the asynchronously produced return value.

`Callable` processing works as follows:

- The controller returns a `Callable`.
- Spring MVC calls `request.startAsync()` and submits the `Callable` to a `TaskExecutor` for processing in a separate thread.
- Meanwhile, the `DispatcherServlet` and all filters exit the Servlet container thread, but the

response remains open.

- Eventually the `Callable` produces a result, and Spring MVC dispatches the request back to the Servlet container to complete processing.
- The `DispatcherServlet` is invoked again, and processing resumes with the asynchronously produced return value from the `Callable`.

For further background and context, you can also read the blog posts that introduced asynchronous request processing support in Spring MVC 3.2.

Exception Handling

When you use a `DeferredResult`, you can choose whether to call `setResult` or `setErrorResult` with an exception. In both cases, Spring MVC dispatches the request back to the Servlet container to complete processing. It is then treated either as if the controller method returned the given value or as if it produced the given exception. The exception then goes through the regular exception handling mechanism (for example, invoking `@ExceptionHandler` methods).

When you use `Callable`, similar processing logic occurs, the main difference being that the result is returned from the `Callable` or an exception is raised by it.

Interception

`HandlerInterceptor` instances can be of type `AsyncHandlerInterceptor`, to receive the `afterConcurrentHandlingStarted` callback on the initial request that starts asynchronous processing (instead of `postHandle` and `afterCompletion`).

`HandlerInterceptor` implementations can also register a `CallableProcessingInterceptor` or a `DeferredResultProcessingInterceptor`, to integrate more deeply with the lifecycle of an asynchronous request (for example, to handle a timeout event). See `AsyncHandlerInterceptor` for more details.

`DeferredResult` provides `onTimeout(Runnable)` and `onCompletion(Runnable)` callbacks. See the javadoc of `DeferredResult` for more details. `Callable` can be substituted for `WebAsyncTask` that exposes additional methods for timeout and completion callbacks.

Compared to WebFlux

The Servlet API was originally built for making a single pass through the Filter-Servlet chain. Asynchronous request processing lets applications exit the Filter-Servlet chain but leave the response open for further processing. The Spring MVC asynchronous support is built around that mechanism. When a controller returns a `DeferredResult`, the Filter-Servlet chain is exited, and the Servlet container thread is released. Later, when the `DeferredResult` is set, an `ASYNC` dispatch (to the same URL) is made, during which the controller is mapped again but, rather than invoking it, the `DeferredResult` value is used (as if the controller returned it) to resume processing.

By contrast, Spring WebFlux is neither built on the Servlet API, nor does it need such an asynchronous request processing feature, because it is asynchronous by design. Asynchronous handling is built into all framework contracts and is intrinsically supported through all stages of request processing.

From a programming model perspective, both Spring MVC and Spring WebFlux support asynchronous and Reactive Types as return values in controller methods. Spring MVC even supports streaming, including reactive back pressure. However, individual writes to the response remain blocking (and are performed on a separate thread), unlike WebFlux, which relies on non-blocking I/O and does not need an extra thread for each write.

Another fundamental difference is that Spring MVC does not support asynchronous or reactive types in controller method arguments (for example, `@RequestBody`, `@RequestPart`, and others), nor does it have any explicit support for asynchronous and reactive types as model attributes. Spring WebFlux does support all that.

HTTP Streaming

WebFlux

You can use `DeferredResult` and `Callable` for a single asynchronous return value. What if you want to produce multiple asynchronous values and have those written to the response? This section describes how to do so.

Objects

You can use the `ResponseBodyEmitter` return value to produce a stream of objects, where each object is serialized with an `HttpMessageConverter` and written to the response, as the following example shows:

Java

```
@GetMapping("/events")
public ResponseBodyEmitter handle() {
    ResponseBodyEmitter emitter = new ResponseBodyEmitter();
    // Save the emitter somewhere..
    return emitter;
}

// In some other thread
emitter.send("Hello once");

// and again later on
emitter.send("Hello again");

// and done at some point
emitter.complete();
```

Kotlin

```
@GetMapping("/events")
fun handle() = ResponseBodyEmitter().apply {
    // Save the emitter somewhere..
}

// In some other thread
emitter.send("Hello once")

// and again later on
emitter.send("Hello again")

// and done at some point
emitter.complete()
```

You can also use `ResponseBodyEmitter` as the body in a `ResponseEntity`, letting you customize the status and headers of the response.

When an `emitter` throws an `IOException` (for example, if the remote client went away), applications are not responsible for cleaning up the connection and should not invoke `emitter.complete` or `emitter.completeWithError`. Instead, the servlet container automatically initiates an `AsyncListener` error notification, in which Spring MVC makes a `completeWithError` call. This call, in turn, performs one final `ASYNC` dispatch to the application, during which Spring MVC invokes the configured exception resolvers and completes the request.

SSE

`SseEmitter` (a subclass of `ResponseBodyEmitter`) provides support for Server-Sent Events, where events sent from the server are formatted according to the W3C SSE specification. To produce an SSE stream from a controller, return `SseEmitter`, as the following example shows:

Java

```
@GetMapping(path="/events", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
public SseEmitter handle() {
    SseEmitter emitter = new SseEmitter();
    // Save the emitter somewhere..
    return emitter;
}

// In some other thread
emitter.send("Hello once");

// and again later on
emitter.send("Hello again");

// and done at some point
emitter.complete();
```

Kotlin

```
@GetMapping("/events", produces = [MediaType.TEXT_EVENT_STREAM_VALUE])
fun handle() = SseEmitter().apply {
    // Save the emitter somewhere..
}

// In some other thread
emitter.send("Hello once")

// and again later on
emitter.send("Hello again")

// and done at some point
emitter.complete()
```

While SSE is the main option for streaming into browsers, note that Internet Explorer does not support Server-Sent Events. Consider using Spring's WebSocket messaging with SockJS fallback transports (including SSE) that target a wide range of browsers.

See also previous section for notes on exception handling.

Raw Data

Sometimes, it is useful to bypass message conversion and stream directly to the response `OutputStream` (for example, for a file download). You can use the `StreamingResponseBody` return value type to do so, as the following example shows:

Java

```
@GetMapping("/download")
public StreamingResponseBody handle() {
    return new StreamingResponseBody() {
        @Override
        public void writeTo(OutputStream outputStream) throws IOException {
            // write...
        }
    };
}
```

Kotlin

```
@GetMapping("/download")
fun handle() = StreamingResponseBody {
    // write...
}
```

You can use `StreamingResponseBody` as the body in a `ResponseEntity` to customize the status and headers of the response.

Reactive Types

WebFlux

Spring MVC supports use of reactive client libraries in a controller (also read Reactive Libraries in the WebFlux section). This includes the `WebClient` from `spring-webflux` and others, such as Spring Data reactive data repositories. In such scenarios, it is convenient to be able to return reactive types from the controller method.

Reactive return values are handled as follows:

- A single-value promise is adapted to, similar to using `DeferredResult`. Examples include `Mono` (Reactor) or `Single` (RxJava).
- A multi-value stream with a streaming media type (such as `application/x-ndjson` or `text/event-stream`) is adapted to, similar to using `ResponseBodyEmitter` or `SseEmitter`. Examples include `Flux` (Reactor) or `Observable` (RxJava). Applications can also return `Flux<ServerSentEvent>` or `Observable<ServerSentEvent>`.
- A multi-value stream with any other media type (such as `application/json`) is adapted to, similar to using `DeferredResult<List<?>>`.

Spring MVC supports Reactor and RxJava through the `ReactiveAdapterRegistry` from `spring-core`, which lets it adapt from multiple reactive libraries.

For streaming to the response, reactive back pressure is supported, but writes to the response are still blocking and are run on a separate thread through the configured `TaskExecutor`, to avoid blocking the upstream source (such as a `Flux` returned from `WebClient`). By default, `SimpleAsyncTaskExecutor` is used for the blocking writes, but that is not suitable under load. If you plan to stream with a reactive type, you should use the MVC configuration to configure a task executor.

Context Propagation

It is common to propagate context via `java.lang.ThreadLocal`. This works transparently for handling on the same thread, but requires additional work for asynchronous handling across multiple threads. The Micrometer Context Propagation library simplifies context propagation across threads, and across context mechanisms such as `ThreadLocal` values, Reactor context, GraphQL Java context, and others.

If Micrometer Context Propagation is present on the classpath, when a controller method returns a reactive type such as `Flux` or `Mono`, all `ThreadLocal` values, for which there is a registered `io.micrometer.ThreadLocalAccessor`, are written to the Reactor `Context` as key-value pairs, using the key assigned by the `ThreadLocalAccessor`.

For other asynchronous handling scenarios, you can use the Context Propagation library directly. For example:

Java

```
// Capture ThreadLocal values from the main thread ...
ContextSnapshot snapshot = ContextSnapshot.captureAll();

// On a different thread: restore ThreadLocal values
try (ContextSnapshot.Scope scoped = snapshot.setThreadLocals()) {
    // ...
}
```

For more details, see the documentation of the Micrometer Context Propagation library.

Disconnects

WebFlux

The Servlet API does not provide any notification when a remote client goes away. Therefore, while streaming to the response, whether through SseEmitter or reactive types, it is important to send data periodically, since the write fails if the client has disconnected. The send could take the form of an empty (comment-only) SSE event or any other data that the other side would have to interpret as a heartbeat and ignore.

Alternatively, consider using web messaging solutions (such as STOMP over WebSocket or WebSocket with SockJS) that have a built-in heartbeat mechanism.

Configuration

Compared to WebFlux

The asynchronous request processing feature must be enabled at the Servlet container level. The MVC configuration also exposes several options for asynchronous requests.

Servlet Container

Filter and Servlet declarations have an `asyncSupported` flag that needs to be set to `true` to enable asynchronous request processing. In addition, Filter mappings should be declared to handle the `ASYNC` `jakarta.servlet.DispatchType`.

In Java configuration, when you use `AbstractAnnotationConfigDispatcherServletInitializer` to initialize the Servlet container, this is done automatically.

In `web.xml` configuration, you can add `<async-supported>true</async-supported>` to the `DispatcherServlet` and to `Filter` declarations and add `<dispatcher>ASYNC</dispatcher>` to filter mappings.

Spring MVC

The MVC configuration exposes the following options related to asynchronous request processing:

- Java configuration: Use the `configureAsyncSupport` callback on `WebMvcConfigurer`.
- XML namespace: Use the `<async-support>` element under `<mvc:annotation-driven>`.

You can configure the following:

- Default timeout value for async requests, which if not set, depends on the underlying Servlet container.
- `AsyncTaskExecutor` to use for blocking writes when streaming with Reactive Types and for executing `Callable` instances returned from controller methods. We highly recommended configuring this property if you stream with reactive types or have controller methods that return `Callable`, since by default, it is a `SimpleAsyncTaskExecutor`.
- `DeferredResultProcessingInterceptor` implementations and `CallableProcessingInterceptor` implementations.

Note that you can also set the default timeout value on a `DeferredResult`, a `ResponseBodyEmitter`, and an `SseEmitter`. For a `Callable`, you can use `WebAsyncTask` to provide a timeout value.

5.1.7. CORS

WebFlux

Spring MVC lets you handle CORS (Cross-Origin Resource Sharing). This section describes how to do so.

Introduction

WebFlux

For security reasons, browsers prohibit AJAX calls to resources outside the current origin. For example, you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials — for example withdrawing money from your account!

Cross-Origin Resource Sharing (CORS) is a W3C specification implemented by most browsers that lets you specify what kind of cross-domain requests are authorized, rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

Processing

WebFlux

The CORS specification distinguishes between preflight, simple, and actual requests. To learn how CORS works, you can read this article, among many others, or see the specification for more details.

Spring MVC `HandlerMapping` implementations provide built-in support for CORS. After successfully mapping a request to a handler, `HandlerMapping` implementations check the CORS configuration for the given request and handler and take further actions. Preflight requests are handled directly, while simple and actual CORS requests are intercepted, validated, and have required CORS response headers set.

In order to enable cross-origin requests (that is, the `Origin` header is present and differs from the host of the request), you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the

responses of simple and actual CORS requests and, consequently, browsers reject them.

Each `HandlerMapping` can be configured individually with URL pattern-based `CorsConfiguration` mappings. In most cases, applications use the MVC Java configuration or the XML namespace to declare such mappings, which results in a single global map being passed to all `HandlerMapping` instances.

You can combine global CORS configuration at the `HandlerMapping` level with more fine-grained, handler-level CORS configuration. For example, annotated controllers can use class- or method-level `@CrossOrigin` annotations (other handlers can implement `CorsConfigurationSource`).

The rules for combining global and local configuration are generally additive — for example, all global and all local origins. For those attributes where only a single value can be accepted, e.g. `allowCredentials` and `maxAge`, the local overrides the global value. See `CorsConfiguration#combine(CorsConfiguration)` for more details.

To learn more from the source or make advanced customizations, check the code behind:

- `CorsConfiguration`
- `CorsProcessor`, `DefaultCorsProcessor`
- `AbstractHandlerMapping`

`@CrossOrigin`

WebFlux

The `@CrossOrigin` annotation enables cross-origin requests on annotated controller methods, as the following example shows:

Java

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@RestController
@RequestMapping("/account")
class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

By default, `@CrossOrigin` allows:

- All origins.
- All headers.
- All HTTP methods to which the controller method is mapped.

`allowCredentials` is not enabled by default, since that establishes a trust level that exposes sensitive user-specific information (such as cookies and CSRF tokens) and should only be used where appropriate. When it is enabled either `allowOrigins` must be set to one or more specific domain (but not the special value `"*"`) or alternatively the `allowOriginPatterns` property may be used to match to a dynamic set of origins.

`maxAge` is set to 30 minutes.

`@CrossOrigin` is supported at the class level, too, and is inherited by all methods, as the following example shows:

Java

```
@CrossOrigin(origins = "https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@CrossOrigin(origins = ["https://domain2.com"], maxAge = 3600)
@RestController
@RequestMapping("/account")
class AccountController {

    @GetMapping("/{id}")
    fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    fun remove(@PathVariable id: Long) {
        // ...
    }
```

You can use `@CrossOrigin` at both the class level and the method level, as the following example shows:

Java

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("https://domain2.com")
    @GetMapping("/{id}")
    public Account retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public void remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@CrossOrigin(maxAge = 3600)
@RestController
@RequestMapping("/account")
class AccountController {

    @CrossOrigin("https://domain2.com")
    @GetMapping("/{id}")
    fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

Global Configuration

WebFlux

In addition to fine-grained, controller method level configuration, you probably want to define some global CORS configuration, too. You can set URL-based `CorsConfiguration` mappings individually on any `HandlerMapping`. Most applications, however, use the MVC Java configuration or the MVC XML namespace to do that.

By default, global configuration enables the following:

- All origins.

- All headers.
- `GET`, `HEAD`, and `POST` methods.

`allowCredentials` is not enabled by default, since that establishes a trust level that exposes sensitive user-specific information (such as cookies and CSRF tokens) and should only be used where appropriate. When it is enabled either `allowOrigins` must be set to one or more specific domain (but not the special value `"*"`) or alternatively the `allowOriginPatterns` property may be used to match to a dynamic set of origins.

`maxAge` is set to 30 minutes.

Java Configuration

WebFlux

To enable CORS in the MVC Java config, you can use the `CorsRegistry` callback, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addCorsMappings(registry: CorsRegistry) {

        registry.addMapping("/api/**")
                .allowedOrigins("https://domain2.com")
                .allowedMethods("PUT", "DELETE")
                .allowedHeaders("header1", "header2", "header3")
                .exposedHeaders("header1", "header2")
                .allowCredentials(true).maxAge(3600)

        // Add more mappings...
    }
}
```

XML Configuration

To enable CORS in the XML namespace, you can use the `<mvc:cors>` element, as the following example shows:

```
<mvc:cors>

    <mvc:mapping path="/api/**"
        allowed-origins="https://domain1.com, https://domain2.com"
        allowed-methods="GET, PUT"
        allowed-headers="header1, header2, header3"
        exposed-headers="header1, header2" allow-credentials="true"
        max-age="123" />

    <mvc:mapping path="/resources/**"
        allowed-origins="https://domain1.com" />

</mvc:cors>
```

CORS Filter

WebFlux

You can apply CORS support through the built-in `CorsFilter`.

If you try to use the `CorsFilter` with Spring Security, keep in mind that Spring Security has built-in support for CORS.

To configure the filter, pass a `CorsConfigurationSource` to its constructor, as the following example shows:

Java

```
CorsConfiguration config = new CorsConfiguration();

// Possibly...
// config.applyPermitDefaultValues()

config.setAllowCredentials(true);
config.addAllowedOrigin("https://domain1.com");
config.addAllowedHeader("*");
config.addAllowedMethod("*");

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", config);

CorsFilter filter = new CorsFilter(source);
```

Kotlin

```
val config = CorsConfiguration()

// Possibly...
// config.applyPermitDefaultValues()

config.allowCredentials = true
config.addAllowedOrigin("https://domain1.com")
config.addAllowedHeader("*")
config.addAllowedMethod("*")

val source = UrlBasedCorsConfigurationSource()
source.registerCorsConfiguration("/**", config)

val filter = CorsFilter(source)
```

5.1.8. Error Responses

WebFlux

A common requirement for REST services is to include details in the body of error responses. The Spring Framework supports the "Problem Details for HTTP APIs" specification, RFC 7807.

The following are the main abstractions for this support:

- `ProblemDetail` — representation for an RFC 7807 problem detail; a simple container for both standard fields defined in the spec, and for non-standard ones.
- `ErrorResponse` — contract to expose HTTP error response details including HTTP status, response headers, and a body in the format of RFC 7807; this allows exceptions to encapsulate and expose the details of how they map to an HTTP response. All Spring MVC exceptions implement this.
- `ErrorResponseException` — basic `ErrorResponse` implementation that others can use as a

convenient base class.

- `ResponseEntityExceptionHandler` — convenient base class for an @ControllerAdvice that handles all Spring MVC exceptions, and any `ErrorResponseException`, and renders an error response with a body.

Render

WebFlux

You can return `ProblemDetail` or `ErrorResponse` from any `@ExceptionHandler` or from any `@RequestMapping` method to render an RFC 7807 response. This is processed as follows:

- The `status` property of `ProblemDetail` determines the HTTP status.
- The `instance` property of `ProblemDetail` is set from the current URL path, if not already set.
- For content negotiation, the Jackson `HttpMessageConverter` prefers "application/problem+json" over "application/json" when rendering a `ProblemDetail`, and also falls back on it if no compatible media type is found.

To enable RFC 7807 responses for Spring WebFlux exceptions and for any `ErrorResponseException`, extend `ResponseEntityExceptionHandler` and declare it as an @ControllerAdvice in Spring configuration. The handler has an `@ExceptionHandler` method that handles any `ErrorResponse` exception, which includes all built-in web exceptions. You can add more exception handling methods, and use a protected method to map any exception to a `ProblemDetail`.

Non-Standard Fields

WebFlux

You can extend an RFC 7807 response with non-standard fields in one of two ways.

One, insert into the "properties" `Map` of `ProblemDetail`. When using the Jackson library, the Spring Framework registers `ProblemDetailJacksonMixin` that ensures this "properties" `Map` is unwrapped and rendered as top level JSON properties in the response, and likewise any unknown property during deserialization is inserted into this `Map`.

You can also extend `ProblemDetail` to add dedicated non-standard properties. The copy constructor in `ProblemDetail` allows a subclass to make it easy to be created from an existing `ProblemDetail`. This could be done centrally, e.g. from an `@ControllerAdvice` such as `ResponseEntityExceptionHandler` that re-creates the `ProblemDetail` of an exception into a subclass with the additional non-standard fields.

Internationalization

WebFlux

It is a common requirement to internationalize error response details, and good practice to customize the problem details for Spring MVC exceptions. This is supported as follows:

- Each `ErrorResponse` exposes a message code and arguments to resolve the "detail" field through a MessageSource. The actual message code value is parameterized with placeholders, e.g. `"HTTP method {0} not supported"` to be expanded from the arguments.

- Each `ErrorResponse` also exposes a message code to resolve the "title" field.
- `ResponseEntityExceptionHandler` uses the message code and arguments to resolve the "detail" and the "title" fields.

By default, the message code for the "detail" field is "problemDetail." + the fully qualified exception class name. Some exceptions may expose additional message codes in which case a suffix is added to the default message code. The table below lists message arguments and codes for Spring MVC exceptions:

Exception	Message Code	Message Code Arguments
`AsyncRequestTimeoutException`	(default)	
`ConversionNotSupportedException`	(default)	`{0}` property name, `{1}` property value
`HttpMediaTypeNotAcceptableException`	(default)	`{0}` list of supported media types
`HttpMediaTypeNotAcceptableException`	(default) + ".parseError"	
`HttpMediaTypeNotSupportedException`	(default)	`{0}` the media type that is not supported, `{1}` list of supported media types
`HttpMediaTypeNotSupportedException`	(default) + ".parseError"	
`HttpMessageNotReadableException`	(default)	
`HttpMessageNotWritableException`	(default)	
`HttpRequestMethodNotSupportedException`	(default)	`{0}` the current HTTP method, `{1}` the list of supported HTTP methods
`MethodArgumentNotValidException`	(default)	`{0}` the list of global errors, `{1}` the list of field errors. Message codes and arguments for each error within the `BindingResult` are also resolved via `MessageSource`.
`MissingRequestHeaderException`	(default)	`{0}` the header name
`MissingServletRequestParameterException`	(default)	`{0}` the request parameter name
`MissingMatrixVariableException`	(default)	`{0}` the matrix variable name
`MissingPathVariableException`	(default)	`{0}` the path variable name
`MissingRequestCookieException`	(default)	`{0}` the cookie name
`MissingServletRequestPartException`	(default)	`{0}` the part name

Exception	Message Code	Message Code Arguments
`NoHandlerFoundException`	(default)	
`TypeMismatchException`	(default)	`{0}` property name, `{1}` property value
`UnsatisfiedServletRequestParameterException`	(default)	`{0}` the list of parameter conditions

By default, the message code for the "title" field is "problemDetail.title." + the fully qualified exception class name.

Client Handling

WebFlux

A client application can catch `WebClientResponseException`, when using the `WebClient`, or `RestClientResponseException` when using the `RestTemplate`, and use their `getResponseBodyAs` methods to decode the error response body to any target type such as `ProblemDetail`, or a subclass of `ProblemDetail`.

5.1.9. Web Security

WebFlux

The Spring Security project provides support for protecting web applications from malicious exploits. See the Spring Security reference documentation, including:

- Spring MVC Security
- Spring MVC Test Support
- CSRF protection
- Security Response Headers

HDIV is another web security framework that integrates with Spring MVC.

5.1.10. HTTP Caching

WebFlux

HTTP caching can significantly improve the performance of a web application. HTTP caching revolves around the `Cache-Control` response header and, subsequently, conditional request headers (such as `Last-Modified` and `ETag`). `Cache-Control` advises private (for example, browser) and public (for example, proxy) caches on how to cache and re-use responses. An `ETag` header is used to make a conditional request that may result in a 304 (NOT_MODIFIED) without a body, if the content has not changed. `ETag` can be seen as a more sophisticated successor to the `Last-Modified` header.

This section describes the HTTP caching-related options that are available in Spring Web MVC.

`CacheControl`

WebFlux

`CacheControl` provides support for configuring settings related to the `Cache-Control` header and is accepted as an argument in a number of places:

- `WebContentInterceptor`
- `WebContentGenerator`
- Controllers
- Static Resources

While RFC 7234 describes all possible directives for the `Cache-Control` response header, the `CacheControl` type takes a use case-oriented approach that focuses on the common scenarios:

Java

```
// Cache for an hour - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10,
TimeUnit.DAYS).noTransform().cachePublic();
```

Kotlin

```
// Cache for an hour - "Cache-Control: max-age=3600"
val ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS)

// Prevent caching - "Cache-Control: no-store"
val ccNoStore = CacheControl.noStore()

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
val ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic()
```

`WebContentGenerator` also accepts a simpler `cachePeriod` property (defined in seconds) that works as follows:

- A `-1` value does not generate a `Cache-Control` response header.
- A `0` value prevents caching by using the `'Cache-Control: no-store'` directive.
- An `n > 0` value caches the given response for `n` seconds by using the `'Cache-Control: max-age=n'`

directive.

Controllers

WebFlux

Controllers can add explicit support for HTTP caching. We recommended doing so, since the `lastModified` or `ETag` value for a resource needs to be calculated before it can be compared against conditional request headers. A controller can add an `ETag` header and `Cache-Control` settings to a `ResponseEntity`, as the following example shows:

Java

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book);
}
```

Kotlin

```
@GetMapping("/book/{id}")
fun showBook(@PathVariable id: Long): ResponseEntity<Book> {

    val book = findBook(id);
    val version = book.getVersion()

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book)
}
```

The preceding example sends a 304 (NOT_MODIFIED) response with an empty body if the comparison to the conditional request headers indicates that the content has not changed. Otherwise, the `ETag` and `Cache-Control` headers are added to the response.

You can also make the check against conditional request headers in the controller, as the following example shows:

Java

```
@RequestMapping
public String myHandleMethod(WebRequest request, Model model) {

    long eTag = ... ①

    if (request.checkNotModified(eTag)) {
        return null; ②
    }

    model.addAttribute(...); ③
    return "myViewName";
}
```

① Application-specific calculation.

② The response has been set to 304 (NOT_MODIFIED) — no further processing.

③ Continue with the request processing.

Kotlin

```
@RequestMapping
fun myHandleMethod(request: WebRequest, model: Model): String? {

    val eTag: Long = ... ①

    if (request.checkNotModified(eTag)) {
        return null ②
    }

    model[...] = ... ③
    return "myViewName"
}
```

① Application-specific calculation.

② The response has been set to 304 (NOT_MODIFIED) — no further processing.

③ Continue with the request processing.

There are three variants for checking conditional requests against `eTag` values, `lastModified` values, or both. For conditional `GET` and `HEAD` requests, you can set the response to 304 (NOT_MODIFIED). For conditional `POST`, `PUT`, and `DELETE`, you can instead set the response to 412 (PRECONDITION_FAILED), to prevent concurrent modification.

Static Resources

WebFlux

You should serve static resources with a `Cache-Control` and conditional response headers for optimal performance. See the section on configuring Static Resources.

`ETag` Filter

You can use the `ShallowEtagHeaderFilter` to add "shallow" `eTag` values that are computed from the response content and, thus, save bandwidth but not CPU time. See Shallow ETag.

5.1.11. View Technologies

WebFlux

The use of view technologies in Spring MVC is pluggable. Whether you decide to use Thymeleaf, Groovy Markup Templates, JSPs, or other technologies is primarily a matter of a configuration change. This chapter covers view technologies integrated with Spring MVC. We assume you are already familiar with View Resolution.

The views of a Spring MVC application live within the internal trust boundaries of that application. Views have access to all the beans of your application context. As such, it is not recommended to use Spring MVC's template support in applications where the templates are editable by external sources, since this can have security implications.

Thymeleaf

WebFlux

Thymeleaf is a modern server-side Java template engine that emphasizes natural HTML templates that can be previewed in a browser by double-clicking, which is very helpful for independent work on UI templates (for example, by a designer) without the need for a running server. If you want to replace JSPs, Thymeleaf offers one of the most extensive sets of features to make such a transition easier. Thymeleaf is actively developed and maintained. For a more complete introduction, see the Thymeleaf project home page.

The Thymeleaf integration with Spring MVC is managed by the Thymeleaf project. The configuration involves a few bean declarations, such as `ServletContextTemplateResolver`, `SpringTemplateEngine`, and `ThymeleafViewResolver`. See Thymeleaf+Spring for more details.

FreeMarker

WebFlux

Apache FreeMarker is a template engine for generating any kind of text output from HTML to email and others. The Spring Framework has built-in integration for using Spring MVC with FreeMarker templates.

View Configuration

WebFlux

The following example shows how to configure FreeMarker as a view technology:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure FreeMarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/WEB-INF/freemarker");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()
    }

    // Configure FreeMarker...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("/WEB-INF/freemarker")
    }
}
```

The following example shows how to configure the same in XML:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:freemarker/>
</mvc:view-resolvers>

<!-- Configure FreeMarker... -->
<mvc:freemarker-configurer>
    <mvc:template-loader-path location="/WEB-INF/freemarker"/>
</mvc:freemarker-configurer>
```

Alternatively, you can also declare the `FreeMarkerConfigurer` bean for full control over all properties, as the following example shows:

```
<bean id="freemarkerConfig"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>
```

Your templates need to be stored in the directory specified by the `FreeMarkerConfigurer` shown in the preceding example. Given the preceding configuration, if your controller returns a view name of `welcome`, the resolver looks for the `/WEB-INF/freemarker/welcome.ftl` template.

FreeMarker Configuration

WebFlux

You can pass FreeMarker 'Settings' and 'SharedVariables' directly to the FreeMarker `Configuration` object (which is managed by Spring) by setting the appropriate bean properties on the `FreeMarkerConfigurer` bean. The `freemarkerSettings` property requires a `java.util.Properties` object, and the `freemarkerVariables` property requires a `java.util.Map`. The following example shows how to use a `FreeMarkerConfigurer`:

```
<bean id="freemarkerConfig"
class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
    <property name="freemarkerVariables">
        <map>
            <entry key="xml_escape" value-ref="fmXmlEscape"/>
        </map>
    </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

See the FreeMarker documentation for details of settings and variables as they apply to the `Configuration` object.

Form Handling

Spring provides a tag library for use in JSPs that contains, among others, a `<spring:bind/>` element. This element primarily lets forms display values from form-backing objects and show the results of failed validations from a `Validator` in the web or business tier. Spring also has support for the same functionality in FreeMarker, with additional convenience macros for generating form input elements themselves.

The Bind Macros

WebFlux

A standard set of macros are maintained within the `spring-webmvc.jar` file for FreeMarker, so they are always available to a suitably configured application.

Some of the macros defined in the Spring templating libraries are considered internal (private), but no such scoping exists in the macro definitions, making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to directly call from within your templates. If you wish to view the macro code directly, the file is called `spring.ftl` and is in the `org.springframework.web.servlet.view.freemarker` package.

Simple Binding

In your HTML forms based on FreeMarker templates that act as a form view for a Spring MVC controller, you can use code similar to the next example to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. The following example shows a `personForm` view:

```
<!-- FreeMarker macros have to be imported into a namespace.
    We strongly recommend sticking to 'spring'. -->
<#import "/spring.ftl" as spring/>
<html>
    ...
    <form action="" method="POST">
        Name:
        <@spring.bind "personForm.name"/>
        <input type="text"
            name="${spring.status.expression}"
            value="${spring.status.value?html}"/><br />
        <#list spring.status.errorMessages as error> <b>${error}</b> <br /> </#list>
        <br />
        ...
        <input type="submit" value="submit"/>
    </form>
    ...
</html>
```

`<@spring.bind>` requires a 'path' argument, which consists of the name of your command object (it is 'command', unless you changed it in your controller configuration) followed by a period and the name of the field on the command object to which you wish to bind. You can also use nested fields,

such as `command.address.street`. The `bind` macro assumes the default HTML escaping behavior specified by the `ServletContext` parameter `defaultHtmlEscape` in `web.xml`.

An alternative form of the macro called `<@spring.bindEscaped>` takes a second argument that explicitly specifies whether HTML escaping should be used in the status error messages or values. You can set it to `true` or `false` as required. Additional form handling macros simplify the use of HTML escaping, and you should use these macros wherever possible. They are explained in the next section.

Input Macros

Additional convenience macros for FreeMarker simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and you can mix and match them with simple HTML or direct calls to the Spring bind macros that we highlighted previously.

The following table of available macros shows the FreeMarker Template (FTL) definitions and the parameter list that each takes:

Table 26. Table of macro definitions

macro	FTL definition
`message` (output a string from a resource bundle based on the code parameter)	<@spring.message code/>
`messageText` (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)	<@spring.messageText code, text/>
`url` (prefix a relative URL with the application's context root)	<@spring.url relativeUrl/>
`formInput` (standard input field for gathering user input)	<@spring.formInput path, attributes, fieldType/>
`formHiddenInput` (hidden input field for submitting non-user input)	<@spring.formHiddenInput path, attributes/>
`formPasswordInput` (standard input field for gathering passwords. Note that no value is ever populated in fields of this type.)	<@spring.formPasswordInput path, attributes/>
`formTextarea` (large text field for gathering long, freeform text input)	<@spring.formTextarea path, attributes/>
`formSingleSelect` (drop down box of options that let a single required value be selected)	<@spring.formSingleSelect path, options, attributes/>
`formMultiSelect` (a list box of options that let the user select 0 or more values)	<@spring.formMultiSelect path, options, attributes/>

macro	FTL definition
formRadioButtons (a set of radio buttons that let a single selection be made from the available choices)	<@spring.formRadioButtons path, options separator, attributes/>
formCheckboxes (a set of checkboxes that let 0 or more values be selected)	<@spring.formCheckboxes path, options, separator, attributes/>
formCheckbox (a single checkbox)	<@spring.formCheckbox path, attributes/>
showErrors (simplify display of validation errors for the bound field)	<@spring.showErrors separator, classOrStyle/>

In FreeMarker templates, formHiddenInput and formPasswordInput are not actually required, as you can use the normal formInput macro, specifying hidden or password as the value for the fieldType parameter.

The parameters to any of the above macros have consistent meanings:

- path: The name of the field to bind to (ie "command.name")
- options: A Map of all the available values that can be selected from in the input field. The keys to the map represent the values that are POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the corresponding values posted back by the form. Usually, such a map is supplied as reference data by the controller. You can use any Map implementation, depending on required behavior. For strictly sorted maps, you can use a SortedMap (such as a TreeMap) with a suitable Comparator and, for arbitrary Maps that should return values in insertion order, use a LinkedHashMap or a LinkedMap from commons-collections.
- separator: Where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (such as
).
- attributes: An additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field, you may supply attributes (such as 'rows="5" cols="60"'), or you could pass style information such as 'style="border:1px solid silver"'.
- classOrStyle: For the showErrors macro, the name of the CSS class that the span element that wraps each error uses. If no information is supplied (or the value is empty), the errors are wrapped in tags.

The following sections outline examples of the macros.

Input Fields

The formInput macro takes the path parameter (command.name) and an additional attributes parameter (which is empty in the upcoming example). The macro, along with all other form generation macros, performs an implicit Spring bind on the path parameter. The binding remains

valid until a new bind occurs, so the `showErrors` macro does not need to pass the path parameter again — it operates on the field for which a binding was last created.

The `showErrors` macro takes a separator parameter (the characters that are used to separate multiple errors on a given field) and also accepts a second parameter — this time, a class name or style attribute. Note that FreeMarker can specify default values for the attributes parameter. The following example shows how to use the `formInput` and `showErrors` macros:

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

The next example shows the output of the form fragment, generating the name field and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring's Validation framework.

The generated HTML resembles the following example:

```
Name:
<input type="text" name="name" value="">
<br>
    <b>required</b>
<br>
<br>
```

The `formTextarea` macro works the same way as the `formInput` macro and accepts the same parameter list. Commonly, the second parameter (`attributes`) is used to pass style information or `rows` and `cols` attributes for the `textarea`.

Selection Fields

You can use four selection field macros to generate common UI value selection inputs in your HTML forms:

- `formSingleSelect`
- `formMultiSelect`
- `formRadioButtons`
- `formCheckboxes`

Each of the four macros accepts a `Map` of options that contains the value for the form field and the label that corresponds to that value. The value and the label can be the same.

The next example is for radio buttons in FTL. The form-backing object specifies a default value of 'London' for this field, so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'. The following listing shows the example:

```
...
Town:
<@spring.formRadioButtons "command.address.town", cityMap, ""/><br><br>
```

The preceding listing renders a line of radio buttons, one for each value in `cityMap`, and uses a separator of `""`. No additional attributes are supplied (the last parameter to the macro is missing). The `cityMap` uses the same `String` for each key-value pair in the map. The map's keys are what the form actually submits as `POST` request parameters. The map values are the labels that the user sees. In the preceding example, given a list of three well known cities and a default value in the form backing object, the HTML resembles the following:

```
Town:
<input type="radio" name="address.town" value="London">London</input>
<input type="radio" name="address.town" value="Paris" checked="checked">Paris</input>
<input type="radio" name="address.town" value="New York">New York</input>
```

If your application expects to handle cities by internal codes (for example), you can create the map of codes with suitable keys, as the following example shows:

Java

```
protected Map<String, ?> referenceData(HttpServletRequest request) throws Exception {
    Map<String, String> cityMap = new LinkedHashMap<>();
    cityMap.put("LDN", "London");
    cityMap.put("PRS", "Paris");
    cityMap.put("NYC", "New York");

    Map<String, Object> model = new HashMap<>();
    model.put("cityMap", cityMap);
    return model;
}
```

Kotlin

```
protected fun referenceData(request: HttpServletRequest): Map<String, *> {
    val cityMap = linkedMapOf(
            "LDN" to "London",
            "PRS" to "Paris",
            "NYC" to "New York"
    )
    return hashMapOf("cityMap" to cityMap)
}
```

The code now produces output where the radio values are the relevant codes, but the user still sees the more user-friendly city names, as follows:

```
Town:
<input type="radio" name="address.town" value="LDN">London</input>
<input type="radio" name="address.town" value="PRS" checked="checked">Paris</input>
<input type="radio" name="address.town" value="NYC">New York</input>
```

HTML Escaping

Default usage of the form macros described earlier results in HTML elements that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your `web.xml` file, as used by Spring's bind support. To make the elements be XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model, where they are visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of `true` for a model or context variable named `xhtmlCompliant`, as the following example shows:

```
<#-- for FreeMarker -->
<#assign xhtmlCompliant = true>
```

After processing this directive, any elements generated by the Spring macros are now XHTML compliant.

In similar fashion, you can specify HTML escaping per field, as the following example shows:

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name"/>

<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

Groovy Markup

The Groovy Markup Template Engine is primarily aimed at generating XML-like markup (XML, XHTML, HTML5, and others), but you can use it to generate any text-based content. The Spring Framework has a built-in integration for using Spring MVC with Groovy Markup.

The Groovy Markup Template engine requires Groovy 2.3.1+.

Configuration

The following example shows how to configure the Groovy Markup Template Engine:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.groovy();
    }

    // Configure the Groovy Markup Template Engine...

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();
        configurer.setResourceLoaderPath("/WEB-INF/");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.groovy()
    }

    // Configure the Groovy Markup Template Engine...

    @Bean
    fun groovyMarkupConfigurer() = GroovyMarkupConfigurer().apply {
        resourceLoaderPath = "/WEB-INF/"
    }
}
```

The following example shows how to configure the same in XML:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:groovy/>
</mvc:view-resolvers>

<!-- Configure the Groovy Markup Template Engine... -->
<mvc:groovy-configurer resource-loader-path="/WEB-INF/"/>
```

Example

Unlike traditional template engines, Groovy Markup relies on a DSL that uses a builder syntax. The following example shows a sample template for an HTML page:

```
yieldUnescaped '<!DOCTYPE html>'
html(lang:'en') {
    head {
        meta('http-equiv':'"Content-Type" content="text/html; charset=utf-8"')
        title('My page')
    }
    body {
        p('This is an example of HTML contents')
    }
}
```

Script Views

WebFlux

The Spring Framework has a built-in integration for using Spring MVC with any templating library that can run on top of the JSR-223 Java scripting engine. We have tested the following templating libraries on different script engines:

Scripting Library	Scripting Engine
Handlebars	Nashorn
Mustache	Nashorn
React	Nashorn
EJS	Nashorn
ERB	JRuby
String templates	Jython
Kotlin Script templating	Kotlin

The basic rule for integrating any other script engine is that it must implement the `ScriptEngine` and `Invocable` interfaces.

Requirements

WebFlux

You need to have the script engine on your classpath, the details of which vary by script engine:

- The Nashorn JavaScript engine is provided with Java 8+. Using the latest update release available is highly recommended.
- JRuby should be added as a dependency for Ruby support.
- Jython should be added as a dependency for Python support.
- `org.jetbrains.kotlin:kotlin-script-util` dependency and a `META-INF/services/javax.script.ScriptEngineFactory` file containing a `org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory` line should be added for Kotlin script support. See this example for more details.

You need to have the script templating library. One way to do that for JavaScript is through WebJars.

Script Templates

WebFlux

You can declare a `ScriptTemplateConfigurer` bean to specify the script engine to use, the script files to load, what function to call to render templates, and so on. The following example uses Mustache templates and the Nashorn JavaScript engine:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.scriptTemplate()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply {
        engineName = "nashorn"
        setScripts("mustache.js")
        renderObject = "Mustache"
        renderFunction = "render"
    }
}
```

The following example shows the same arrangement in XML:

```
<mvc:annotation-driven/>

<mvc:view-resolvers>
    <mvc:script-template/>
</mvc:view-resolvers>

<mvc:script-template-configurer engine-name="nashorn" render-object="Mustache" render-
function="render">
    <mvc:script location="mustache.js"/>
</mvc:script-template-configurer>
```

The controller would look no different for the Java and XML configurations, as the following example shows:

Java

```
@Controller
public class SampleController {

    @GetMapping("/sample")
    public String test(Model model) {
        model.addAttribute("title", "Sample title");
        model.addAttribute("body", "Sample body");
        return "template";
    }
}
```

Kotlin

```
@Controller
class SampleController {

    @GetMapping("/sample")
    fun test(model: Model): String {
        model["title"] = "Sample title"
        model["body"] = "Sample body"
        return "template"
    }
}
```

The following example shows the Mustache template:

```
<html>
    <head>
        <title>{{title}}</title>
    </head>
    <body>
        <p>{{body}}</p>
    </body>
</html>
```

The render function is called with the following parameters:

- `String template`: The template content
- `Map model`: The view model
- `RenderingContext renderingContext`: The `RenderingContext` that gives access to the application context, the locale, the template loader, and the URL (since 5.0)

`Mustache.render()` is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you can provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them and requires a polyfill to emulate some browser facilities that are not available in the server-side script engine.

The following example shows how to do so:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("polyfill.js", "handlebars.js", "render.js");
        configurer.setRenderFunction("render");
        configurer.setSharedEngine(false);
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.scriptTemplate()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply {
        engineName = "nashorn"
        setScripts("polyfill.js", "handlebars.js", "render.js")
        renderFunction = "render"
        isSharedEngine = false
    }
}
```

Setting the `sharedEngine` property to `false` is required when using non-thread-safe script engines with templating libraries not designed for concurrency, such as Handlebars or React running on Nashorn. In that case, Java SE 8 update 60 is required, due to this bug, but it is generally recommended to use a recent Java SE patch release in any case.

`polyfill.js` defines only the `window` object needed by Handlebars to run properly, as follows:

```
var window = {};
```

This basic `render.js` implementation compiles the template before using it. A production-ready implementation should also store any reused cached templates or pre-compiled templates. You can do so on the script side (and handle any customization you need — managing template engine configuration, for example). The following example shows how to do so:

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

Check out the Spring Framework unit tests, Java, and resources, for more configuration examples.

JSP and JSTL

The Spring Framework has a built-in integration for using Spring MVC with JSP and JSTL.

View Resolvers

When developing with JSPs, you typically declare an `InternalResourceViewResolver` bean.

`InternalResourceViewResolver` can be used for dispatching to any Servlet resource but in particular for JSPs. As a best practice, we strongly encourage placing your JSP files in a directory under the `'WEB-INF'` directory so there can be no direct access by clients.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

JSPs versus JSTL

When using the JSP Standard Tag Library (JSTL) you must use a special view class, the `JstlView`, as JSTL needs some preparation before things such as the I18N features can work.

Spring's JSP Tag Library

Spring provides data binding of request parameters to command objects, as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags have HTML escaping features to enable or disable escaping of characters.

The `spring.tld` tag library descriptor (TLD) is included in the `spring-webmvc.jar`. For a comprehensive reference on individual tags, browse the API reference or see the tag library

description.

Spring's form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As we show in the following examples, the form tags make JSPs easier to develop, read, and maintain.

We go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

Configuration

The form tag library comes bundled in `spring-webmvc.jar`. The library descriptor is called `spring-form.tld`.

To use the tags from this library, add the following directive to the top of your JSP page:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

where `form` is the tag name prefix you want to use for the tags from this library.

The Form Tag

This tag renders an HTML 'form' element and exposes a binding path to inner tags for binding. It puts the command object in the `PageContext` so that the command object can be accessed by inner tags. All the other tags in this library are nested tags of the `form` tag.

Assume that we have a domain object called `User`. It is a JavaBean with properties such as `firstName` and `lastName`. We can use it as the form-backing object of our form controller, which returns `form.jsp`. The following example shows what `form.jsp` could look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller. Keep reading to see more complex examples of how inner tags are used with the `form` tag.

The following listing shows the generated HTML, which looks like a standard form:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value="Harry"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value="Potter"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

The preceding JSP assumes that the variable name of the form-backing object is `command`. If you have put the form-backing object into the model under another name (definitely a best practice), you can bind the form to the named variable, as the following example shows:

```
<form:form modelAttribute="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

The `input` Tag

This tag renders an HTML `input` element with the bound value and `type='text'` by default. For an example of this tag, see The Form Tag. You can also use HTML5-specific types, such as `email`, `tel`, `date`, and others.

The `checkbox` Tag

This tag renders an HTML `input` tag with the `type` set to `checkbox`.

Assume that our `User` has preferences such as newsletter subscription and a list of hobbies. The following example shows the `Preferences` class:

Java

```
public class Preferences {

    private boolean receiveNewsletter;
    private String[] interests;
    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

Kotlin

```
class Preferences(
        var receiveNewsletter: Boolean,
        var interests: StringArray,
        var favouriteWord: String
)
```

The corresponding `form.jsp` could then resemble the following:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
        </tr>

        <tr>
            <td>Interests:</td>
            <%-- Approach 2: Property is of an array or of type java.util.Collection
--%>
            <td>
                Quidditch: <form:checkbox path="preferences.interests"
value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests"
value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox
path="preferences.interests" value="Defence Against the Dark Arts"/>
            </td>
        </tr>

        <tr>
            <td>Favourite Word:</td>
            <%-- Approach 3: Property is of type java.lang.Object --%>
            <td>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
        </tr>
    </table>
</form:form>
```

There are three approaches to the `checkbox` tag, which should meet all your checkbox needs.

- Approach One: When the bound value is of type `java.lang.Boolean`, the `input(checkbox)` is marked as `checked` if the bound value is `true`. The `value` attribute corresponds to the resolved value of the `setValue(Object)` value property.
- Approach Two: When the bound value is of type `array` or `java.util.Collection`, the `input(checkbox)` is marked as `checked` if the configured `setValue(Object)` value is present in the bound `Collection`.
- Approach Three: For any other bound value type, the `input(checkbox)` is marked as `checked` if the configured `setValue(Object)` is equal to the bound value.

Note that, regardless of the approach, the same HTML structure is generated. The following HTML snippet defines some checkboxes:

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox"
value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox"
value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests"
type="checkbox" value="Defence Against the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
</tr>
```

You might not expect to see the additional hidden field after each checkbox. When a checkbox in an HTML page is not checked, its value is not sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML for Spring form data binding to work. The `checkbox` tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore (`_`) for each checkbox. By doing this, you are effectively telling Spring that "the checkbox was visible in the form, and I want my object to which the form data binds to reflect the state of the checkbox, no matter what."

The `checkboxes` Tag

This tag renders multiple HTML `input` tags with the `type` set to `checkbox`.

This section build on the example from the previous `checkbox` tag section. Sometimes, you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the `checkboxes` tag. You can pass in an `Array`, a `List`, or a `Map` that contains the available options in the `items` property. Typically, the bound property is a collection so that it can hold multiple values selected by the user. The following example shows a JSP that uses this tag:

```
<form:form>
    <table>
        <tr>
            <td>Interests:</td>
            <td>
                <%-- Property is of an array or of type java.util.Collection --%>
                <form:checkboxes path="preferences.interests"
items="${interestList}"/>
            </td>
        </tr>
    </table>
</form:form>
```

This example assumes that the `interestList` is a `List` available as a model attribute that contains

strings of the values to be selected from. If you use a `Map`, the map entry key is used as the value, and the map entry's value is used as the label to be displayed. You can also use a custom object where you can provide the property names for the value by using `itemValue` and the label by using `itemLabel`.

The `radiobutton` Tag

This tag renders an HTML `input` element with the `type` set to `radio`.

A typical usage pattern involves multiple tag instances bound to the same property but with different values, as the following example shows:

```
<tr>
    <td>Sex:</td>
    <td>
        Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/>
    </td>
</tr>
```

The `radiobuttons` Tag

This tag renders multiple HTML `input` elements with the `type` set to `radio`.

As with the `checkboxes` tag, you might want to pass in the available options as a runtime variable. For this usage, you can use the `radiobuttons` tag. You pass in an `Array`, a `List`, or a `Map` that contains the available options in the `items` property. If you use a `Map`, the map entry key is used as the value and the map entry's value are used as the label to be displayed. You can also use a custom object where you can provide the property names for the value by using `itemValue` and the label by using `itemLabel`, as the following example shows:

```
<tr>
    <td>Sex:</td>
    <td><form:radiobuttons path="sex" items="${sexOptions}"/></td>
</tr>
```

The `password` Tag

This tag renders an HTML `input` tag with the type set to `password` with the bound value.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password"/>
    </td>
</tr>
```

Note that, by default, the password value is not shown. If you do want the password value to be shown, you can set the value of the `showPassword` attribute to `true`, as the following example shows:

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showPassword="true"/>
    </td>
</tr>
```

The `select` Tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested `option` and `options` tags.

Assume that a `User` has a list of skills. The corresponding HTML could be as follows:

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
</tr>
```

If the `User`'s skill are in Herbology, the HTML source of the 'Skills' row could be as follows:

```
<tr>
    <td>Skills:</td>
    <td>
        <select name="skills" multiple="true">
            <option value="Potions">Potions</option>
            <option value="Herbology" selected="selected">Herbology</option>
            <option value="Quidditch">Quidditch</option>
        </select>
    </td>
</tr>
```

The `option` Tag

This tag renders an HTML `option` element. It sets `selected`, based on the bound value. The following HTML shows typical output for it:

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

If the `User`'s house was in Gryffindor, the HTML source of the 'House' row would be as follows:

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="selected">Gryffindor</option> ①
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
</tr>
```

① Note the addition of a `selected` attribute.

The `options` Tag

This tag renders a list of HTML `option` elements. It sets the `selected` attribute, based on the bound value. The following HTML shows typical output for it:

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
        </form:select>
    </td>
</tr>
```

If the `User` lived in the UK, the HTML source of the 'Country' row would be as follows:

```
<tr>
    <td>Country:</td>
    <td>
        <select name="country">
            <option value="-">--Please Select</option>
            <option value="AT">Austria</option>
            <option value="UK" selected="selected">United Kingdom</option> ①
            <option value="US">United States</option>
        </select>
    </td>
</tr>
```

① Note the addition of a `selected` attribute.

As the preceding example shows, the combined usage of an `option` tag with the `options` tag generates the same standard HTML but lets you explicitly specify a value in the JSP that is for display only (where it belongs), such as the default string in the example: "-- Please Select".

The `items` attribute is typically populated with a collection or array of item objects. `itemValue` and `itemLabel` refer to bean properties of those item objects, if specified. Otherwise, the item objects themselves are turned into strings. Alternatively, you can specify a `Map` of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If `itemValue` or `itemLabel` (or both) happen to be specified as well, the item value property applies to the map key, and the item label property applies to the map value.

The `textarea` Tag

This tag renders an HTML `textarea` element. The following HTML shows typical output for it:

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20"/></td>
    <td><form:errors path="notes"/></td>
</tr>
```

The `hidden` Tag

This tag renders an HTML `input` tag with the `type` set to `hidden` with the bound value. To submit an unbound hidden value, use the HTML `input` tag with the `type` set to `hidden`. The following HTML shows typical output for it:

```
<form:hidden path="house"/>
```

If we choose to submit the `house` value as a hidden one, the HTML would be as follows:

```
<input name="house" type="hidden" value="Gryffindor"/>
```

The `errors` Tag

This tag renders field errors in an HTML `span` element. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Assume that we want to display all error messages for the `firstName` and `lastName` fields once we submit the form. We have a validator for instances of the `User` class called `UserValidator`, as the following example shows:

Java

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required",
"Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required",
"Field is required.");
    }
}
```

Kotlin

```
class UserValidator : Validator {

    override fun supports(candidate: Class<*>): Boolean {
        return User::class.java.isAssignableFrom(candidate)
    }

    override fun validate(obj: Any, errors: Errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required",
"Field is required.")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required",
"Field is required.")
    }
}
```

The `form.jsp` could be as follows:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName"/></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

If we submit a form with empty values in the `firstName` and `lastName` fields, the HTML would be as follows:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

What if we want to display the entire list of errors for a given page? The next example shows that the `errors` tag also supports some basic wildcarding functionality.

- `path="*"`: Displays all errors.
- `path="lastName"`: Displays all errors associated with the `lastName` field.
- If `path` is omitted, only object errors are displayed.

The following example displays a list of errors at the top of the page, followed by field-specific errors next to the fields:

```
<form:form>
    <form:errors path="*" cssClass="errorBox"/>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName"/></td>
            <td><form:errors path="firstName"/></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName"/></td>
            <td><form:errors path="lastName"/></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form:form>
```

The HTML would be as follows:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is
required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes"/>
            </td>
        </tr>
    </table>
</form>
```

The `spring-form.tld` tag library descriptor (TLD) is included in the `spring-webmvc.jar`. For a comprehensive reference on individual tags, browse the API reference or see the tag library description.

HTTP Method Conversion

A key principle of REST is the use of the “Uniform Interface”. This means that all resources (URLs) can be manipulated by using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that it has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or you can do a POST with the “real” method as an additional parameter (modeled as a hidden input field in an HTML form). Spring’s `HiddenHttpMethodFilter` uses this latter trick. This filter is a plain Servlet filter and, therefore, it can be used in combination with any web framework (not just Spring MVC). Add this filter to your web.xml, and a POST with a hidden `method` parameter is converted into the corresponding HTTP method request.

To support HTTP method conversion, the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet comes from the Pet Clinic sample:

```
<form:form method="delete">
    <p class="submit"><input type="submit" value="Delete Pet"/></p>
</form:form>
```

The preceding example performs an HTTP POST, with the "real" DELETE method hidden behind a request parameter. It is picked up by the `HiddenHttpMethodFilter`, which is defined in web.xml, as the following example shows:

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

The following example shows the corresponding `@Controller` method:

Java

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

Kotlin

```
@RequestMapping(method = [RequestMethod.DELETE])
fun deletePet(@PathVariable ownerId: Int, @PathVariable petId: Int): String {
    clinic.deletePet(petId)
    return "redirect:/owners/$ownerId"
}
```

HTML5 Tags

The Spring form tag library allows entering dynamic attributes, which means you can enter any HTML5 specific attributes.

The form `input` tag supports entering a type attribute other than `text`. This is intended to allow rendering new HTML5 specific input types, such as `email`, `date`, `range`, and others. Note that entering `type='text'` is not required, since `text` is the default type.

Tiles

You can integrate Tiles - just as any other view technology - in web applications that use Spring. This section describes, in a broad way, how to do so.

This section focuses on Spring's support for Tiles version 3 in the `org.springframework.web.servlet.view.tiles3` package.

Dependencies

To be able to use Tiles, you have to add a dependency on Tiles version 3.0.1 or higher and its transitive dependencies to your project.

Configuration

To be able to use Tiles, you have to configure it by using files that contain definitions (for basic information on definitions and other Tiles concepts, see https://tiles.apache.org). In Spring, this is done by using the `TilesConfigurer`. The following example `ApplicationContext` configuration shows how to do so:

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>
</bean>
```

The preceding example defines five files that contain definitions. The files are all located in the `WEB-INF/defs` directory. At initialization of the `WebApplicationContext`, the files are loaded, and the definitions factory are initialized. After that has been done, the Tiles included in the definition files can be used as views within your Spring web application. To be able to use the views, you have to have a `ViewResolver` as with any other view technology in Spring: typically a convenient `TilesViewResolver`.

You can specify locale-specific Tiles definitions by adding an underscore and then the locale, as the following example shows:

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/tiles.xml</value>
            <value>/WEB-INF/defs/tiles_fr_FR.xml</value>
        </list>
    </property>
</bean>
```

With the preceding configuration, `tiles_fr_FR.xml` is used for requests with the `fr_FR` locale, and `tiles.xml` is used by default.

Since underscores are used to indicate locales, we recommended not using them otherwise in the file names for Tiles definitions.

`UrlBasedViewResolver`

The `UrlBasedViewResolver` instantiates the given `viewClass` for each view it has to resolve. The following bean defines a `UrlBasedViewResolver`:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="viewClass"
value="org.springframework.web.servlet.view.tiles3.TilesView"/>
</bean>
```

`SimpleSpringPreparerFactory` and `SpringBeanPreparerFactory`

As an advanced feature, Spring also supports two special Tiles `PreparerFactory` implementations. See the Tiles documentation for details on how to use `ViewPreparer` references in your Tiles definition files.

You can specify `SimpleSpringPreparerFactory` to autowire `ViewPreparer` instances based on specified preparer classes, applying Spring's container callbacks as well as applying configured Spring BeanPostProcessors. If Spring's context-wide annotation configuration has been activated, annotations in `ViewPreparer` classes are automatically detected and applied. Note that this expects preparer classes in the Tiles definition files, as the default `PreparerFactory` does.

You can specify `SpringBeanPreparerFactory` to operate on specified preparer names (instead of classes), obtaining the corresponding Spring bean from the DispatcherServlet's application context. The full bean creation process is in the control of the Spring application context in this case, allowing for the use of explicit dependency injection configuration, scoped beans, and so on. Note that you need to define one Spring bean definition for each preparer name (as used in your Tiles definitions). The following example shows how to define a `SpringBeanPreparerFactory` property on a `TilesConfigurer` bean:

```
<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>

    <!-- resolving preparer names as Spring bean definition names -->
    <property name="preparerFactoryClass"

value="org.springframework.web.servlet.view.tiles3.SpringBeanPreparerFactory"/>

</bean>
```

RSS and Atom

Both `AbstractAtomFeedView` and `AbstractRssFeedView` inherit from the `AbstractFeedView` base class and are used to provide Atom and RSS Feed views, respectively. They are based on ROME project and are located in the package `org.springframework.web.servlet.view.feed`.

`AbstractAtomFeedView` requires you to implement the `buildFeedEntries()` method and optionally override the `buildFeedMetadata()` method (the default implementation is empty). The following example shows how to do so:

Java

```
public class SampleContentAtomView extends AbstractAtomFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
            Feed feed, HttpServletRequest request) {
        // implementation omitted
    }

    @Override
    protected List<Entry> buildFeedEntries(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response) throws Exception
{
        // implementation omitted
    }
}
```

Kotlin

```
class SampleContentAtomView : AbstractAtomFeedView() {

    override fun buildFeedMetadata(model: Map<String, Any>,
            feed: Feed, request: HttpServletRequest) {
        // implementation omitted
    }

    override fun buildFeedEntries(model: Map<String, Any>,
            request: HttpServletRequest, response: HttpServletResponse): List<Entry> {
        // implementation omitted
    }
}
```

Similar requirements apply for implementing `AbstractRssFeedView`, as the following example shows:

Java

```
public class SampleContentRssView extends AbstractRssFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model,
            Channel feed, HttpServletRequest request) {
        // implementation omitted
    }

    @Override
    protected List<Item> buildFeedItems(Map<String, Object> model,
            HttpServletRequest request, HttpServletResponse response) throws Exception
{
        // implementation omitted
    }
}
```

Kotlin

```
class SampleContentRssView : AbstractRssFeedView() {

    override fun buildFeedMetadata(model: Map<String, Any>,
                                   feed: Channel, request: HttpServletRequest) {
        // implementation omitted
    }

    override fun buildFeedItems(model: Map<String, Any>,
            request: HttpServletRequest, response: HttpServletResponse): List<Item> {
        // implementation omitted
    }
}
```

The `buildFeedItems()` and `buildFeedEntries()` methods pass in the HTTP request, in case you need to access the Locale. The HTTP response is passed in only for the setting of cookies or other HTTP headers. The feed is automatically written to the response object after the method returns.

For an example of creating an Atom view, see Alef Arendsen's Spring Team Blog entry.

PDF and Excel

Spring offers ways to return output other than HTML, including PDF and Excel spreadsheets. This section describes how to use those features.

Introduction to Document Views

An HTML page is not always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and is streamed from the server with the correct content type, to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the Apache POI library to your classpath. For PDF generation, you need to add (preferably) the OpenPDF library.

You should use the latest versions of the underlying document-generation libraries, if possible. In particular, we strongly recommend OpenPDF (for example, OpenPDF 1.2.12) instead of the outdated original iText 2.1.7, since OpenPDF is actively maintained and fixes an important vulnerability for untrusted PDF content.

PDF Views

A simple PDF view for a word list could extend `org.springframework.web.servlet.view.document.AbstractPdfView` and implement the `buildPdfDocument()` method, as the following example shows:

Java

```
public class PdfWordList extends AbstractPdfView {

    protected void buildPdfDocument(Map<String, Object> model, Document doc, PdfWriter
writer,
            HttpServletRequest request, HttpServletResponse response) throws Exception
{

        List<String> words = (List<String>) model.get("wordList");
        for (String word : words) {
            doc.add(new Paragraph(word));
        }
    }
}
```

Kotlin

```
class PdfWordList : AbstractPdfView() {

    override fun buildPdfDocument(model: Map<String, Any>, doc: Document, writer:
PdfWriter,
            request: HttpServletRequest, response: HttpServletResponse) {

        val words = model["wordList"] as List<String>
        for (word in words) {
            doc.add(Paragraph(word))
        }
    }
}
```

A controller can return such a view either from an external view definition (referencing it by name) or as a `View` instance from the handler method.

Excel Views

Since Spring Framework 4.2, `org.springframework.web.servlet.view.document.AbstractXlsView` is provided as a base class for Excel views. It is based on Apache POI, with specialized subclasses (`AbstractXlsxView` and `AbstractXlsxStreamingView`) that supersede the outdated `AbstractExcelView` class.

The programming model is similar to `AbstractPdfView`, with `buildExcelDocument()` as the central template method and controllers being able to return such a view from an external definition (by name) or as a `View` instance from the handler method.

Jackson

WebFlux

Spring offers support for the Jackson JSON library.

Jackson-based JSON MVC Views

WebFlux

The `MappingJackson2JsonView` uses the Jackson library's `ObjectMapper` to render the response content as JSON. By default, the entire contents of the model map (with the exception of framework-specific classes) are encoded as JSON. For cases where the contents of the map need to be filtered, you can specify a specific set of model attributes to encode by using the `modelKeys` property. You can also use the `extractValueFromSingleKeyModel` property to have the value in single-key models extracted and serialized directly rather than as a map of model attributes.

You can customize JSON mapping as needed by using Jackson's provided annotations. When you need further control, you can inject a custom `ObjectMapper` through the `ObjectMapper` property, for cases where you need to provide custom JSON serializers and deserializers for specific types.

Jackson-based XML Views

WebFlux

`MappingJackson2XmlView` uses the Jackson XML extension's `XmlMapper` to render the response content as XML. If the model contains multiple entries, you should explicitly set the object to be serialized by using the `modelKey` bean property. If the model contains a single entry, it is serialized automatically.

You can customized XML mapping as needed by using JAXB or Jackson's provided annotations. When you need further control, you can inject a custom `XmlMapper` through the `ObjectMapper` property, for cases where custom XML you need to provide serializers and deserializers for specific types.

XML Marshalling

The `MarshallingView` uses an XML `Marshaller` (defined in the `org.springframework.oxm` package) to render the response content as XML. You can explicitly set the object to be marshalled by using a `MarshallingView` instance's `modelKey` bean property. Alternatively, the view iterates over all model properties and marshals the first type that is supported by the `Marshaller`. For more information on the functionality in the `org.springframework.oxm` package, see Marshalling XML using O/X Mappers.

XSLT Views

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring Web MVC application.

This example is a trivial Spring application that creates a list of words in the `Controller` and adds them to the model map. The map is returned, along with the view name of our XSLT view. See Annotated Controllers for details of Spring Web MVC's `Controller` interface. The XSLT controller turns the list of words into a simple XML document ready for transformation.

Beans

Configuration is standard for a simple Spring web application: The MVC configuration has to define an `XsltViewResolver` bean and regular MVC annotation configuration. The following example shows how to do so:

Java

```
@EnableWebMvc
@ComponentScan
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Bean
    public XsltViewResolver xsltViewResolver() {
        XsltViewResolver viewResolver = new XsltViewResolver();
        viewResolver.setPrefix("/WEB-INF/xsl/");
        viewResolver.setSuffix(".xslt");
        return viewResolver;
    }
}
```

Kotlin

```
@EnableWebMvc
@ComponentScan
@Configuration
class WebConfig : WebMvcConfigurer {

    @Bean
    fun xsltViewResolver() = XsltViewResolver().apply {
        setPrefix("/WEB-INF/xsl/")
        setSuffix(".xslt")
    }
}
```

Controller

We also need a Controller that encapsulates our word-generation logic.

The controller logic is encapsulated in a `@Controller` class, with the handler method being defined as follows:

Java

```
@Controller
public class XsltController {

    @RequestMapping("/")
    public String home(Model model) throws Exception {
        Document document =
DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element root = document.createElement("wordList");

        List<String> words = Arrays.asList("Hello", "Spring", "Framework");
        for (String word : words) {
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(word);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }

        model.addAttribute("wordList", root);
        return "home";
    }
}
```

Kotlin

```
import org.springframework.ui.set

@Controller
class XsltController {

    @RequestMapping("/")
    fun home(model: Model): String {
        val document =
DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument()
        val root = document.createElement("wordList")

        val words = listOf("Hello", "Spring", "Framework")
        for (word in words) {
            val wordNode = document.createElement("word")
            val textNode = document.createTextNode(word)
            wordNode.appendChild(textNode)
            root.appendChild(wordNode)
        }

        model["wordList"] = root
        return "home"
    }
}
```

So far, we have only created a DOM document and added it to the Model map. Note that you can also load an XML file as a `Resource` and use it instead of a custom DOM document.

There are software packages available that automatically 'domify' an object graph, but, within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data, which is a danger when using tools to manage the DOMification process.

Transformation

Finally, the `XsltViewResolver` resolves the “home” XSLT template file and merges the DOM document into it to generate our view. As shown in the `XsltViewResolver` configuration, XSLT templates live in the `war` file in the `WEB-INF/xsl` directory and end with an `xslt` file extension.

The following example shows an XSLT transform:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
                <h1>My First Words</h1>
                <ul>
                    <xsl:apply-templates/>
                </ul>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <li><xsl:value-of select="."/></li>
    </xsl:template>

</xsl:stylesheet>
```

The preceding transform is rendered as the following HTML:

```
<html>
    <head>
        <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Hello!</title>
    </head>
    <body>
        <h1>My First Words</h1>
        <ul>
            <li>Hello</li>
            <li>Spring</li>
            <li>Framework</li>
        </ul>
    </body>
</html>
```

5.1.12. MVC Config

WebFlux

The MVC Java configuration and the MVC XML namespace provide default configuration suitable for most applications and a configuration API to customize it.

For more advanced customizations, which are not available in the configuration API, see Advanced Java Config and Advanced XML Config.

You do not need to understand the underlying beans created by the MVC Java configuration and the MVC namespace. If you want to learn more, see Special Bean Types and Web MVC Config.

Enable MVC Configuration

WebFlux

In Java configuration, you can use the `@EnableWebMvc` annotation to enable MVC configuration, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig {
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig
```

In XML configuration, you can use the `<mvc:annotation-driven>` element to enable MVC

configuration, as the following example shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven/>

</beans>
```

The preceding example registers a number of Spring MVC infrastructure beans and adapts to dependencies available on the classpath (for example, payload converters for JSON, XML, and others).

MVC Config API

WebFlux

In Java configuration, you can implement the `WebMvcConfigurer` interface, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    // Implement configuration methods...
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    // Implement configuration methods...
}
```

In XML, you can check attributes and sub-elements of `<mvc:annotation-driven/>`. You can view the Spring MVC XML schema or use the code completion feature of your IDE to discover what attributes and sub-elements are available.

Type Conversion

WebFlux

By default, formatters for various number and date types are installed, along with support for customization via `@NumberFormat` and `@DateTimeFormat` on fields.

To register custom formatters and converters in Java config, use the following:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // ...
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addFormatters(registry: FormatterRegistry) {
        // ...
    }
}
```

To do the same in XML config, use the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven conversion-service="conversionService"/>

    <bean id="conversionService"

class="org.springframework.format.support.FormattingConversionServiceFactoryBean">
        <property name="converters">
            <set>
                <bean class="org.example.MyConverter"/>
            </set>
        </property>
        <property name="formatters">
            <set>
                <bean class="org.example.MyFormatter"/>
                <bean class="org.example.MyAnnotationFormatterFactory"/>
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.example.MyFormatterRegistrar"/>
            </set>
        </property>
    </bean>

</beans>
```

By default Spring MVC considers the request Locale when parsing and formatting date values. This works for forms where dates are represented as Strings with "input" form fields. For "date" and "time" form fields, however, browsers use a fixed format defined in the HTML spec. For such cases date and time formatting can be customized as follows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        DateTimeFormatterRegistrar registrar = new DateTimeFormatterRegistrar();
        registrar.setUseIsoFormat(true);
        registrar.registerFormatters(registry);
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addFormatters(registry: FormatterRegistry) {
        val registrar = DateTimeFormatterRegistrar()
        registrar.setUseIsoFormat(true)
        registrar.registerFormatters(registry)
    }
}
```

See the `FormatterRegistrar` SPI and the `FormattingConversionServiceFactoryBean` for more information on when to use FormatterRegistrar implementations.

Validation

WebFlux

By default, if Bean Validation is present on the classpath (for example, Hibernate Validator), the `LocalValidatorFactoryBean` is registered as a global Validator for use with `@Valid` and `Validated` on controller method arguments.

In Java configuration, you can customize the global `Validator` instance, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public Validator getValidator() {
        // ...
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun getValidator(): Validator {
        // ...
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        https://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven validator="globalValidator"/>

</beans>
```

Note that you can also register `Validator` implementations locally, as the following example shows:

Java

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }
}
```

Kotlin

```
@Controller
class MyController {

    @InitBinder
    protected fun initBinder(binder: WebDataBinder) {
        binder.addValidators(FooValidator())
    }
}
```

If you need to have a `LocalValidatorFactoryBean` injected somewhere, create a bean and mark it with `@Primary` in order to avoid conflict with the one declared in the MVC configuration.

Interceptors

In Java configuration, you can register interceptors to apply to incoming requests, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleChangeInterceptor());
        registry.addInterceptor(new
ThemeChangeInterceptor()).addPathPatterns("/**").excludePathPatterns("/admin/**");
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addInterceptors(registry: InterceptorRegistry) {
        registry.addInterceptor(LocaleChangeInterceptor())

registry.addInterceptor(ThemeChangeInterceptor()).addPathPatterns("/**").excludePathPa
tterns("/admin/**")
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <mvc:exclude-mapping path="/admin/**"/>
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

Mapped interceptors are not ideally suited as a security layer due to the potential for a mismatch with annotated controller path matching, which can also match trailing slashes and path extensions transparently, along with other path matching options. Many of these options have been deprecated but the potential for a mismatch remains. Generally, we recommend using Spring Security which includes a dedicated MvcRequestMatcher to align with Spring MVC path matching and also has a security firewall that blocks many unwanted characters in URL paths.

Content Types

WebFlux

You can configure how Spring MVC determines the requested media types from the request (for example, `Accept` header, URL path extension, query parameter, and others).

By default, only the `Accept` header is checked.

If you must use URL-based content type resolution, consider using the query parameter strategy over path extensions. See Suffix Match and Suffix Match and RFD for more details.

In Java configuration, you can customize requested content type resolution, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.mediaType("json", MediaType.APPLICATION_JSON);
        configurer.mediaType("xml", MediaType.APPLICATION_XML);
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureContentNegotiation(configurer: ContentNegotiationConfigurer)
{
        configurer.mediaType("json", MediaType.APPLICATION_JSON)
        configurer.mediaType("xml", MediaType.APPLICATION_XML)
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager"
class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">
    <property name="mediaTypes">
        <value>
            json=application/json
            xml=application/xml
        </value>
    </property>
</bean>
```

Message Converters

WebFlux

You can customize `HttpMessageConverter` in Java configuration by overriding `configureMessageConverters()` (to replace the default converters created by Spring MVC) or by overriding `extendMessageConverters()` (to customize the default converters or add additional converters to the default ones).

The following example adds XML and Jackson JSON converters with a customized `ObjectMapper` instead of the default ones:

Java

```
@Configuration
@EnableWebMvc
public class WebConfiguration implements WebMvcConfigurer {

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
        Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()
                .indentOutput(true)
                .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))
                .modulesToInstall(new ParameterNamesModule());
        converters.add(new MappingJackson2HttpMessageConverter(builder.build()));
        converters.add(new
MappingJackson2XmlHttpMessageConverter(builder.createXmlMapper(true).build()));
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfiguration : WebMvcConfigurer {

    override fun configureMessageConverters(converters:
MutableList<HttpMessageConverter<*>>) {
        val builder = Jackson2ObjectMapperBuilder()
                .indentOutput(true)
                .dateFormat(SimpleDateFormat("yyyy-MM-dd"))
                .modulesToInstall(ParameterNamesModule())
        converters.add(MappingJackson2HttpMessageConverter(builder.build()))

converters.add(MappingJackson2XmlHttpMessageConverter(builder.createXmlMapper(true).bu
ild()))
```

In the preceding example, `Jackson2ObjectMapperBuilder` is used to create a common configuration for both `MappingJackson2HttpMessageConverter` and `MappingJackson2XmlHttpMessageConverter` with indentation enabled, a customized date format, and the registration of `jackson-module-parameter-names`, Which adds support for accessing parameter names (a feature added in Java 8).

This builder customizes Jackson's default properties as follows:

- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

It also automatically registers the following well-known modules if they are detected on the

classpath:

- jackson-datatype-joda: Support for Joda-Time types.
- jackson-datatype-jsr310: Support for Java 8 Date and Time API types.
- jackson-datatype-jdk8: Support for other Java 8 types, such as `Optional`.
- `jackson-module-kotlin`: Support for Kotlin classes and data classes.

Enabling indentation with Jackson XML support requires `woodstox-core-asl` dependency in addition to `jackson-dataformat-xml` one.

Other interesting Jackson modules are available:

- jackson-datatype-money: Support for `javax.money` types (unofficial module).
- jackson-datatype-hibernate: Support for Hibernate-specific types and properties (including lazy-loading aspects).

The following example shows how to achieve the same configuration in XML:

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean
class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">
            <property name="objectMapper" ref="objectMapper"/>
        </bean>
        <bean
class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">
            <property name="objectMapper" ref="xmlMapper"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="objectMapper"
class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"
      p:indentOutput="true"
      p:simpleDateFormat="yyyy-MM-dd"

p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>
```

View Controllers

This is a shortcut for defining a `ParameterizableViewController` that immediately forwards to a view when invoked. You can use it in static cases when there is no Java controller logic to run before the view generates the response.

The following example of Java configuration forwards a request for `/` to a view called `home`:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addViewControllers(registry: ViewControllerRegistry) {
        registry.addViewController("/").setViewName("home")
    }
}
```

The following example achieves the same thing as the preceding example, but with XML, by using the `<mvc:view-controller>` element:

```
<mvc:view-controller path="/" view-name="home"/>
```

If an `@RequestMapping` method is mapped to a URL for any HTTP method then a view controller cannot be used to handle the same URL. This is because a match by URL to an annotated controller is considered a strong enough indication of endpoint ownership so that a 405 (METHOD_NOT_ALLOWED), a 415 (UNSUPPORTED_MEDIA_TYPE), or similar response can be sent to the client to help with debugging. For this reason it is recommended to avoid splitting URL handling across an annotated controller and a view controller.

View Resolvers

WebFlux

The MVC configuration simplifies the registration of view resolvers.

The following Java configuration example configures content negotiation view resolution by using JSP and Jackson as a default `View` for JSON rendering:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.jsp();
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.enableContentNegotiation(MappingJackson2JsonView())
        registry.jsp()
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean
class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:jsp/>
</mvc:view-resolvers>
```

Note, however, that FreeMarker, Tiles, Groovy Markup, and script templates also require configuration of the underlying view technology.

The MVC namespace provides dedicated elements. The following example works with FreeMarker:

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean
class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:freemarker cache="false"/>
</mvc:view-resolvers>

<mvc:freemarker-configurer>
    <mvc:template-loader-path location="/freemarker"/>
</mvc:freemarker-configurer>
```

In Java configuration, you can add the respective `Configurer` bean, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.freeMarker().cache(false);
    }

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/freemarker");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.enableContentNegotiation(MappingJackson2JsonView())
        registry.freeMarker().cache(false)
    }

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("/freemarker")
    }
}
```

Static Resources

WebFlux

This option provides a convenient way to serve static resources from a list of `Resource`-based locations.

In the next example, given a request that starts with `/resources`, the relative path is used to find and serve static resources relative to `/public` under the web application root or on the classpath under `/static`. The resources are served with a one-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The `Last-Modified` information is deduced from `Resource#lastModified` so that HTTP conditional requests are supported with `"Last-Modified"` headers.

The following listing shows how to do so with Java configuration:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public", "classpath:/static/")
                .setCacheControl(CacheControl.maxAge(Duration.ofDays(365)));
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addResourceHandlers(registry: ResourceHandlerRegistry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public", "classpath:/static/")
                .setCacheControl(CacheControl.maxAge(Duration.ofDays(365)))
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:resources mapping="/resources/**"
    location="/public, classpath:/static/"
    cache-period="31556926" />
```

See also HTTP caching support for static resources.

The resource handler also supports a chain of `ResourceResolver` implementations and `ResourceTransformer` implementations, which you can use to create a toolchain for working with optimized resources.

You can use the `VersionResourceResolver` for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other. A `ContentVersionStrategy` (MD5 hash) is a good choice — with some notable exceptions, such as JavaScript resources used with a module loader.

The following example shows how to use `VersionResourceResolver` in Java configuration:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)
                .addResolver(new
VersionResourceResolver().addContentVersionStrategy("/**"));
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun addResourceHandlers(registry: ResourceHandlerRegistry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)

.addResolver(VersionResourceResolver().addContentVersionStrategy("/**"))
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:resources mapping="/resources/**" location="/public/">
    <mvc:resource-chain resource-cache="true">
        <mvc:resolvers>
            <mvc:version-resolver>
                <mvc:content-version-strategy patterns="/**"/>
            </mvc:version-resolver>
        </mvc:resolvers>
    </mvc:resource-chain>
</mvc:resources>
```

You can then use `ResourceUrlProvider` to rewrite URLs and apply the full chain of resolvers and transformers — for example, to insert versions. The MVC configuration provides a `ResourceUrlProvider` bean so that it can be injected into others. You can also make the rewrite transparent with the `ResourceUrlEncodingFilter` for Thymeleaf, JSPs, FreeMarker, and others with URL tags that rely on `HttpServletResponse#encodeURL`.

Note that, when using both `EncodedResourceResolver` (for example, for serving gzipped or brotli-encoded resources) and `VersionResourceResolver`, you must register them in this order. That ensures content-based versions are always computed reliably, based on the unencoded file.

For WebJars, versioned URLs like `/webjars/jquery/1.2.0/jquery.min.js` are the recommended and most efficient way to use them. The related resource location is configured out of the box with Spring Boot (or can be configured manually via `ResourceHandlerRegistry`) and does not require to add the `org.webjars:webjars-locator-core` dependency.

Version-less URLs like `/webjars/jquery/jquery.min.js` are supported through the `WebJarsResourceResolver` which is automatically registered when the `org.webjars:webjars-locator-core` library is present on the classpath, at the cost of a classpath scanning that could slow down application startup. The resolver can re-write URLs to include the version of the jar and can also match against incoming URLs without versions — for example, from `/webjars/jquery/jquery.min.js` to `/webjars/jquery/1.2.0/jquery.min.js`.

The Java configuration based on `ResourceHandlerRegistry` provides further options for fine-grained control, e.g. last-modified behavior and optimized resource resolution.

Default Servlet

Spring MVC allows for mapping the `DispatcherServlet` to `/` (thus overriding the mapping of the container's default Servlet), while still allowing static resource requests to be handled by the container's default Servlet. It configures a `DefaultServletHttpRequestHandler` with a URL mapping of `/**` and the lowest priority relative to other URL mappings.

This handler forwards all requests to the default Servlet. Therefore, it must remain last in the order of all other URL `HandlerMappings`. That is the case if you use `<mvc:annotation-driven>`. Alternatively, if you set up your own customized `HandlerMapping` instance, be sure to set its `order` property to a value lower than that of the `DefaultServletHttpRequestHandler`, which is `Integer.MAX_VALUE`.

The following example shows how to enable the feature by using the default setup:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable();
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureDefaultServletHandling(configurer:
DefaultServletHandlerConfigurer) {
        configurer.enable()
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:default-servlet-handler/>
```

The caveat to overriding the `/` Servlet mapping is that the `RequestDispatcher` for the default Servlet

must be retrieved by name rather than by path. The `DefaultServletHttpRequestHandler` tries to auto-detect the default Servlet for the container at startup time, using a list of known names for most of the major Servlet containers (including Tomcat, Jetty, GlassFish, JBoss, Resin, WebLogic, and WebSphere). If the default Servlet has been custom-configured with a different name, or if a different Servlet container is being used where the default Servlet name is unknown, then you must explicitly provide the default Servlet's name, as the following example shows:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer
configurer) {
        configurer.enable("myCustomDefaultServlet");
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configureDefaultServletHandling(configurer:
DefaultServletHandlerConfigurer) {
        configurer.enable("myCustomDefaultServlet")
    }
}
```

The following example shows how to achieve the same configuration in XML:

```
<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>
```

Path Matching

WebFlux

You can customize options related to path matching and treatment of the URL. For details on the individual options, see the `PathMatchConfigurer` javadoc.

The following example shows how to customize path matching in Java configuration:

Java

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer.addPathPrefix("/api",
HandlerTypePredicate.forAnnotation(RestController.class));
    }

    private PathPatternParser patternParser() {
        // ...
    }
}
```

Kotlin

```
@Configuration
@EnableWebMvc
class WebConfig : WebMvcConfigurer {

    override fun configurePathMatch(configurer: PathMatchConfigurer) {
        configurer.addPathPrefix("/api",
HandlerTypePredicate.forAnnotation(RestController::class.java))
    }

    fun patternParser(): PathPatternParser {
        //...
    }
}
```

The following example shows how to customize path matching in XML configuration:

```
<mvc:annotation-driven>
    <mvc:path-matching
        path-helper="pathHelper"
        path-matcher="pathMatcher"/>
</mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>
<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>
```

Advanced Java Config

WebFlux

`@EnableWebMvc` imports `DelegatingWebMvcConfiguration`, which:

- Provides default Spring configuration for Spring MVC applications
- Detects and delegates to `WebMvcConfigurer` implementations to customize that configuration.

For advanced mode, you can remove `@EnableWebMvc` and extend directly from `DelegatingWebMvcConfiguration` instead of implementing `WebMvcConfigurer`, as the following example shows:

Java

```
@Configuration
public class WebConfig extends DelegatingWebMvcConfiguration {

    // ...
}
```

Kotlin

```
@Configuration
class WebConfig : DelegatingWebMvcConfiguration() {

    // ...
}
```

You can keep existing methods in `WebConfig`, but you can now also override bean declarations from the base class, and you can still have any number of other `WebMvcConfigurer` implementations on the classpath.

Advanced XML Config

The MVC namespace does not have an advanced mode. If you need to customize a property on a bean that you cannot change otherwise, you can use the `BeanPostProcessor` lifecycle hook of the Spring `ApplicationContext`, as the following example shows:

Java

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String name) throws
BeansException {
        // ...
    }
}
```

Kotlin

```
@Component
class MyPostProcessor : BeanPostProcessor {

    override fun postProcessBeforeInitialization(bean: Any, name: String): Any {
        // ...
    }
}
```

Note that you need to declare `MyPostProcessor` as a bean, either explicitly in XML or by letting it be detected through a `<component-scan/>` declaration.

5.1.13. HTTP/2

WebFlux

Servlet 4 containers are required to support HTTP/2, and Spring Framework 5 is compatible with Servlet API 4. From a programming model perspective, there is nothing specific that applications need to do. However, there are considerations related to server configuration. For more details, see the HTTP/2 wiki page.

The Servlet API does expose one construct related to HTTP/2. You can use the `jakarta.servlet.http.PushBuilder` to proactively push resources to clients, and it is supported as a method argument to `@RequestMapping` methods.

5.2. REST Clients

This section describes options for client-side access to REST endpoints.

5.2.1. `RestTemplate`

`RestTemplate` is a synchronous client to perform HTTP requests. It is the original Spring REST client and exposes a simple, template-method API over underlying HTTP client libraries.

As of 5.0 the `RestTemplate` is in maintenance mode, with only requests for minor changes and bugs to be accepted. Please, consider using the WebClient which offers a more modern API and supports sync, async, and streaming scenarios.

See REST Endpoints for details.

5.2.2. `WebClient`

`WebClient` is a non-blocking, reactive client to perform HTTP requests. It was introduced in 5.0 and offers a modern alternative to the `RestTemplate`, with efficient support for both synchronous and asynchronous, as well as streaming scenarios.

In contrast to `RestTemplate`, `WebClient` supports the following:

- Non-blocking I/O.
- Reactive Streams back pressure.
- High concurrency with fewer hardware resources.
- Functional-style, fluent API that takes advantage of Java 8 lambdas.
- Synchronous and asynchronous interactions.
- Streaming up to or streaming down from a server.

See WebClient for more details.

5.2.3. HTTP Interface

The Spring Frameworks lets you define an HTTP service as a Java interface with HTTP exchange methods. You can then generate a proxy that implements this interface and performs the exchanges. This helps to simplify HTTP remote access and provides additional flexibility for to choose an API style such as synchronous or reactive.

See REST Endpoints for details.

5.3. Testing

Same in Spring WebFlux

This section summarizes the options available in `spring-test` for Spring MVC applications.

- Servlet API Mocks: Mock implementations of Servlet API contracts for unit testing controllers, filters, and other web components. See Servlet API mock objects for more details.
- TestContext Framework: Support for loading Spring configuration in JUnit and TestNG tests, including efficient caching of the loaded configuration across test methods and support for loading a `WebApplicationContext` with a `MockServletContext`. See TestContext Framework for more details.
- Spring MVC Test: A framework, also known as `MockMvc`, for testing annotated controllers through the `DispatcherServlet` (that is, supporting annotations), complete with the Spring MVC infrastructure but without an HTTP server. See Spring MVC Test for more details.
- Client-side REST: `spring-test` provides a `MockRestServiceServer` that you can use as a mock server for testing client-side code that internally uses the `RestTemplate`. See Client REST Tests for more details.
- `WebTestClient`: Built for testing WebFlux applications, but it can also be used for end-to-end integration testing, to any server, over an HTTP connection. It is a non-blocking, reactive client and is well suited for testing asynchronous and streaming scenarios.

5.4. WebSockets

WebFlux

This part of the reference documentation covers support for Servlet stack, WebSocket messaging

that includes raw WebSocket interactions, WebSocket emulation through SockJS, and publish-subscribe messaging through STOMP as a sub-protocol over WebSocket.

= Introduction to WebSocket

The WebSocket protocol, RFC 6455, provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection. It is a different TCP protocol from HTTP but is designed to work over HTTP, using ports 80 and 443 and allowing re-use of existing firewall rules.

A WebSocket interaction begins with an HTTP request that uses the HTTP `Upgrade` header to upgrade or, in this case, to switch to the WebSocket protocol. The following example shows such an interaction:

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket ①
Connection: Upgrade ②
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

① The `Upgrade` header.

② Using the `Upgrade` connection.

Instead of the usual 200 status code, a server with WebSocket support returns output similar to the following:

```
HTTP/1.1 101 Switching Protocols ①
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

① Protocol switch

After a successful handshake, the TCP socket underlying the HTTP upgrade request remains open for both the client and the server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. See RFC 6455, the WebSocket chapter of HTML5, or any of the many introductions and tutorials on the Web.

Note that, if a WebSocket server is running behind a web server (e.g. nginx), you likely need to configure it to pass WebSocket upgrade requests on to the WebSocket server. Likewise, if the application runs in a cloud environment, check the instructions of the cloud provider related to WebSocket support.

== HTTP Versus WebSocket

Even though WebSocket is designed to be HTTP-compatible and starts with an HTTP request, it is important to understand that the two protocols lead to very different architectures and application programming models.

In HTTP and REST, an application is modeled as many URLs. To interact with the application, clients access those URLs, request-response style. Servers route requests to the appropriate handler based on the HTTP URL, method, and headers.

By contrast, in WebSockets, there is usually only one URL for the initial connect. Subsequently, all application messages flow on that same TCP connection. This points to an entirely different asynchronous, event-driven, messaging architecture.

WebSocket is also a low-level transport protocol, which, unlike HTTP, does not prescribe any semantics to the content of messages. That means that there is no way to route or process a message unless the client and the server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (for example, STOMP), through the `Sec-WebSocket-Protocol` header on the HTTP handshake request. In the absence of that, they need to come up with their own conventions.

== When to Use WebSockets

WebSockets can make a web page be dynamic and interactive. However, in many cases, a combination of Ajax and HTTP streaming or long polling can provide a simple and effective solution.

For example, news, mail, and social feeds need to update dynamically, but it may be perfectly okay to do so every few minutes. Collaboration, games, and financial apps, on the other hand, need to be much closer to real-time.

Latency alone is not a deciding factor. If the volume of messages is relatively low (for example, monitoring network failures) HTTP streaming or polling can provide an effective solution. It is the combination of low latency, high frequency, and high volume that make the best case for the use of WebSocket.

Keep in mind also that over the Internet, restrictive proxies that are outside of your control may preclude WebSocket interactions, either because they are not configured to pass on the `Upgrade` header or because they close long-lived connections that appear idle. This means that the use of WebSocket for internal applications within the firewall is a more straightforward decision than it is for public facing applications.

5.4.1. WebSocket API

WebFlux

The Spring Framework provides a WebSocket API that you can use to write client- and server-side applications that handle WebSocket messages.

`WebSocketHandler`

WebFlux

Creating a WebSocket server is as simple as implementing `WebSocketHandler` or, more likely, extending either `TextWebSocketHandler` or `BinaryWebSocketHandler`. The following example uses `TextWebSocketHandler`:

```
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {

    @Override
    public void handleTextMessage(WebSocketSession session, TextMessage message) {
        // ...
    }

}
```

There is dedicated WebSocket Java configuration and XML namespace support for mapping the preceding WebSocket handler to a specific URL, as the following example shows:

```
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

The preceding example is for use in Spring MVC applications and should be included in the configuration of a `DispatcherServlet`. However, Spring's WebSocket support does not depend on Spring MVC. It is relatively simple to integrate a `WebSocketHandler` into other HTTP-serving environments with the help of `WebSocketHttpRequestHandler`.

When using the `WebSocketHandler` API directly vs indirectly, e.g. through the STOMP messaging, the application must synchronize the sending of messages since the underlying standard WebSocket session (JSR-356) does not allow concurrent sending. One option is to wrap the `WebSocketSession` with `ConcurrentWebSocketSessionDecorator`.

WebSocket Handshake

WebFlux

The easiest way to customize the initial HTTP WebSocket handshake request is through a `HandshakeInterceptor`, which exposes methods for "before" and "after" the handshake. You can use such an interceptor to preclude the handshake or to make any attributes available to the `WebSocketSession`. The following example uses a built-in interceptor to pass HTTP session attributes to the WebSocket session:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(new MyHandler(), "/myHandler")
            .addInterceptors(new HttpSessionHandshakeInterceptor());
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
        <websocket:handshake-interceptors>
            <bean
class="org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor"/
>
        </websocket:handshake-interceptors>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

A more advanced option is to extend the `DefaultHandshakeHandler` that performs the steps of the WebSocket handshake, including validating the client origin, negotiating a sub-protocol, and other details. An application may also need to use this option if it needs to configure a custom `RequestUpgradeStrategy` in order to adapt to a WebSocket server engine and version that is not yet supported (see Deployment for more on this subject). Both the Java configuration and XML namespace make it possible to configure a custom `HandshakeHandler`.

Spring provides a `WebSocketHandlerDecorator` base class that you can use to decorate a `WebSocketHandler` with additional behavior. Logging and exception handling implementations are provided and added by default when using the WebSocket Java configuration or XML namespace. The `ExceptionWebSocketHandlerDecorator` catches all uncaught exceptions that arise from any `WebSocketHandler` method and closes the WebSocket session with status `1011`, which indicates a server error.

Deployment

The Spring WebSocket API is easy to integrate into a Spring MVC application where the `DispatcherServlet` serves both HTTP WebSocket handshake and other HTTP requests. It is also easy to integrate into other HTTP processing scenarios by invoking `WebSocketHttpRequestHandler`. This is convenient and easy to understand. However, special considerations apply with regards to JSR-356 runtimes.

The Java WebSocket API (JSR-356) provides two deployment mechanisms. The first involves a

Servlet container classpath scan (a Servlet 3 feature) at startup. The other is a registration API to use at Servlet container initialization. Neither of these mechanism makes it possible to use a single "front controller" for all HTTP processing — including WebSocket handshake and all other HTTP requests — such as Spring MVC's `DispatcherServlet`.

This is a significant limitation of JSR-356 that Spring's WebSocket support addresses with server-specific `RequestUpgradeStrategy` implementations even when running in a JSR-356 runtime. Such strategies currently exist for Tomcat, Jetty, GlassFish, WebLogic, WebSphere, and Undertow (and WildFly).

A request to overcome the preceding limitation in the Java WebSocket API has been created and can be followed at eclipse-ee4j/websocket-api#211. Tomcat, Undertow, and WebSphere provide their own API alternatives that make it possible to do this, and it is also possible with Jetty. We are hopeful that more servers will do the same.

A secondary consideration is that Servlet containers with JSR-356 support are expected to perform a `ServletContainerInitializer` (SCI) scan that can slow down application startup — in some cases, dramatically. If a significant impact is observed after an upgrade to a Servlet container version with JSR-356 support, it should be possible to selectively enable or disable web fragments (and SCI scanning) through the use of the `<absolute-ordering />` element in `web.xml`, as the following example shows:

```
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
    version="5.0">

    <absolute-ordering/>

</web-app>
```

You can then selectively enable web fragments by name, such as Spring's own `SpringServletContainerInitializer` that provides support for the Servlet 3 Java initialization API. The following example shows how to do so:

```
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        https://jakarta.ee/xml/ns/jakartaee
        https://jakarta.ee/xml/ns/jakartaee/web-app_5_0.xsd"
    version="5.0">

    <absolute-ordering>
        <name>spring_web</name>
    </absolute-ordering>

</web-app>
```

Server Configuration

WebFlux

Each underlying WebSocket engine exposes configuration properties that control runtime characteristics, such as the size of message buffer sizes, idle timeout, and others.

For Tomcat, WildFly, and GlassFish, you can add a `ServletServerContainerFactoryBean` to your WebSocket Java config, as the following example shows:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Bean
    public ServletServerContainerFactoryBean createWebSocketContainer() {
        ServletServerContainerFactoryBean container = new
ServletServerContainerFactoryBean();
        container.setMaxTextMessageBufferSize(8192);
        container.setMaxBinaryMessageBufferSize(8192);
        return container;
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <bean class="org.springframework...ServletServerContainerFactoryBean">
        <property name="maxTextMessageBufferSize" value="8192"/>
        <property name="maxBinaryMessageBufferSize" value="8192"/>
    </bean>

</beans>
```

For client-side WebSocket configuration, you should use `WebSocketContainerFactoryBean` (XML) or `ContainerProvider.getWebSocketContainer()` (Java configuration).

For Jetty, you need to supply a pre-configured Jetty `WebSocketServerFactory` and plug that into Spring's `DefaultHandshakeHandler` through your WebSocket Java config. The following example shows how to do so:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(echoWebSocketHandler(),
            "/echo").setHandshakeHandler(handshakeHandler());
    }

    @Bean
    public DefaultHandshakeHandler handshakeHandler() {

        WebSocketPolicy policy = new WebSocketPolicy(WebSocketBehavior.SERVER);
        policy.setInputBufferSize(8192);
        policy.setIdleTimeout(600000);

        return new DefaultHandshakeHandler(
                new JettyRequestUpgradeStrategy(new WebSocketServerFactory(policy)));
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/echo" handler="echoHandler"/>
        <websocket:handshake-handler ref="handshakeHandler"/>
    </websocket:handlers>

    <bean id="handshakeHandler" class="org.springframework...DefaultHandshakeHandler">
        <constructor-arg ref="upgradeStrategy"/>
    </bean>

    <bean id="upgradeStrategy"
class="org.springframework...JettyRequestUpgradeStrategy">
        <constructor-arg ref="serverFactory"/>
    </bean>

    <bean id="serverFactory" class="org.eclipse.jetty...WebSocketServerFactory">
        <constructor-arg>
            <bean class="org.eclipse.jetty...WebSocketPolicy">
                <constructor-arg value="SERVER"/>
                <property name="inputBufferSize" value="8092"/>
                <property name="idleTimeout" value="600000"/>
            </bean>
        </constructor-arg>
    </bean>

</beans>
```

Allowed Origins

WebFlux

As of Spring Framework 4.1.5, the default behavior for WebSocket and SockJS is to accept only same-origin requests. It is also possible to allow all or a specified list of origins. This check is mostly designed for browser clients. Nothing prevents other types of clients from modifying the `Origin` header value (see RFC 6454: The Web Origin Concept for more details).

The three possible behaviors are:

- Allow only same-origin requests (default): In this mode, when SockJS is enabled, the Iframe HTTP response header `X-Frame-Options` is set to `SAMEORIGIN`, and JSONP transport is disabled, since it does not allow checking the origin of a request. As a consequence, IE6 and IE7 are not supported when this mode is enabled.

- Allow a specified list of origins: Each allowed origin must start with `http://` or `https://`. In this mode, when SockJS is enabled, IFrame transport is disabled. As a consequence, IE6 through IE9 are not supported when this mode is enabled.
- Allow all origins: To enable this mode, you should provide `*` as the allowed origin value. In this mode, all transports are available.

You can configure WebSocket and SockJS allowed origins, as the following example shows:

```
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(),
"/myHandler").setAllowedOrigins("https://mydomain.com");
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers allowed-origins="https://mydomain.com">
        <websocket:mapping path="/myHandler" handler="myHandler" />
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

5.4.2. SockJS Fallback

Over the public Internet, restrictive proxies outside your control may preclude WebSocket interactions, either because they are not configured to pass on the `Upgrade` header or because they close long-lived connections that appear to be idle.

The solution to this problem is WebSocket emulation — that is, attempting to use WebSocket first and then falling back on HTTP-based techniques that emulate a WebSocket interaction and expose the same application-level API.

On the Servlet stack, the Spring Framework provides both server (and also client) support for the SockJS protocol.

Overview

The goal of SockJS is to let applications use a WebSocket API but fall back to non-WebSocket alternatives when necessary at runtime, without the need to change application code.

SockJS consists of:

- The SockJS protocol defined in the form of executable narrated tests.
- The SockJS JavaScript client — a client library for use in browsers.
- SockJS server implementations, including one in the Spring Framework `spring-websocket` module.
- A SockJS Java client in the `spring-websocket` module (since version 4.1).

SockJS is designed for use in browsers. It uses a variety of techniques to support a wide range of browser versions. For the full list of SockJS transport types and browsers, see the SockJS client page. Transports fall in three general categories: WebSocket, HTTP Streaming, and HTTP Long Polling. For an overview of these categories, see this blog post.

The SockJS client begins by sending `GET /info` to obtain basic information from the server. After that, it must decide what transport to use. If possible, WebSocket is used. If not, in most browsers, there is at least one HTTP streaming option. If not, then HTTP (long) polling is used.

All transport requests have the following URL structure:

```
https://host:port/myApp/myEndpoint/{server-id}/{session-id}/{transport}
```

where:

- `{server-id}` is useful for routing requests in a cluster but is not used otherwise.
- `{session-id}` correlates HTTP requests belonging to a SockJS session.
- `{transport}` indicates the transport type (for example, `websocket`, `xhr-streaming`, and others).

The WebSocket transport needs only a single HTTP request to do the WebSocket handshake. All messages thereafter are exchanged on that socket.

HTTP transports require more requests. Ajax/XHR streaming, for example, relies on one long-running request for server-to-client messages and additional HTTP POST requests for client-to-server messages. Long polling is similar, except that it ends the current request after each server-to-client send.

SockJS adds minimal message framing. For example, the server sends the letter `o` ("open" frame) initially, messages are sent as `a["message1","message2"]` (JSON-encoded array), the letter `h` ("heartbeat" frame) if no messages flow for 25 seconds (by default), and the letter `c` ("close" frame) to close the session.

To learn more, run an example in a browser and watch the HTTP requests. The SockJS client allows fixing the list of transports, so it is possible to see each transport one at a time. The SockJS client also provides a debug flag, which enables helpful messages in the browser console. On the server side, you can enable `TRACE` logging for `org.springframework.web.socket`. For even more detail, see the SockJS protocol narrated test.

Enabling SockJS

You can enable SockJS through Java configuration, as the following example shows:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(myHandler(), "/myHandler").withSockJS();
    }

    @Bean
    public WebSocketHandler myHandler() {
        return new MyHandler();
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:handlers>
        <websocket:mapping path="/myHandler" handler="myHandler"/>
        <websocket:sockjs/>
    </websocket:handlers>

    <bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>
```

The preceding example is for use in Spring MVC applications and should be included in the configuration of a `DispatcherServlet`. However, Spring's WebSocket and SockJS support does not depend on Spring MVC. It is relatively simple to integrate into other HTTP serving environments with the help of `SockJsHttpRequestHandler`.

On the browser side, applications can use the `sockjs-client` (version 1.0.x). It emulates the W3C WebSocket API and communicates with the server to select the best transport option, depending on the browser in which it runs. See the sockjs-client page and the list of transport types supported by browser. The client also provides several configuration options — for example, to specify which transports to include.

IE 8 and 9

Internet Explorer 8 and 9 remain in use. They are a key reason for having SockJS. This section covers important considerations about running in those browsers.

The SockJS client supports Ajax/XHR streaming in IE 8 and 9 by using Microsoft's `XDomainRequest`. That works across domains but does not support sending cookies. Cookies are often essential for Java applications. However, since the SockJS client can be used with many server types (not just Java ones), it needs to know whether cookies matter. If so, the SockJS client prefers Ajax/XHR for streaming. Otherwise, it relies on an iframe-based technique.

The first `/info` request from the SockJS client is a request for information that can influence the client's choice of transports. One of those details is whether the server application relies on cookies (for example, for authentication purposes or clustering with sticky sessions). Spring's SockJS support includes a property called `sessionCookieNeeded`. It is enabled by default, since most Java applications rely on the `JSESSIONID` cookie. If your application does not need it, you can turn off this option, and SockJS client should then choose `xdr-streaming` in IE 8 and 9.

If you do use an iframe-based transport, keep in mind that browsers can be instructed to block the use of IFrames on a given page by setting the HTTP response header `X-Frame-Options` to `DENY`,

SAMEORIGIN, or ALLOW-FROM <origin>. This is used to prevent clickjacking.

Spring Security 3.2+ provides support for setting X-Frame-Options on every response. By default, the Spring Security Java configuration sets it to DENY. In 3.2, the Spring Security XML namespace does not set that header by default but can be configured to do so. In the future, it may set it by default.

See Default Security Headers of the Spring Security documentation for details on how to configure the setting of the X-Frame-Options header. You can also see gh-2718 for additional background.

If your application adds the X-Frame-Options response header (as it should!) and relies on an iframe-based transport, you need to set the header value to SAMEORIGIN or ALLOW-FROM <origin>. The Spring SockJS support also needs to know the location of the SockJS client, because it is loaded from the iframe. By default, the iframe is set to download the SockJS client from a CDN location. It is a good idea to configure this option to use a URL from the same origin as the application.

The following example shows how to do so in Java configuration:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS()
                .setClientLibraryUrl("http://localhost:8080/myapp/js/sockjs-
client.js");
    }

    // ...

}
```

The XML namespace provides a similar option through the <websocket:sockjs> element.

During initial development, do enable the SockJS client devel mode that prevents the browser from caching SockJS requests (like the iframe) that would otherwise be cached. For details on how to enable it see the SockJS client page.

Heartbeats

The SockJS protocol requires servers to send heartbeat messages to preclude proxies from concluding that a connection is hung. The Spring SockJS configuration has a property called heartbeatTime that you can use to customize the frequency. By default, a heartbeat is sent after 25 seconds, assuming no other messages were sent on that connection. This 25-second value is in line with the following IETF recommendation for public Internet applications.

When using STOMP over WebSocket and SockJS, if the STOMP client and server negotiate heartbeats to be exchanged, the SockJS heartbeats are disabled.

The Spring SockJS support also lets you configure the `TaskScheduler` to schedule heartbeats tasks. The task scheduler is backed by a thread pool, with default settings based on the number of available processors. Your should consider customizing the settings according to your specific needs.

Client Disconnects

HTTP streaming and HTTP long polling SockJS transports require a connection to remain open longer than usual. For an overview of these techniques, see this blog post.

In Servlet containers, this is done through Servlet 3 asynchronous support that allows exiting the Servlet container thread, processing a request, and continuing to write to the response from another thread.

A specific issue is that the Servlet API does not provide notifications for a client that has gone away. See eclipse-ee4j/servlet-api#44. However, Servlet containers raise an exception on subsequent attempts to write to the response. Since Spring's SockJS Service supports server-sent heartbeats (every 25 seconds by default), that means a client disconnect is usually detected within that time period (or earlier, if messages are sent more frequently).

As a result, network I/O failures can occur because a client has disconnected, which can fill the log with unnecessary stack traces. Spring makes a best effort to identify such network failures that represent client disconnects (specific to each server) and log a minimal message by using the dedicated log category, `DISCONNECTED_CLIENT_LOG_CATEGORY` (defined in `AbstractSockJsSession`). If you need to see the stack traces, you can set that log category to TRACE.

SockJS and CORS

If you allow cross-origin requests (see Allowed Origins), the SockJS protocol uses CORS for cross-domain support in the XHR streaming and polling transports. Therefore, CORS headers are added automatically, unless the presence of CORS headers in the response is detected. So, if an application is already configured to provide CORS support (for example, through a Servlet Filter), Spring's `SockJsService` skips this part.

It is also possible to disable the addition of these CORS headers by setting the `suppressCors` property in Spring's SockJsService.

SockJS expects the following headers and values:

- `Access-Control-Allow-Origin`: Initialized from the value of the `Origin` request header.
- `Access-Control-Allow-Credentials`: Always set to `true`.
- `Access-Control-Request-Headers`: Initialized from values from the equivalent request header.
- `Access-Control-Allow-Methods`: The HTTP methods a transport supports (see `TransportType` enum).

- `Access-Control-Max-Age`: Set to 31536000 (1 year).

For the exact implementation, see `addCorsHeaders` in `AbstractSockJsService` and the `TransportType` enum in the source code.

Alternatively, if the CORS configuration allows it, consider excluding URLs with the SockJS endpoint prefix, thus letting Spring's `SockJsService` handle it.

`SockJsClient`

Spring provides a SockJS Java client to connect to remote SockJS endpoints without using a browser. This can be especially useful when there is a need for bidirectional communication between two servers over a public network (that is, where network proxies can preclude the use of the WebSocket protocol). A SockJS Java client is also very useful for testing purposes (for example, to simulate a large number of concurrent users).

The SockJS Java client supports the `websocket`, `xhr-streaming`, and `xhr-polling` transports. The remaining ones only make sense for use in a browser.

You can configure the `WebSocketTransport` with:

- `StandardWebSocketClient` in a JSR-356 runtime.
- `JettyWebSocketClient` by using the Jetty 9+ native WebSocket API.
- Any implementation of Spring's `WebSocketClient`.

An `XhrTransport`, by definition, supports both `xhr-streaming` and `xhr-polling`, since, from a client perspective, there is no difference other than in the URL used to connect to the server. At present there are two implementations:

- `RestTemplateXhrTransport` uses Spring's `RestTemplate` for HTTP requests.
- `JettyXhrTransport` uses Jetty's `HttpClient` for HTTP requests.

The following example shows how to create a SockJS client and connect to a SockJS endpoint:

```
List<Transport> transports = new ArrayList<>(2);
transports.add(new WebSocketTransport(new StandardWebSocketClient()));
transports.add(new RestTemplateXhrTransport());

SockJsClient sockJsClient = new SockJsClient(transports);
sockJsClient.doHandshake(new MyWebSocketHandler(), "ws://example.com:8080/sockjs");
```

SockJS uses JSON formatted arrays for messages. By default, Jackson 2 is used and needs to be on the classpath. Alternatively, you can configure a custom implementation of `SockJsMessageCodec` and configure it on the `SockJsClient`.

To use `SockJsClient` to simulate a large number of concurrent users, you need to configure the underlying HTTP client (for XHR transports) to allow a sufficient number of connections and threads. The following example shows how to do so with Jetty:

```
HttpClient jettyHttpClient = new HttpClient();
jettyHttpClient.setMaxConnectionsPerDestination(1000);
jettyHttpClient.setExecutor(new QueuedThreadPool(1000));
```

The following example shows the server-side SockJS-related properties (see javadoc for details) that you should also consider customizing:

```
@Configuration
public class WebSocketConfig extends WebSocketMessageBrokerConfigurationSupport {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/sockjs").withSockJS()
            .setStreamBytesLimit(512 * 1024) ①
            .setHttpMessageCacheSize(1000) ②
            .setDisconnectDelay(30 * 1000); ③
    }

    // ...
}
```

① Set the `streamBytesLimit` property to 512KB (the default is 128KB — `128 * 1024`).

② Set the `httpMessageCacheSize` property to 1,000 (the default is `100`).

③ Set the `disconnectDelay` property to 30 property seconds (the default is five seconds — `5 * 1000`).

5.4.3. STOMP

The WebSocket protocol defines two types of messages (text and binary), but their content is undefined. The protocol defines a mechanism for client and server to negotiate a sub-protocol (that is, a higher-level messaging protocol) to use on top of WebSocket to define what kind of messages each can send, what the format is, the content of each message, and so on. The use of a sub-protocol is optional but, either way, the client and the server need to agree on some protocol that defines message content.

Overview

STOMP (Simple Text Oriented Messaging Protocol) was originally created for scripting languages (such as Ruby, Python, and Perl) to connect to enterprise message brokers. It is designed to address a minimal subset of commonly used messaging patterns. STOMP can be used over any reliable two-way streaming network protocol, such as TCP and WebSocket. Although STOMP is a text-oriented protocol, message payloads can be either text or binary.

STOMP is a frame-based protocol whose frames are modeled on HTTP. The following listing shows the structure of a STOMP frame:

```
COMMAND
header1:value1
header2:value2

Body^@
```

Clients can use the `SEND` or `SUBSCRIBE` commands to send or subscribe for messages, along with a `destination` header that describes what the message is about and who should receive it. This enables a simple publish-subscribe mechanism that you can use to send messages through the broker to other connected clients or to send messages to the server to request that some work be performed.

When you use Spring's STOMP support, the Spring WebSocket application acts as the STOMP broker to clients. Messages are routed to `@Controller` message-handling methods or to a simple in-memory broker that keeps track of subscriptions and broadcasts messages to subscribed users. You can also configure Spring to work with a dedicated STOMP broker (such as RabbitMQ, ActiveMQ, and others) for the actual broadcasting of messages. In that case, Spring maintains TCP connections to the broker, relays messages to it, and passes messages from it down to connected WebSocket clients. Thus, Spring web applications can rely on unified HTTP-based security, common validation, and a familiar programming model for message handling.

The following example shows a client subscribing to receive stock quotes, which the server may emit periodically (for example, via a scheduled task that sends messages through a `SimpMessagingTemplate` to the broker):

```
SUBSCRIBE
id:sub-1
destination:/topic/price.stock.*

^@
```

The following example shows a client that sends a trade request, which the server can handle through an `@MessageMapping` method:

```
SEND
destination:/queue/trade
content-type:application/json
content-length:44

{"action":"BUY","ticker":"MMM","shares",44}^@
```

After the execution, the server can broadcast a trade confirmation message and details down to the client.

The meaning of a destination is intentionally left opaque in the STOMP spec. It can be any string, and it is entirely up to STOMP servers to define the semantics and the syntax of the destinations

that they support. It is very common, however, for destinations to be path-like strings where `/topic/..` implies publish-subscribe (one-to-many) and `/queue/` implies point-to-point (one-to-one) message exchanges.

STOMP servers can use the `MESSAGE` command to broadcast messages to all subscribers. The following example shows a server sending a stock quote to a subscribed client:

```
MESSAGE
message-id:nxahklf6-1
subscription:sub-1
destination:/topic/price.stock.MMM

{"ticker":"MMM","price":129.45}^@
```

A server cannot send unsolicited messages. All messages from a server must be in response to a specific client subscription, and the `subscription` header of the server message must match the `id` header of the client subscription.

The preceding overview is intended to provide the most basic understanding of the STOMP protocol. We recommended reviewing the protocol specification in full.

Benefits

Using STOMP as a sub-protocol lets the Spring Framework and Spring Security provide a richer programming model versus using raw WebSockets. The same point can be made about HTTP versus raw TCP and how it lets Spring MVC and other web frameworks provide rich functionality. The following is a list of benefits:

- No need to invent a custom messaging protocol and message format.
- STOMP clients, including a Java client in the Spring Framework, are available.
- You can (optionally) use message brokers (such as RabbitMQ, ActiveMQ, and others) to manage subscriptions and broadcast messages.
- Application logic can be organized in any number of `@Controller` instances and messages can be routed to them based on the STOMP destination header versus handling raw WebSocket messages with a single `WebSocketHandler` for a given connection.
- You can use Spring Security to secure messages based on STOMP destinations and message types.

Enable STOMP

STOMP over WebSocket support is available in the `spring-messaging` and `spring-websocket` modules. Once you have those dependencies, you can expose a STOMP endpoints, over WebSocket with SockJS Fallback, as the following example shows:

```
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS();  ①
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.setApplicationDestinationPrefixes("/app"); ②
        config.enableSimpleBroker("/topic", "/queue"); ③
    }
}
```

① `/portfolio` is the HTTP URL for the endpoint to which a WebSocket (or SockJS) client needs to connect for the WebSocket handshake.

② STOMP messages whose destination header begins with `/app` are routed to `@MessageMapping` methods in `@Controller` classes.

③ Use the built-in message broker for subscriptions and broadcasting and route messages whose destination header begins with `/topic` or `/queue` to the broker.

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker application-destination-prefix="/app">
        <websocket:stomp-endpoint path="/portfolio">
            <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:simple-broker prefix="/topic, /queue"/>
    </websocket:message-broker>

</beans>
```

For the built-in simple broker, the `/topic` and `/queue` prefixes do not have any special meaning. They are merely a convention to differentiate between pub-sub versus point-to-point messaging (that is, many subscribers versus one consumer). When you use an external broker, check the STOMP page of the broker to understand what kind of STOMP destinations and prefixes it supports.

To connect from a browser, for SockJS, you can use the `sockjs-client`. For STOMP, many applications have used the jmesnil/stomp-websocket library (also known as stomp.js), which is feature-complete and has been used in production for years but is no longer maintained. At present the JSteunou/webstomp-client is the most actively maintained and evolving successor of that library. The following example code is based on it:

```
var socket = new SockJS("/spring-websocket-portfolio/portfolio");
var stompClient = webstomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

Alternatively, if you connect through WebSocket (without SockJS), you can use the following code:

```
var socket = new WebSocket("/spring-websocket-portfolio/portfolio");
var stompClient = Stomp.over(socket);

stompClient.connect({}, function(frame) {
}
```

Note that `stompClient` in the preceding example does not need to specify `login` and `passcode` headers. Even if it did, they would be ignored (or, rather, overridden) on the server side. See Connecting to a Broker and Authentication for more information on authentication.

For more example code see:

- Using WebSocket to build an interactive web application — a getting started guide.
- Stock Portfolio — a sample application.

WebSocket Server

To configure the underlying WebSocket server, the information in Server Configuration applies. For Jetty, however you need to set the `HandshakeHandler` and `WebSocketPolicy` through the `StompEndpointRegistry`:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").setHandshakeHandler(handshakeHandler());
    }

    @Bean
    public DefaultHandshakeHandler handshakeHandler() {

        WebSocketPolicy policy = new WebSocketPolicy(WebSocketBehavior.SERVER);
        policy.setInputBufferSize(8192);
        policy.setIdleTimeout(600000);

        return new DefaultHandshakeHandler(
                new JettyRequestUpgradeStrategy(new WebSocketServerFactory(policy)));
    }
}
```

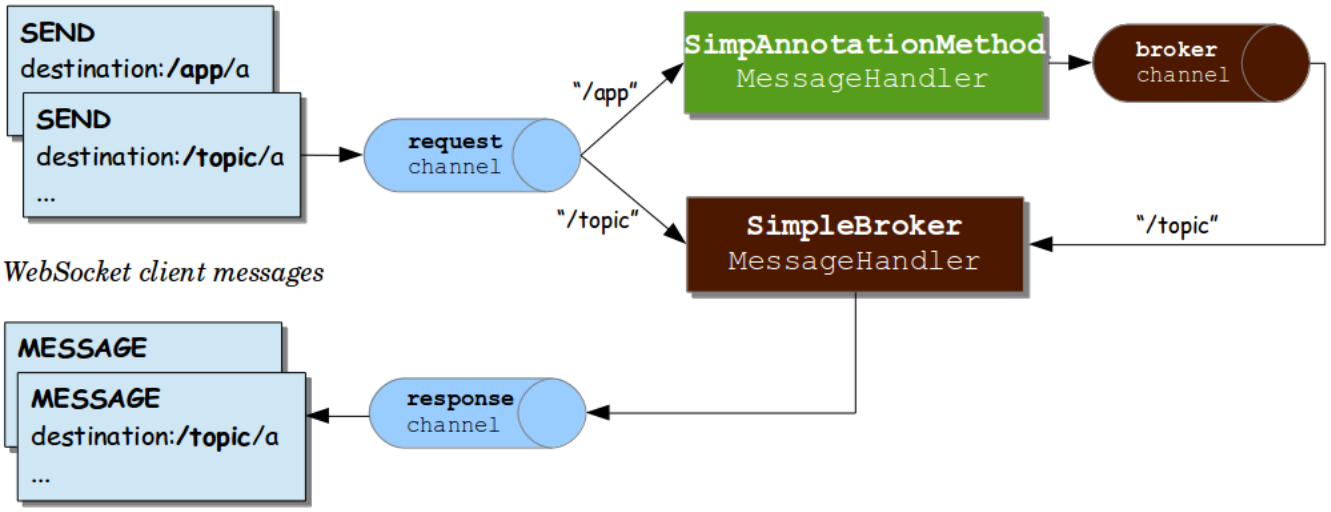
Flow of Messages

Once a STOMP endpoint is exposed, the Spring application becomes a STOMP broker for connected clients. This section describes the flow of messages on the server side.

The `spring-messaging` module contains foundational support for messaging applications that originated in Spring Integration and was later extracted and incorporated into the Spring Framework for broader use across many Spring projects and application scenarios. The following list briefly describes a few of the available messaging abstractions:

- Message: Simple representation for a message, including headers and payload.
- MessageHandler: Contract for handling a message.
- MessageChannel: Contract for sending a message that enables loose coupling between producers and consumers.
- SubscribableChannel: `MessageChannel` with `MessageHandler` subscribers.
- ExecutorSubscribableChannel: `SubscribableChannel` that uses an `Executor` for delivering messages.

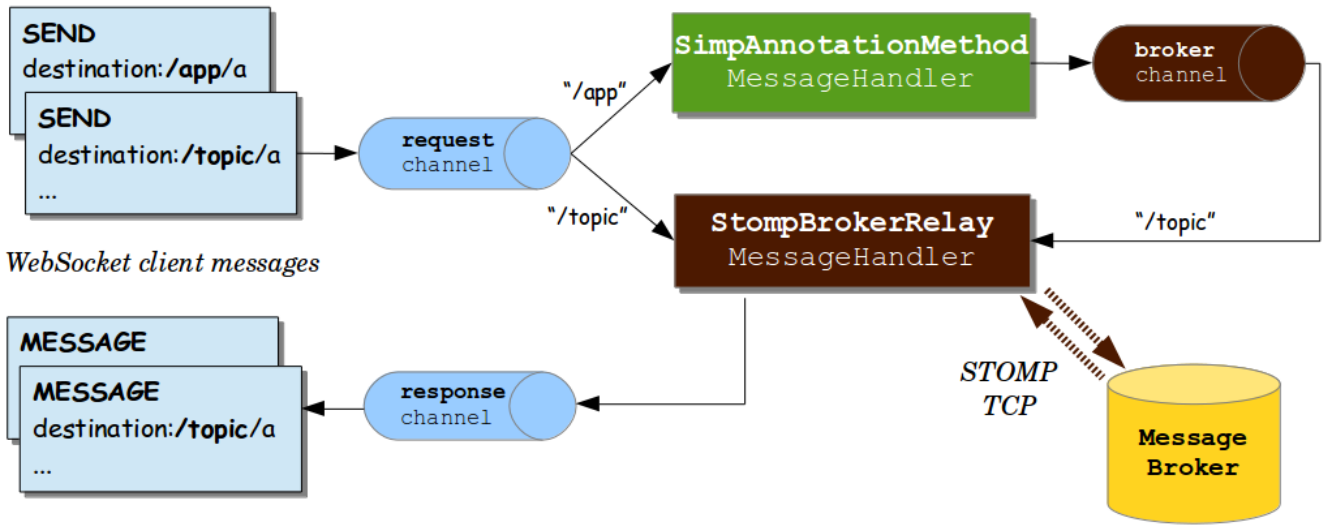
Both the Java configuration (that is, `@EnableWebSocketMessageBroker`) and the XML namespace configuration (that is, `<websocket:message-broker>`) use the preceding components to assemble a message workflow. The following diagram shows the components used when the simple built-in message broker is enabled:



The preceding diagram shows three message channels:

- `clientInboundChannel`: For passing messages received from WebSocket clients.
- `clientOutboundChannel`: For sending server messages to WebSocket clients.
- `brokerChannel`: For sending messages to the message broker from within server-side application code.

The next diagram shows the components used when an external broker (such as RabbitMQ) is configured for managing subscriptions and broadcasting messages:



The main difference between the two preceding diagrams is the use of the "broker relay" for passing messages up to the external STOMP broker over TCP and for passing messages down from the broker to subscribed clients.

When messages are received from a WebSocket connection, they are decoded to STOMP frames, turned into a Spring `Message` representation, and sent to the `clientInboundChannel` for further processing. For example, STOMP messages whose destination headers start with `/app` may be routed to `@MessageMapping` methods in annotated controllers, while `/topic` and `/queue` messages may be routed directly to the message broker.

An annotated `@Controller` that handles a STOMP message from a client may send a message to the

message broker through the `brokerChannel`, and the broker broadcasts the message to matching subscribers through the `clientOutboundChannel`. The same controller can also do the same in response to HTTP requests, so a client can perform an HTTP POST, and then a `@PostMapping` method can send a message to the message broker to broadcast to subscribed clients.

We can trace the flow through a simple example. Consider the following example, which sets up a server:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio");
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setApplicationDestinationPrefixes("/app");
        registry.enableSimpleBroker("/topic");
    }
}

@Controller
public class GreetingController {

    @MessageMapping("/greeting")
    public String handle(String greeting) {
        return "[" + getTimestamp() + ": " + greeting;
    }
}
```

The preceding example supports the following flow:

1. The client connects to `http://localhost:8080/portfolio` and, once a WebSocket connection is established, STOMP frames begin to flow on it.
2. The client sends a SUBSCRIBE frame with a destination header of `/topic/greeting`. Once received and decoded, the message is sent to the `clientInboundChannel` and is then routed to the message broker, which stores the client subscription.
3. The client sends a SEND frame to `/app/greeting`. The `/app` prefix helps to route it to annotated controllers. After the `/app` prefix is stripped, the remaining `/greeting` part of the destination is mapped to the `@MessageMapping` method in `GreetingController`.
4. The value returned from `GreetingController` is turned into a Spring `Message` with a payload based on the return value and a default destination header of `/topic/greeting` (derived from the input destination with `/app` replaced by `/topic`). The resulting message is sent to the `brokerChannel` and handled by the message broker.

5. The message broker finds all matching subscribers and sends a MESSAGE frame to each one through the `clientOutboundChannel`, from where messages are encoded as STOMP frames and sent on the WebSocket connection.

The next section provides more details on annotated methods, including the kinds of arguments and return values that are supported.

Annotated Controllers

Applications can use annotated `@Controller` classes to handle messages from clients. Such classes can declare `@MessageMapping`, `@SubscribeMapping`, and `@ExceptionHandler` methods, as described in the following topics:

- `@MessageMapping`
- `@SubscribeMapping`
- `@MessageExceptionHandler`

`@MessageMapping`

You can use `@MessageMapping` to annotate methods that route messages based on their destination. It is supported at the method level as well as at the type level. At the type level, `@MessageMapping` is used to express shared mappings across all methods in a controller.

By default, the mapping values are Ant-style path patterns (for example `/thing*`, `/thing/**`), including support for template variables (for example, `/thing/{id}`). The values can be referenced through `@DestinationVariable` method arguments. Applications can also switch to a dot-separated destination convention for mappings, as explained in Dots as Separators.

Supported Method Arguments

The following table describes the method arguments:

Method argument	Description
`Message`	For access to the complete message.
`MessageHeaders`	For access to the headers within the `Message`.
`MessageHeaderAccessor`, `SimpMessageHeaderAccessor`, and `StompHeaderAccessor`	For access to the headers through typed accessor methods.
`@Payload`	For access to the payload of the message, converted (for example, from JSON) by a configured `MessageConverter`. The presence of this annotation is not required since it is, by default, assumed if no other argument is matched. You can annotate payload arguments with `@jakarta.validation.Valid` or Spring's `@Validated`, to have the payload arguments be automatically validated.

Method argument	Description
`@Header`	For access to a specific header value — along with type conversion using an `org.springframework.core.convert.converter.Converter`, if necessary.
`@Headers`	For access to all headers in the message. This argument must be assignable to `java.util.Map`.
`@DestinationVariable`	For access to template variables extracted from the message destination. Values are converted to the declared method argument type as necessary.
`java.security.Principal`	Reflects the user logged in at the time of the WebSocket HTTP handshake.

Return Values

By default, the return value from a `@MessageMapping` method is serialized to a payload through a matching `MessageConverter` and sent as a `Message` to the `brokerChannel`, from where it is broadcast to subscribers. The destination of the outbound message is the same as that of the inbound message but prefixed with `/topic`.

You can use the `@SendTo` and `@SendToUser` annotations to customize the destination of the output message. `@SendTo` is used to customize the target destination or to specify multiple destinations. `@SendToUser` is used to direct the output message to only the user associated with the input message. See User Destinations.

You can use both `@SendTo` and `@SendToUser` at the same time on the same method, and both are supported at the class level, in which case they act as a default for methods in the class. However, keep in mind that any method-level `@SendTo` or `@SendToUser` annotations override any such annotations at the class level.

Messages can be handled asynchronously and a `@MessageMapping` method can return `ListenableFuture`, `CompletableFuture`, or `CompletionStage`.

Note that `@SendTo` and `@SendToUser` are merely a convenience that amounts to using the `SimpMessagingTemplate` to send messages. If necessary, for more advanced scenarios, `@MessageMapping` methods can fall back on using the `SimpMessagingTemplate` directly. This can be done instead of, or possibly in addition to, returning a value. See Sending Messages.

`@SubscribeMapping`

`@SubscribeMapping` is similar to `@MessageMapping` but narrows the mapping to subscription messages only. It supports the same method arguments as `@MessageMapping`. However for the return value, by default, a message is sent directly to the client (through `clientOutboundChannel`, in response to the subscription) and not to the broker (through `brokerChannel`, as a broadcast to matching subscriptions). Adding `@SendTo` or `@SendToUser` overrides this behavior and sends to the broker instead.

When is this useful? Assume that the broker is mapped to `/topic` and `/queue`, while application

controllers are mapped to `/app`. In this setup, the broker stores all subscriptions to `/topic` and `/queue` that are intended for repeated broadcasts, and there is no need for the application to get involved. A client could also subscribe to some `/app` destination, and a controller could return a value in response to that subscription without involving the broker without storing or using the subscription again (effectively a one-time request-reply exchange). One use case for this is populating a UI with initial data on startup.

When is this not useful? Do not try to map broker and controllers to the same destination prefix unless you want both to independently process messages, including subscriptions, for some reason. Inbound messages are handled in parallel. There are no guarantees whether a broker or a controller processes a given message first. If the goal is to be notified when a subscription is stored and ready for broadcasts, a client should ask for a receipt if the server supports it (simple broker does not). For example, with the Java STOMP client, you could do the following to add a receipt:

```
@Autowired
private TaskScheduler messageBrokerTaskScheduler;

// During initialization..
stompClient.setTaskScheduler(this.messageBrokerTaskScheduler);

// When subscribing..
StompHeaders headers = new StompHeaders();
headers.setDestination("/topic/...");
headers.setReceipt("r1");
FrameHandler handler = ...;
stompSession.subscribe(headers, handler).addReceiptTask(receiptHeaders -> {
    // Subscription ready...
});
```

A server side option is to register an `ExecutorChannelInterceptor` on the `brokerChannel` and implement the `afterMessageHandled` method that is invoked after messages, including subscriptions, have been handled.

`@MessageExceptionHandler`

An application can use `@MessageExceptionHandler` methods to handle exceptions from `@MessageMapping` methods. You can declare exceptions in the annotation itself or through a method argument if you want to get access to the exception instance. The following example declares an exception through a method argument:

```
@Controller
public class MyController {

    // ...

    @MessageExceptionHandler
    public ApplicationError handleException(MyException exception) {
        // ...
        return appError;
    }
}
```

`@MessageExceptionHandler` methods support flexible method signatures and support the same method argument types and return values as `@MessageMapping` methods.

Typically, `@MessageExceptionHandler` methods apply within the `@Controller` class (or class hierarchy) in which they are declared. If you want such methods to apply more globally (across controllers), you can declare them in a class marked with `@ControllerAdvice`. This is comparable to the similar support available in Spring MVC.

Sending Messages

What if you want to send messages to connected clients from any part of the application? Any application component can send messages to the `brokerChannel`. The easiest way to do so is to inject a `SimpMessagingTemplate` and use it to send messages. Typically, you would inject it by type, as the following example shows:

```
@Controller
public class GreetingController {

    private SimpMessagingTemplate template;

    @Autowired
    public GreetingController(SimpMessagingTemplate template) {
        this.template = template;
    }

    @RequestMapping(path="/greetings", method=POST)
    public void greet(String greeting) {
        String text = "[" + getTimestamp() + "]:" + greeting;
        this.template.convertAndSend("/topic/greetings", text);
    }

}
```

However, you can also qualify it by its name (`brokerMessagingTemplate`), if another bean of the same type exists.

Simple Broker

The built-in simple message broker handles subscription requests from clients, stores them in memory, and broadcasts messages to connected clients that have matching destinations. The broker supports path-like destinations, including subscriptions to Ant-style destination patterns.

Applications can also use dot-separated (rather than slash-separated) destinations. See Dots as Separators.

If configured with a task scheduler, the simple broker supports STOMP heartbeats. To configure a scheduler, you can declare your own `TaskScheduler` bean and set it through the `MessageBrokerRegistry`. Alternatively, you can use the one that is automatically declared in the built-in WebSocket configuration, however, you'll' need `@Lazy` to avoid a cycle between the built-in WebSocket configuration and your `WebSocketMessageBrokerConfigurer`. For example:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    private TaskScheduler messageBrokerTaskScheduler;

    @Autowired
    public void setMessageBrokerTaskScheduler(@Lazy TaskScheduler taskScheduler) {
        this.messageBrokerTaskScheduler = taskScheduler;
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableSimpleBroker("/queue/", "/topic/")
                .setHeartbeatValue(new long[] {10000, 20000})
                .setTaskScheduler(this.messageBrokerTaskScheduler);

        // ...
    }
}
```

External Broker

The simple broker is great for getting started but supports only a subset of STOMP commands (it does not support acks, receipts, and some other features), relies on a simple message-sending loop, and is not suitable for clustering. As an alternative, you can upgrade your applications to use a full-featured message broker.

See the STOMP documentation for your message broker of choice (such as RabbitMQ, ActiveMQ, and others), install the broker, and run it with STOMP support enabled. Then you can enable the STOMP broker relay (instead of the simple broker) in the Spring configuration.

The following example configuration enables a full-featured broker:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/portfolio").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableStompBrokerRelay("/topic", "/queue");
        registry.setApplicationDestinationPrefixes("/app");
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker application-destination-prefix="/app">
        <websocket:stomp-endpoint path="/portfolio" />
            <websocket:sockjs/>
        </websocket:stomp-endpoint>
        <websocket:stomp-broker-relay prefix="/topic,/queue" />
    </websocket:message-broker>

</beans>
```

The STOMP broker relay in the preceding configuration is a Spring `MessageHandler` that handles messages by forwarding them to an external message broker. To do so, it establishes TCP connections to the broker, forwards all messages to it, and then forwards all messages received from the broker to clients through their WebSocket sessions. Essentially, it acts as a "relay" that forwards messages in both directions.

Add `io.projectreactor.netty:reactor-netty` and `io.netty:netty-all` dependencies to your project for TCP connection management.

Furthermore, application components (such as HTTP request handling methods, business services, and others) can also send messages to the broker relay, as described in Sending Messages, to

broadcast messages to subscribed WebSocket clients.

In effect, the broker relay enables robust and scalable message broadcasting.

Connecting to a Broker

A STOMP broker relay maintains a single "system" TCP connection to the broker. This connection is used for messages originating from the server-side application only, not for receiving messages. You can configure the STOMP credentials (that is, the STOMP frame `login` and `passcode` headers) for this connection. This is exposed in both the XML namespace and Java configuration as the `systemLogin` and `systemPasscode` properties with default values of `guest` and `guest`.

The STOMP broker relay also creates a separate TCP connection for every connected WebSocket client. You can configure the STOMP credentials that are used for all TCP connections created on behalf of clients. This is exposed in both the XML namespace and Java configuration as the `clientLogin` and `clientPasscode` properties with default values of `guest` and `guest`.

The STOMP broker relay always sets the `login` and `passcode` headers on every `CONNECT` frame that it forwards to the broker on behalf of clients. Therefore, WebSocket clients need not set those headers. They are ignored. As the Authentication section explains, WebSocket clients should instead rely on HTTP authentication to protect the WebSocket endpoint and establish the client identity.

The STOMP broker relay also sends and receives heartbeats to and from the message broker over the "system" TCP connection. You can configure the intervals for sending and receiving heartbeats (10 seconds each by default). If connectivity to the broker is lost, the broker relay continues to try to reconnect, every 5 seconds, until it succeeds.

Any Spring bean can implement `ApplicationListener<BrokerAvailabilityEvent>` to receive notifications when the "system" connection to the broker is lost and re-established. For example, a Stock Quote service that broadcasts stock quotes can stop trying to send messages when there is no active "system" connection.

By default, the STOMP broker relay always connects, and reconnects as needed if connectivity is lost, to the same host and port. If you wish to supply multiple addresses, on each attempt to connect, you can configure a supplier of addresses, instead of a fixed host and port. The following example shows how to do that:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    // ...

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.enableStompBrokerRelay("/queue/",
"/topic/").setTcpClient(createTcpClient());
        registry.setApplicationDestinationPrefixes("/app");
    }

    private ReactorNettyTcpClient<byte[]> createTcpClient() {
        return new ReactorNettyTcpClient<>(
                client -> client.addressSupplier(() -> ... ),
                new StompReactorNettyCodec());
    }
}
```

You can also configure the STOMP broker relay with a `virtualHost` property. The value of this property is set as the `host` header of every `CONNECT` frame and can be useful (for example, in a cloud environment where the actual host to which the TCP connection is established differs from the host that provides the cloud-based STOMP service).

Dots as Separators

When messages are routed to `@MessageMapping` methods, they are matched with `AntPathMatcher`. By default, patterns are expected to use slash (`/`) as the separator. This is a good convention in web applications and similar to HTTP URLs. However, if you are more used to messaging conventions, you can switch to using dot (`.`) as the separator.

The following example shows how to do so in Java configuration:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    // ...

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        registry.setPathMatcher(new AntPathMatcher("."));
        registry.enableStompBrokerRelay("/queue", "/topic");
        registry.setApplicationDestinationPrefixes("/app");
    }
}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:websocket="http://www.springframework.org/schema/websocket"
        xsi:schemaLocation="
                http://www.springframework.org/schema/beans
                https://www.springframework.org/schema/beans/spring-beans.xsd
                http://www.springframework.org/schema/websocket
                https://www.springframework.org/schema/websocket/spring-
websocket.xsd">

    <websocket:message-broker application-destination-prefix="/app" path-
matcher="pathMatcher">
        <websocket:stomp-endpoint path="/stomp"/>
        <websocket:stomp-broker-relay prefix="/topic,/queue" />
    </websocket:message-broker>

    <bean id="pathMatcher" class="org.springframework.util.AntPathMatcher">
        <constructor-arg index="0" value="."/>
    </bean>

</beans>
```

After that, a controller can use a dot (`.`) as the separator in `@MessageMapping` methods, as the following example shows:

```
@Controller
@MessageMapping("red")
public class RedController {

    @MessageMapping("blue.{green}")
    public void handleGreen(@DestinationVariable String green) {
        // ...
    }
}
```

The client can now send a message to `/app/red.blue.green123`.

In the preceding example, we did not change the prefixes on the “broker relay”, because those depend entirely on the external message broker. See the STOMP documentation pages for the broker you use to see what conventions it supports for the destination header.

The “simple broker”, on the other hand, does rely on the configured `PathMatcher`, so, if you switch the separator, that change also applies to the broker and the way the broker matches destinations from a message to patterns in subscriptions.

Authentication

Every STOMP over WebSocket messaging session begins with an HTTP request. That can be a request to upgrade to WebSockets (that is, a WebSocket handshake) or, in the case of SockJS fallbacks, a series of SockJS HTTP transport requests.

Many web applications already have authentication and authorization in place to secure HTTP requests. Typically, a user is authenticated through Spring Security by using some mechanism such as a login page, HTTP basic authentication, or another way. The security context for the authenticated user is saved in the HTTP session and is associated with subsequent requests in the same cookie-based session.

Therefore, for a WebSocket handshake or for SockJS HTTP transport requests, typically, there is already an authenticated user accessible through `HttpServletRequest#getUserPrincipal()`. Spring automatically associates that user with a WebSocket or SockJS session created for them and, subsequently, with all STOMP messages transported over that session through a user header.

In short, a typical web application needs to do nothing beyond what it already does for security. The user is authenticated at the HTTP request level with a security context that is maintained through a cookie-based HTTP session (which is then associated with WebSocket or SockJS sessions created for that user) and results in a user header being stamped on every `Message` flowing through the application.

The STOMP protocol does have `login` and `passcode` headers on the `CONNECT` frame. Those were originally designed for and are needed for STOMP over TCP. However, for STOMP over WebSocket, by default, Spring ignores authentication headers at the STOMP protocol level, and assumes that the user is already authenticated at the HTTP transport level. The expectation is that the WebSocket or SockJS session contain the authenticated user.

Token Authentication

Spring Security OAuth provides support for token based security, including JSON Web Token (JWT). You can use this as the authentication mechanism in Web applications, including STOMP over WebSocket interactions, as described in the previous section (that is, to maintain identity through a cookie-based session).

At the same time, cookie-based sessions are not always the best fit (for example, in applications that do not maintain a server-side session or in mobile applications where it is common to use headers for authentication).

The WebSocket protocol, RFC 6455 "doesn't prescribe any particular way that servers can authenticate clients during the WebSocket handshake." In practice, however, browser clients can use only standard authentication headers (that is, basic HTTP authentication) or cookies and cannot (for example) provide custom headers. Likewise, the SockJS JavaScript client does not provide a way to send HTTP headers with SockJS transport requests. See sockjs-client issue 196. Instead, it does allow sending query parameters that you can use to send a token, but that has its own drawbacks (for example, the token may be inadvertently logged with the URL in server logs).

The preceding limitations are for browser-based clients and do not apply to the Spring Java-based STOMP client, which does support sending headers with both WebSocket and SockJS requests.

Therefore, applications that wish to avoid the use of cookies may not have any good alternatives for authentication at the HTTP protocol level. Instead of using cookies, they may prefer to authenticate with headers at the STOMP messaging protocol level. Doing so requires two simple steps:

1. Use the STOMP client to pass authentication headers at connect time.
2. Process the authentication headers with a `ChannelInterceptor`.

The next example uses server-side configuration to register a custom authentication interceptor. Note that an interceptor needs only to authenticate and set the user header on the CONNECT `Message`. Spring notes and saves the authenticated user and associate it with subsequent STOMP messages on the same session. The following example shows how register a custom authentication interceptor:

```
@Configuration
@EnableWebSocketMessageBroker
public class MyConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureClientInboundChannel(ChannelRegistration registration) {
        registration.interceptors(new ChannelInterceptor() {
            @Override
            public Message<?> preSend(Message<?> message, MessageChannel channel) {
                StompHeaderAccessor accessor =
                        MessageHeaderAccessor.getAccessor(message,
StompHeaderAccessor.class);
                if (StompCommand.CONNECT.equals(accessor.getCommand())) {
                    Authentication user = ... ; // access authentication header(s)
                    accessor.setUser(user);
                }
                return message;
            }
        });
    }
}
```

Also, note that, when you use Spring Security's authorization for messages, at present, you need to ensure that the authentication `ChannelInterceptor` config is ordered ahead of Spring Security's. This is best done by declaring the custom interceptor in its own implementation of `WebSocketMessageBrokerConfigurer` that is marked with `@Order(Ordered.HIGHEST_PRECEDENCE + 99)`.

Authorization

Spring Security provides WebSocket sub-protocol authorization that uses a `ChannelInterceptor` to authorize messages based on the user header in them. Also, Spring Session provides WebSocket

integration that ensures the user's HTTP session does not expire while the WebSocket session is still active.

User Destinations

An application can send messages that target a specific user, and Spring's STOMP support recognizes destinations prefixed with `/user/` for this purpose. For example, a client might subscribe to the `/user/queue/position-updates` destination. `UserDestinationMessageHandler` handles this destination and transforms it into a destination unique to the user session (such as `/queue/position-updates-user123`). This provides the convenience of subscribing to a generically named destination while, at the same time, ensuring no collisions with other users who subscribe to the same destination so that each user can receive unique stock position updates.

When working with user destinations, it is important to configure broker and application destination prefixes as shown in Enable STOMP, or otherwise the broker would handle "/user" prefixed messages that should only be handled by `UserDestinationMessageHandler`.

On the sending side, messages can be sent to a destination such as `/user/{username}/queue/position-updates`, which in turn is translated by the `UserDestinationMessageHandler` into one or more destinations, one for each session associated with the user. This lets any component within the application send messages that target a specific user without necessarily knowing anything more than their name and the generic destination. This is also supported through an annotation and a messaging template.

A message-handling method can send messages to the user associated with the message being handled through the `@SendToUser` annotation (also supported on the class-level to share a common destination), as the following example shows:

```
@Controller
public class PortfolioController {

    @MessageMapping("/trade")
    @SendToUser("/queue/position-updates")
    public TradeResult executeTrade(Trade trade, Principal principal) {
        // ...
        return tradeResult;
    }
}
```

If the user has more than one session, by default, all of the sessions subscribed to the given destination are targeted. However, sometimes, it may be necessary to target only the session that sent the message being handled. You can do so by setting the `broadcast` attribute to false, as the following example shows:

```
@Controller
public class MyController {

    @MessageMapping("/action")
    public void handleAction() throws Exception{
        // raise MyBusinessException here
    }

    @MessageExceptionHandler
    @SendToUser(destinations="/queue/errors", broadcast=false)
    public ApplicationError handleException(MyBusinessException exception) {
        // ...
        return appError;
    }
}
```

While user destinations generally imply an authenticated user, it is not strictly required. A WebSocket session that is not associated with an authenticated user can subscribe to a user destination. In such cases, the `@SendToUser` annotation behaves exactly the same as with `broadcast=false` (that is, targeting only the session that sent the message being handled).

You can send a message to user destinations from any application component by, for example, injecting the `SimpMessagingTemplate` created by the Java configuration or the XML namespace. (The bean name is `brokerMessagingTemplate` if required for qualification with `@Qualifier`.) The following example shows how to do so:

```
@Service
public class TradeServiceImpl implements TradeService {

    private final SimpMessagingTemplate messagingTemplate;

    @Autowired
    public TradeServiceImpl(SimpMessagingTemplate messagingTemplate) {
        this.messagingTemplate = messagingTemplate;
    }

    // ...

    public void afterTradeExecuted(Trade trade) {
        this.messagingTemplate.convertAndSendToUser(
                trade.getUserName(), "/queue/position-updates", trade.getResult());
    }
}
```

When you use user destinations with an external message broker, you should check the broker documentation on how to manage inactive queues, so that, when the user session is over, all unique user queues are removed. For example, RabbitMQ creates auto-delete queues when you use destinations such as `/exchange/amq.direct/position-updates`. So, in that case, the client could subscribe to `/user/exchange/amq.direct/position-updates`. Similarly, ActiveMQ has configuration options for purging inactive destinations.

In a multi-application server scenario, a user destination may remain unresolved because the user is connected to a different server. In such cases, you can configure a destination to broadcast unresolved messages so that other servers have a chance to try. This can be done through the `userDestinationBroadcast` property of the `MessageBrokerRegistry` in Java configuration and the `user-destination-broadcast` attribute of the `message-broker` element in XML.

Order of Messages

Messages from the broker are published to the `clientOutboundChannel`, from where they are written to WebSocket sessions. As the channel is backed by a `ThreadPoolExecutor`, messages are processed in different threads, and the resulting sequence received by the client may not match the exact order of publication.

If this is an issue, enable the `setPreservePublishOrder` flag, as the following example shows:

```
@Configuration
@EnableWebSocketMessageBroker
public class MyConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    protected void configureMessageBroker(MessageBrokerRegistry registry) {
        // ...
        registry.setPreservePublishOrder(true);
    }

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker preserve-publish-order="true">
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

When the flag is set, messages within the same client session are published to the `clientOutboundChannel` one at a time, so that the order of publication is guaranteed. Note that this incurs a small performance overhead, so you should enable it only if it is required.

Events

Several `ApplicationContext` events are published and can be received by implementing Spring's `ApplicationListener` interface:

- `BrokerAvailabilityEvent`: Indicates when the broker becomes available or unavailable. While the "simple" broker becomes available immediately on startup and remains so while the application is running, the STOMP "broker relay" can lose its connection to the full featured broker (for example, if the broker is restarted). The broker relay has reconnect logic and re-establishes the "system" connection to the broker when it comes back. As a result, this event is published whenever the state changes from connected to disconnected and vice-versa. Components that use the `SimpMessagingTemplate` should subscribe to this event and avoid sending messages at times when the broker is not available. In any case, they should be prepared to handle `MessageDeliveryException` when sending a message.
- `SessionConnectEvent`: Published when a new STOMP CONNECT is received to indicate the start of a new client session. The event contains the message that represents the connect, including the session ID, user information (if any), and any custom headers the client sent. This is useful for tracking client sessions. Components subscribed to this event can wrap the contained message with `SimpMessageHeaderAccessor` or `StompMessageHeaderAccessor`.
- `SessionConnectedEvent`: Published shortly after a `SessionConnectEvent` when the broker has sent a STOMP CONNECTED frame in response to the CONNECT. At this point, the STOMP session can be considered fully established.
- `SessionSubscribeEvent`: Published when a new STOMP SUBSCRIBE is received.
- `SessionUnsubscribeEvent`: Published when a new STOMP UNSUBSCRIBE is received.
- `SessionDisconnectEvent`: Published when a STOMP session ends. The DISCONNECT may have been sent from the client or it may be automatically generated when the WebSocket session is closed. In some cases, this event is published more than once per session. Components should

be idempotent with regard to multiple disconnect events.

When you use a full-featured broker, the STOMP "broker relay" automatically reconnects the "system" connection if broker becomes temporarily unavailable. Client connections, however, are not automatically reconnected. Assuming heartbeats are enabled, the client typically notices the broker is not responding within 10 seconds. Clients need to implement their own reconnecting logic.

Interception

Events provide notifications for the lifecycle of a STOMP connection but not for every client message. Applications can also register a `ChannelInterceptor` to intercept any message and in any part of the processing chain. The following example shows how to intercept inbound messages from clients:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureClientInboundChannel(ChannelRegistration registration) {
        registration.interceptors(new MyChannelInterceptor());
    }
}
```

A custom `ChannelInterceptor` can use `StompHeaderAccessor` or `SimpMessageHeaderAccessor` to access information about the message, as the following example shows:

```
public class MyChannelInterceptor implements ChannelInterceptor {

    @Override
    public Message<?> preSend(Message<?> message, MessageChannel channel) {
        StompHeaderAccessor accessor = StompHeaderAccessor.wrap(message);
        StompCommand command = accessor.getStompCommand();
        // ...
        return message;
    }
}
```

Applications can also implement `ExecutorChannelInterceptor`, which is a sub-interface of `ChannelInterceptor` with callbacks in the thread in which the messages are handled. While a `ChannelInterceptor` is invoked once for each message sent to a channel, the `ExecutorChannelInterceptor` provides hooks in the thread of each `MessageHandler` subscribed to messages from the channel.

Note that, as with the `SessionDisconnectEvent` described earlier, a DISCONNECT message can be from the client or it can also be automatically generated when the WebSocket session is closed. In

some cases, an interceptor may intercept this message more than once for each session. Components should be idempotent with regard to multiple disconnect events.

STOMP Client

Spring provides a STOMP over WebSocket client and a STOMP over TCP client.

To begin, you can create and configure `WebSocketStompClient`, as the following example shows:

```
WebSocketClient webSocketClient = new StandardWebSocketClient();
WebSocketStompClient stompClient = new WebSocketStompClient(webSocketClient);
stompClient.setMessageConverter(new StringMessageConverter());
stompClient.setTaskScheduler(taskScheduler); // for heartbeats
```

In the preceding example, you could replace `StandardWebSocketClient` with `SockJsClient`, since that is also an implementation of `WebSocketClient`. The `SockJsClient` can use WebSocket or HTTP-based transport as a fallback. For more details, see `SockJsClient`.

Next, you can establish a connection and provide a handler for the STOMP session, as the following example shows:

```
String url = "ws://127.0.0.1:8080/endpoint";
StompSessionHandler sessionHandler = new MyStompSessionHandler();
stompClient.connect(url, sessionHandler);
```

When the session is ready for use, the handler is notified, as the following example shows:

```
public class MyStompSessionHandler extends StompSessionHandlerAdapter {

    @Override
    public void afterConnected(StompSession session, StompHeaders connectedHeaders) {
        // ...
    }
}
```

Once the session is established, any payload can be sent and is serialized with the configured `MessageConverter`, as the following example shows:

```
session.send("/topic/something", "payload");
```

You can also subscribe to destinations. The `subscribe` methods require a handler for messages on the subscription and returns a `Subscription` handle that you can use to unsubscribe. For each received message, the handler can specify the target `Object` type to which the payload should be deserialized, as the following example shows:

```
session.subscribe("/topic/something", new StompFrameHandler() {

    @Override
    public Type getPayloadType(StompHeaders headers) {
        return String.class;
    }

    @Override
    public void handleFrame(StompHeaders headers, Object payload) {
        // ...
    }

});
```

To enable STOMP heartbeat, you can configure `WebSocketStompClient` with a `TaskScheduler` and optionally customize the heartbeat intervals (10 seconds for write inactivity, which causes a heartbeat to be sent, and 10 seconds for read inactivity, which closes the connection).

`WebSocketStompClient` sends a heartbeat only in case of inactivity, i.e. when no other messages are sent. This can present a challenge when using an external broker since messages with a non-broker destination represent activity but aren't actually forwarded to the broker. In that case you can configure a `TaskScheduler` when initializing the External Broker which ensures a heartbeat is forwarded to the broker also when only messages with a non-broker destination are sent.

When you use `WebSocketStompClient` for performance tests to simulate thousands of clients from the same machine, consider turning off heartbeats, since each connection schedules its own heartbeat tasks and that is not optimized for a large number of clients running on the same machine.

The STOMP protocol also supports receipts, where the client must add a `receipt` header to which the server responds with a RECEIPT frame after the send or subscribe are processed. To support this, the `StompSession` offers `setAutoReceipt(boolean)` that causes a `receipt` header to be added on every subsequent send or subscribe event. Alternatively, you can also manually add a receipt header to the `StompHeaders`. Both send and subscribe return an instance of `Receiptable` that you can use to register for receipt success and failure callbacks. For this feature, you must configure the client with a `TaskScheduler` and the amount of time before a receipt expires (15 seconds by default).

Note that `StompSessionHandler` itself is a `StompFrameHandler`, which lets it handle ERROR frames in addition to the `handleException` callback for exceptions from the handling of messages and `handleTransportError` for transport-level errors including `ConnectionLostException`.

WebSocket Scope

Each WebSocket session has a map of attributes. The map is attached as a header to inbound client messages and may be accessed from a controller method, as the following example shows:

```
@Controller
public class MyController {

    @MessageMapping("/action")
    public void handle(SimpMessageHeaderAccessor headerAccessor) {
        Map<String, Object> attrs = headerAccessor.getSessionAttributes();
        // ...
    }
}
```

You can declare a Spring-managed bean in the `websocket` scope. You can inject WebSocket-scoped beans into controllers and any channel interceptors registered on the `clientInboundChannel`. Those are typically singletons and live longer than any individual WebSocket session. Therefore, you need to use a scope proxy mode for WebSocket-scoped beans, as the following example shows:

```
@Component
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class MyBean {

    @PostConstruct
    public void init() {
        // Invoked after dependencies injected
    }

    // ...

    @PreDestroy
    public void destroy() {
        // Invoked when the WebSocket session ends
    }
}

@Controller
public class MyController {

    private final MyBean myBean;

    @Autowired
    public MyController(MyBean myBean) {
        this.myBean = myBean;
    }

    @MessageMapping("/action")
    public void handle() {
        // this.myBean from the current WebSocket session
    }
}
```

As with any custom scope, Spring initializes a new `MyBean` instance the first time it is accessed from the controller and stores the instance in the WebSocket session attributes. The same instance is subsequently returned until the session ends. WebSocket-scoped beans have all Spring lifecycle methods invoked, as shown in the preceding examples.

Performance

There is no silver bullet when it comes to performance. Many factors affect it, including the size and volume of messages, whether application methods perform work that requires blocking, and external factors (such as network speed and other issues). The goal of this section is to provide an overview of the available configuration options along with some thoughts on how to reason about scaling.

In a messaging application, messages are passed through channels for asynchronous executions that are backed by thread pools. Configuring such an application requires good knowledge of the channels and the flow of messages. Therefore, it is recommended to review Flow of Messages.

The obvious place to start is to configure the thread pools that back the `clientInboundChannel` and the `clientOutboundChannel`. By default, both are configured at twice the number of available processors.

If the handling of messages in annotated methods is mainly CPU-bound, the number of threads for the `clientInboundChannel` should remain close to the number of processors. If the work they do is more IO-bound and requires blocking or waiting on a database or other external system, the thread pool size probably needs to be increased.

`ThreadPoolExecutor` has three important properties: the core thread pool size, the max thread pool size, and the capacity for the queue to store tasks for which there are no available threads.

A common point of confusion is that configuring the core pool size (for example, 10) and max pool size (for example, 20) results in a thread pool with 10 to 20 threads. In fact, if the capacity is left at its default value of Integer.MAX_VALUE, the thread pool never increases beyond the core pool size, since all additional tasks are queued.

See the javadoc of `ThreadPoolExecutor` to learn how these properties work and understand the various queuing strategies.

On the `clientOutboundChannel` side, it is all about sending messages to WebSocket clients. If clients are on a fast network, the number of threads should remain close to the number of available processors. If they are slow or on low bandwidth, they take longer to consume messages and put a burden on the thread pool. Therefore, increasing the thread pool size becomes necessary.

While the workload for the `clientInboundChannel` is possible to predict — after all, it is based on what the application does — how to configure the "clientOutboundChannel" is harder, as it is based on factors beyond the control of the application. For this reason, two additional properties relate to the sending of messages: `sendTimeLimit` and `sendBufferSizeLimit`. You can use those methods to configure how long a send is allowed to take and how much data can be buffered when sending

messages to a client.

The general idea is that, at any given time, only a single thread can be used to send to a client. All additional messages, meanwhile, get buffered, and you can use these properties to decide how long sending a message is allowed to take and how much data can be buffered in the meantime. See the javadoc and documentation of the XML schema for important additional details.

The following example shows a possible configuration:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureWebSocketTransport(WebSocketTransportRegistration
registration) {
        registration.setSendTimeLimit(15 * 1000).setSendBufferSizeLimit(512 * 1024);
    }

    // ...

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker>
        <websocket:transport send-timeout="15000" send-buffer-size="524288" />
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

You can also use the WebSocket transport configuration shown earlier to configure the maximum allowed size for incoming STOMP messages. In theory, a WebSocket message can be almost unlimited in size. In practice, WebSocket servers impose limits — for example, 8K on Tomcat and 64K on Jetty. For this reason, STOMP clients (such as the JavaScript webstomp-client and others) split larger STOMP messages at 16K boundaries and send them as multiple WebSocket messages, which requires the server to buffer and re-assemble.

Spring's STOMP-over-WebSocket support does this ,so applications can configure the maximum size for STOMP messages irrespective of WebSocket server-specific message sizes. Keep in mind that the WebSocket message size is automatically adjusted, if necessary, to ensure they can carry 16K WebSocket messages at a minimum.

The following example shows one possible configuration:

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureWebSocketTransport(WebSocketTransportRegistration
registration) {
        registration.setMessageSizeLimit(128 * 1024);
    }

    // ...

}
```

The following example shows the XML configuration equivalent of the preceding example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:websocket="http://www.springframework.org/schema/websocket"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/websocket
        https://www.springframework.org/schema/websocket/spring-websocket.xsd">

    <websocket:message-broker>
        <websocket:transport message-size="131072" />
        <!-- ... -->
    </websocket:message-broker>

</beans>
```

An important point about scaling involves using multiple application instances. Currently, you cannot do that with the simple broker. However, when you use a full-featured broker (such as RabbitMQ), each application instance connects to the broker, and messages broadcast from one application instance can be broadcast through the broker to WebSocket clients connected through any other application instances.

Monitoring

When you use `@EnableWebSocketMessageBroker` or `<websocket:message-broker>`, key infrastructure

components automatically gather statistics and counters that provide important insight into the internal state of the application. The configuration also declares a bean of type `WebSocketMessageBrokerStats` that gathers all available information in one place and by default logs it at the `INFO` level once every 30 minutes. This bean can be exported to JMX through Spring's `MBeanExporter` for viewing at runtime (for example, through JDK's `jconsole`). The following list summarizes the available information:

Client WebSocket Sessions

Current

Indicates how many client sessions there are currently, with the count further broken down by WebSocket versus HTTP streaming and polling SockJS sessions.

Total

Indicates how many total sessions have been established.

Abnormally Closed

Connect Failures

Sessions that got established but were closed after not having received any messages within 60 seconds. This is usually an indication of proxy or network issues.

Send Limit Exceeded

Sessions closed after exceeding the configured send timeout or the send buffer limits, which can occur with slow clients (see previous section).

Transport Errors

Sessions closed after a transport error, such as failure to read or write to a WebSocket connection or HTTP request or response.

STOMP Frames

The total number of CONNECT, CONNECTED, and DISCONNECT frames processed, indicating how many clients connected on the STOMP level. Note that the DISCONNECT count may be lower when sessions get closed abnormally or when clients close without sending a DISCONNECT frame.

STOMP Broker Relay

TCP Connections

Indicates how many TCP connections on behalf of client WebSocket sessions are established to the broker. This should be equal to the number of client WebSocket sessions + 1 additional shared "system" connection for sending messages from within the application.

STOMP Frames

The total number of CONNECT, CONNECTED, and DISCONNECT frames forwarded to or received from the broker on behalf of clients. Note that a DISCONNECT frame is sent to the broker regardless of how the client WebSocket session was closed. Therefore, a lower DISCONNECT frame count is an indication that the broker is pro-actively closing connections (maybe because of a heartbeat that did not arrive in time, an invalid input frame, or other issue).

Client Inbound Channel

Statistics from the thread pool that backs the `clientInboundChannel` that provide insight into the health of incoming message processing. Tasks queueing up here is an indication that the application may be too slow to handle messages. If there I/O bound tasks (for example, slow database queries, HTTP requests to third party REST API, and so on), consider increasing the thread pool size.

Client Outbound Channel

Statistics from the thread pool that backs the `clientOutboundChannel` that provides insight into the health of broadcasting messages to clients. Tasks queueing up here is an indication clients are too slow to consume messages. One way to address this is to increase the thread pool size to accommodate the expected number of concurrent slow clients. Another option is to reduce the send timeout and send buffer size limits (see the previous section).

SockJS Task Scheduler

Statistics from the thread pool of the SockJS task scheduler that is used to send heartbeats. Note that, when heartbeats are negotiated on the STOMP level, the SockJS heartbeats are disabled.

Testing

There are two main approaches to testing applications when you use Spring's STOMP-over-WebSocket support. The first is to write server-side tests to verify the functionality of controllers and their annotated message-handling methods. The second is to write full end-to-end tests that involve running a client and a server.

The two approaches are not mutually exclusive. On the contrary, each has a place in an overall test strategy. Server-side tests are more focused and easier to write and maintain. End-to-end integration tests, on the other hand, are more complete and test much more, but they are also more involved to write and maintain.

The simplest form of server-side tests is to write controller unit tests. However, this is not useful enough, since much of what a controller does depends on its annotations. Pure unit tests simply cannot test that.

Ideally, controllers under test should be invoked as they are at runtime, much like the approach to testing controllers that handle HTTP requests by using the Spring MVC Test framework — that is, without running a Servlet container but relying on the Spring Framework to invoke the annotated controllers. As with Spring MVC Test, you have two possible alternatives here, either use a "context-based" or use a "standalone" setup:

- Load the actual Spring configuration with the help of the Spring TestContext framework, inject `clientInboundChannel` as a test field, and use it to send messages to be handled by controller methods.
- Manually set up the minimum Spring framework infrastructure required to invoke controllers (namely the `SimpAnnotationMethodMessageHandler`) and pass messages for controllers directly to it.

Both of these setup scenarios are demonstrated in the tests for the stock portfolio sample application.

The second approach is to create end-to-end integration tests. For that, you need to run a WebSocket server in embedded mode and connect to it as a WebSocket client that sends WebSocket messages containing STOMP frames. The tests for the stock portfolio sample application also demonstrate this approach by using Tomcat as the embedded WebSocket server and a simple STOMP client for test purposes.

5.5. Other Web Frameworks

This chapter details Spring's integration with third-party web frameworks.

One of the core value propositions of the Spring Framework is that of enabling *choice*. In a general sense, Spring does not force you to use or buy into any particular architecture, technology, or methodology (although it certainly recommends some over others). This freedom to pick and choose the architecture, technology, or methodology that is most relevant to a developer and their development team is arguably most evident in the web area, where Spring provides its own web frameworks (Spring MVC and Spring WebFlux) while, at the same time, supporting integration with a number of popular third-party web frameworks.

5.5.1. Common Configuration

Before diving into the integration specifics of each supported web framework, let us first take a look at common Spring configuration that is not specific to any one web framework. (This section is equally applicable to Spring's own web framework variants.)

One of the concepts (for want of a better word) espoused by Spring's lightweight application model is that of a layered architecture. Remember that in a "classic" layered architecture, the web layer is but one of many layers. It serves as one of the entry points into a server-side application, and it delegates to service objects (facades) that are defined in a service layer to satisfy business-specific (and presentation-technology agnostic) use cases. In Spring, these service objects, any other business-specific objects, data-access objects, and others exist in a distinct "business context", which contains no web or presentation layer objects (presentation objects, such as Spring MVC controllers, are typically configured in a distinct "presentation context"). This section details how you can configure a Spring container (a `WebApplicationContext`) that contains all of the 'business beans' in your application.

Moving on to specifics, all you need to do is declare a `ContextLoaderListener` in the standard Jakarta EE servlet `web.xml` file of your web application and add a `contextConfigLocation <context-param/>` section (in the same file) that defines which set of Spring XML configuration files to load.

Consider the following `<listener/>` configuration:

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
class>
</listener>
```

Further consider the following `<context-param/>` configuration:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

If you do not specify the `contextConfigLocation` context parameter, the `ContextLoaderListener` looks for a file called `/WEB-INF/applicationContext.xml` to load. Once the context files are loaded, Spring creates a `WebApplicationContext` object based on the bean definitions and stores it in the `ServletContext` of the web application.

All Java web frameworks are built on top of the Servlet API, so you can use the following code snippet to get access to this "business context" `ApplicationContext` created by the `ContextLoaderListener`.

The following example shows how to get the `WebApplicationContext`:

```
WebApplicationContext ctx =
WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

The `WebApplicationContextUtils` class is for convenience, so you need not remember the name of the `ServletContext` attribute. Its `getWebApplicationContext()` method returns `null` if an object does not exist under the `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` key. Rather than risk getting `NullPointerExceptions` in your application, it is better to use the `getRequiredWebApplicationContext()` method. This method throws an exception when the `ApplicationContext` is missing.

Once you have a reference to the `WebApplicationContext`, you can retrieve beans by their name or type. Most developers retrieve beans by name and then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they make it easy to get beans from a Spring container, but they also let you use dependency injection on their controllers. Each web framework section has more detail on its specific integration strategies.

5.5.2. JSF

JavaServer Faces (JSF) is the JCP's standard component-based, event-driven web user interface framework. It is an official part of the Jakarta EE umbrella but also individually usable, e.g. through embedding Mojarra or MyFaces within Tomcat.

Please note that recent versions of JSF became closely tied to CDI infrastructure in application servers, with some new JSF functionality only working in such an environment. Spring's JSF support is not actively evolved anymore and primarily exists for migration purposes when modernizing older JSF-based applications.

The key element in Spring's JSF integration is the JSF `ELResolver` mechanism.

Spring Bean Resolver

`SpringBeanFacesELResolver` is a JSF compliant `ELResolver` implementation, integrating with the standard Unified EL as used by JSF and JSP. It delegates to Spring's "business context" `WebApplicationContext` first and then to the default resolver of the underlying JSF implementation.

Configuration-wise, you can define `SpringBeanFacesELResolver` in your JSF `faces-context.xml` file, as the following example shows:

```
<faces-config>
    <application>
        <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-
resolver>
        ...
    </application>
</faces-config>
```

Using `FacesContextUtils`

A custom `ELResolver` works well when mapping your properties to beans in `faces-config.xml`, but, at times, you may need to explicitly grab a bean. The `FacesContextUtils` class makes this easy. It is similar to `WebApplicationContextUtils`, except that it takes a `FacesContext` parameter rather than a `ServletContext` parameter.

The following example shows how to use `FacesContextUtils`:

```
ApplicationContext ctx =
FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

5.5.3. Apache Struts

Invented by Craig McClanahan, Struts is an open-source project hosted by the Apache Software Foundation. Struts 1.x greatly simplified the JSP/Servlet programming paradigm and won over many developers who were using proprietary frameworks. It simplified the programming model; it was open source; and it had a large community, which let the project grow and become popular among Java web developers.

As a successor to the original Struts 1.x, check out Struts 2.x or more recent versions as well as the Struts-provided Spring Plugin for built-in Spring integration.

5.5.4. Apache Tapestry

Tapestry is a "Component oriented framework for creating dynamic, robust, highly scalable web applications in Java."

While Spring has its own powerful web layer, there are a number of unique advantages to building an enterprise Java application by using a combination of Tapestry for the web user interface and the Spring container for the lower layers.

For more information, see Tapestry's dedicated integration module for Spring.

5.5.5. Further Resources

The following links go to further resources about the various web frameworks described in this chapter.

- The JSF homepage
- The Struts homepage
- The Tapestry homepage

Chapter 6. Web on Reactive Stack

This part of the documentation covers support for reactive-stack web applications built on a Reactive Streams API to run on non-blocking servers, such as Netty, Undertow, and Servlet containers. Individual chapters cover the Spring WebFlux framework, the reactive `WebClient`, support for testing, and reactive libraries. For Servlet-stack web applications, see Web on Servlet Stack.

6.1. Spring WebFlux

The original web framework included in the Spring Framework, Spring Web MVC, was purpose-built for the Servlet API and Servlet containers. The reactive-stack web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports Reactive Streams back pressure, and runs on such servers as Netty, Undertow, and Servlet containers.

Both web frameworks mirror the names of their source modules (spring-webmvc and spring-webflux) and co-exist side by side in the Spring Framework. Each module is optional. Applications can use one or the other module or, in some cases, both — for example, Spring MVC controllers with the reactive `WebClient`.

6.1.1. Overview

Why was Spring WebFlux created?

Part of the answer is the need for a non-blocking web stack to handle concurrency with a small number of threads and scale with fewer hardware resources. Servlet non-blocking I/O leads away from the rest of the Servlet API, where contracts are synchronous (`Filter`, `Servlet`) or blocking (`getParameter`, `getPart`). This was the motivation for a new common API to serve as a foundation across any non-blocking runtime. That is important because of servers (such as Netty) that are well-established in the async, non-blocking space.

The other part of the answer is functional programming. Much as the addition of annotations in Java 5 created opportunities (such as annotated REST controllers or unit tests), the addition of lambda expressions in Java 8 created opportunities for functional APIs in Java. This is a boon for non-blocking applications and continuation-style APIs (as popularized by `CompletableFuture` and ReactiveX) that allow declarative composition of asynchronous logic. At the programming-model level, Java 8 enabled Spring WebFlux to offer functional web endpoints alongside annotated controllers.

Define “Reactive”

We touched on “non-blocking” and “functional” but what does reactive mean?

The term, “reactive,” refers to programming models that are built around reacting to change — network components reacting to I/O events, UI controllers reacting to mouse events, and others. In that sense, non-blocking is reactive, because, instead of being blocked, we are now in the mode of reacting to notifications as operations complete or data becomes available.

There is also another important mechanism that we on the Spring team associate with “reactive” and that is non-blocking back pressure. In synchronous, imperative code, blocking calls serve as a natural form of back pressure that forces the caller to wait. In non-blocking code, it becomes important to control the rate of events so that a fast producer does not overwhelm its destination.

Reactive Streams is a small spec (also adopted in Java 9) that defines the interaction between asynchronous components with back pressure. For example a data repository (acting as Publisher) can produce data that an HTTP server (acting as Subscriber) can then write to the response. The main purpose of Reactive Streams is to let the subscriber control how quickly or how slowly the publisher produces data.

Common question: what if a publisher cannot slow down?
The purpose of Reactive Streams is only to establish the mechanism and a boundary. If a publisher cannot slow down, it has to decide whether to buffer, drop, or fail.

Reactive API

Reactive Streams plays an important role for interoperability. It is of interest to libraries and infrastructure components but less useful as an application API, because it is too low-level. Applications need a higher-level and richer, functional API to compose async logic — similar to the Java 8 `Stream` API but not only for collections. This is the role that reactive libraries play.

Reactor is the reactive library of choice for Spring WebFlux. It provides the `Mono` and `Flux` API types to work on data sequences of 0..1 (`Mono`) and 0..N (`Flux`) through a rich set of operators aligned with the ReactiveX vocabulary of operators. Reactor is a Reactive Streams library and, therefore, all of its operators support non-blocking back pressure. Reactor has a strong focus on server-side Java. It is developed in close collaboration with Spring.

WebFlux requires Reactor as a core dependency but it is interoperable with other reactive libraries via Reactive Streams. As a general rule, a WebFlux API accepts a plain `Publisher` as input, adapts it to a Reactor type internally, uses that, and returns either a `Flux` or a `Mono` as output. So, you can pass any `Publisher` as input and you can apply operations on the output, but you need to adapt the output for use with another reactive library. Whenever feasible (for example, annotated controllers), WebFlux adapts transparently to the use of RxJava or another reactive library. See Reactive Libraries for more details.

In addition to Reactive APIs, WebFlux can also be used with Coroutines APIs in Kotlin which provides a more imperative style of programming. The following Kotlin code samples will be provided with Coroutines APIs.

Programming Models

The `spring-web` module contains the reactive foundation that underlies Spring WebFlux, including HTTP abstractions, Reactive Streams adapters for supported servers, codecs, and a core `WebHandler` API comparable to the Servlet API but with non-blocking contracts.

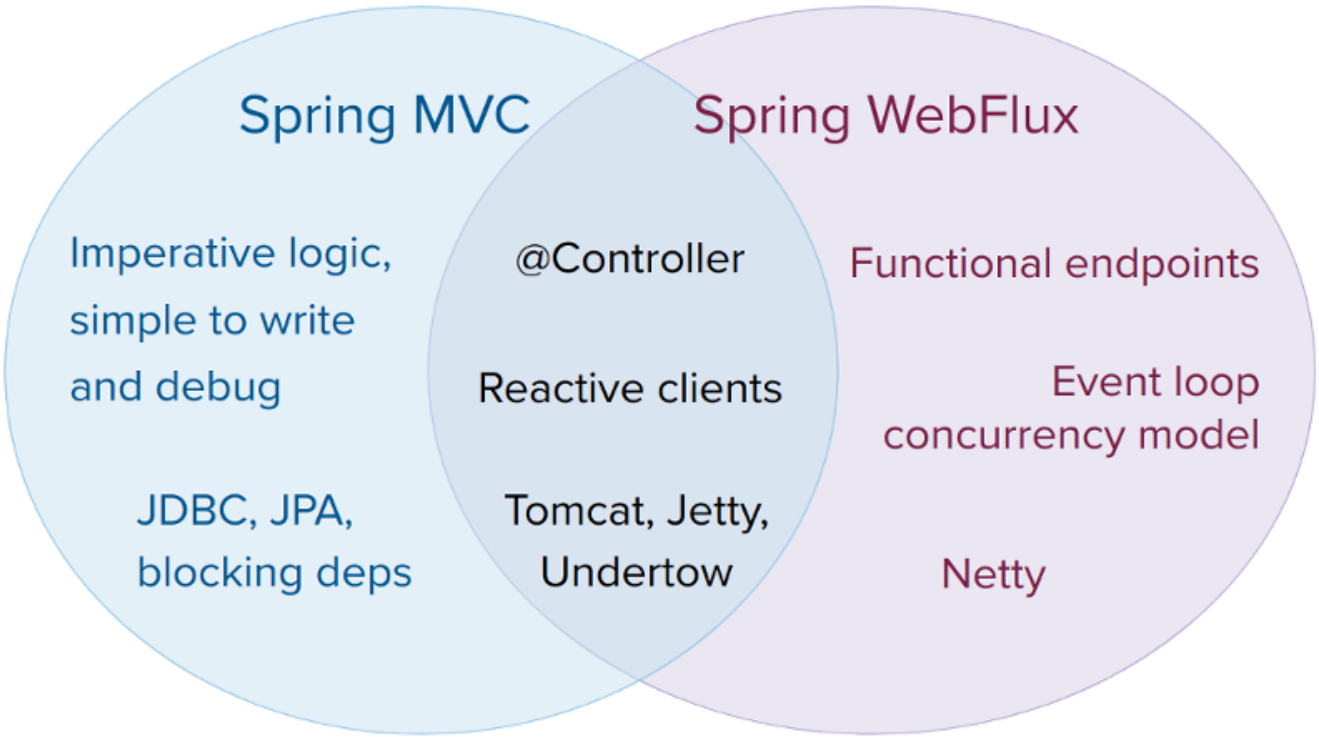
On that foundation, Spring WebFlux provides a choice of two programming models:

- Annotated Controllers: Consistent with Spring MVC and based on the same annotations from the `spring-web` module. Both Spring MVC and WebFlux controllers support reactive (Reactor and RxJava) return types, and, as a result, it is not easy to tell them apart. One notable difference is that WebFlux also supports reactive `@RequestBody` arguments.
- Functional Endpoints: Lambda-based, lightweight, and functional programming model. You can think of this as a small library or a set of utilities that an application can use to route and handle requests. The big difference with annotated controllers is that the application is in charge of request handling from start to finish versus declaring intent through annotations and being called back.

Applicability

Spring MVC or WebFlux?

A natural question to ask but one that sets up an unsound dichotomy. Actually, both work together to expand the range of available options. The two are designed for continuity and consistency with each other, they are available side by side, and feedback from each side benefits both sides. The following diagram shows how the two relate, what they have in common, and what each supports uniquely:



We suggest that you consider the following specific points:

- If you have a Spring MVC application that works fine, there is no need to change. Imperative programming is the easiest way to write, understand, and debug code. You have maximum choice of libraries, since, historically, most are blocking.
- If you are already shopping for a non-blocking web stack, Spring WebFlux offers the same execution model benefits as others in this space and also provides a choice of servers (Netty, Tomcat, Jetty, Undertow, and Servlet containers), a choice of programming models (annotated controllers and functional web endpoints), and a choice of reactive libraries (Reactor, RxJava, or

other).

- If you are interested in a lightweight, functional web framework for use with Java 8 lambdas or Kotlin, you can use the Spring WebFlux functional web endpoints. That can also be a good choice for smaller applications or microservices with less complex requirements that can benefit from greater transparency and control.
- In a microservice architecture, you can have a mix of applications with either Spring MVC or Spring WebFlux controllers or with Spring WebFlux functional endpoints. Having support for the same annotation-based programming model in both frameworks makes it easier to re-use knowledge while also selecting the right tool for the right job.
- A simple way to evaluate an application is to check its dependencies. If you have blocking persistence APIs (JPA, JDBC) or networking APIs to use, Spring MVC is the best choice for common architectures at least. It is technically feasible with both Reactor and RxJava to perform blocking calls on a separate thread but you would not be making the most of a non-blocking web stack.
- If you have a Spring MVC application with calls to remote services, try the reactive `WebClient`. You can return reactive types (Reactor, RxJava, or other) directly from Spring MVC controller methods. The greater the latency per call or the interdependency among calls, the more dramatic the benefits. Spring MVC controllers can call other reactive components too.
- If you have a large team, keep in mind the steep learning curve in the shift to non-blocking, functional, and declarative programming. A practical way to start without a full switch is to use the reactive `WebClient`. Beyond that, start small and measure the benefits. We expect that, for a wide range of applications, the shift is unnecessary. If you are unsure what benefits to look for, start by learning about how non-blocking I/O works (for example, concurrency on single-threaded Node.js) and its effects.

Servers

Spring WebFlux is supported on Tomcat, Jetty, Servlet containers, as well as on non-Servlet runtimes such as Netty and Undertow. All servers are adapted to a low-level, common API so that higher-level programming models can be supported across servers.

Spring WebFlux does not have built-in support to start or stop a server. However, it is easy to assemble an application from Spring configuration and WebFlux infrastructure and run it with a few lines of code.

Spring Boot has a WebFlux starter that automates these steps. By default, the starter uses Netty, but it is easy to switch to Tomcat, Jetty, or Undertow by changing your Maven or Gradle dependencies. Spring Boot defaults to Netty, because it is more widely used in the asynchronous, non-blocking space and lets a client and a server share resources.

Tomcat and Jetty can be used with both Spring MVC and WebFlux. Keep in mind, however, that the way they are used is very different. Spring MVC relies on Servlet blocking I/O and lets applications use the Servlet API directly if they need to. Spring WebFlux relies on Servlet non-blocking I/O and uses the Servlet API behind a low-level adapter. It is not exposed for direct use.

For Undertow, Spring WebFlux uses Undertow APIs directly without the Servlet API.

Performance

Performance has many characteristics and meanings. Reactive and non-blocking generally do not make applications run faster. They can, in some cases, (for example, if using the `WebClient` to run remote calls in parallel). On the whole, it requires more work to do things the non-blocking way and that can slightly increase the required processing time.

The key expected benefit of reactive and non-blocking is the ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load, because they scale in a more predictable way. In order to observe those benefits, however, you need to have some latency (including a mix of slow and unpredictable network I/O). That is where the reactive stack begins to show its strengths, and the differences can be dramatic.

Concurrency Model

Both Spring MVC and Spring WebFlux support annotated controllers, but there is a key difference in the concurrency model and the default assumptions for blocking and threads.

In Spring MVC (and servlet applications in general), it is assumed that applications can block the current thread, (for example, for remote calls). For this reason, servlet containers use a large thread pool to absorb potential blocking during request handling.

In Spring WebFlux (and non-blocking servers in general), it is assumed that applications do not block. Therefore, non-blocking servers use a small, fixed-size thread pool (event loop workers) to handle requests.

"To scale" and "small number of threads" may sound contradictory but to never block the current thread (and rely on callbacks instead) means that you do not need extra threads, as there are no blocking calls to absorb.

Invoking a Blocking API

What if you do need to use a blocking library? Both Reactor and RxJava provide the `publishOn` operator to continue processing on a different thread. That means there is an easy escape hatch. Keep in mind, however, that blocking APIs are not a good fit for this concurrency model.

Mutable State

In Reactor and RxJava, you declare logic through operators. At runtime, a reactive pipeline is formed where data is processed sequentially, in distinct stages. A key benefit of this is that it frees applications from having to protect mutable state because application code within that pipeline is never invoked concurrently.

Threading Model

What threads should you expect to see on a server running with Spring WebFlux?

- On a "vanilla" Spring WebFlux server (for example, no data access nor other optional dependencies), you can expect one thread for the server and several others for request processing (typically as many as the number of CPU cores). Servlet containers, however, may start with more threads (for example, 10 on Tomcat), in support of both servlet (blocking) I/O and servlet 3.1 (non-blocking) I/O usage.

- The reactive `WebClient` operates in event loop style. So you can see a small, fixed number of processing threads related to that (for example, `reactor-http-nio-` with the Reactor Netty connector). However, if Reactor Netty is used for both client and server, the two share event loop resources by default.
- Reactor and RxJava provide thread pool abstractions, called schedulers, to use with the `publishOn` operator that is used to switch processing to a different thread pool. The schedulers have names that suggest a specific concurrency strategy — for example, "parallel" (for CPU-bound work with a limited number of threads) or "elastic" (for I/O-bound work with a large number of threads). If you see such threads, it means some code is using a specific thread pool `Scheduler` strategy.
- Data access libraries and other third party dependencies can also create and use threads of their own.

Configuring

The Spring Framework does not provide support for starting and stopping servers. To configure the threading model for a server, you need to use server-specific configuration APIs, or, if you use Spring Boot, check the Spring Boot configuration options for each server. You can configure the `WebClient` directly. For all other libraries, see their respective documentation.

6.1.2. Reactive Core

The `spring-web` module contains the following foundational support for reactive web applications:

- For server request processing there are two levels of support.
  - HttpHandler: Basic contract for HTTP request handling with non-blocking I/O and Reactive Streams back pressure, along with adapters for Reactor Netty, Undertow, Tomcat, Jetty, and any Servlet container.
  - `WebHandler` API: Slightly higher level, general-purpose web API for request handling, on top of which concrete programming models such as annotated controllers and functional endpoints are built.
- For the client side, there is a basic `ClientHttpConnector` contract to perform HTTP requests with non-blocking I/O and Reactive Streams back pressure, along with adapters for Reactor Netty, reactive Jetty HttpClient and Apache HttpComponents. The higher level WebClient used in applications builds on this basic contract.
- For client and server, codecs for serialization and deserialization of HTTP request and response content.

`HttpHandler`

HttpHandler is a simple contract with a single method to handle a request and a response. It is intentionally minimal, and its main and only purpose is to be a minimal abstraction over different HTTP server APIs.

The following table describes the supported server APIs:

Server name	Server API used	Reactive Streams support
Netty	Netty API	Reactor Netty
Undertow	Undertow API	spring-web: Undertow to Reactive Streams bridge
Tomcat	Servlet non-blocking I/O; Tomcat API to read and write ByteBuffers vs byte[]	spring-web: Servlet non-blocking I/O to Reactive Streams bridge
Jetty	Servlet non-blocking I/O; Jetty API to write ByteBuffers vs byte[]	spring-web: Servlet non-blocking I/O to Reactive Streams bridge
Servlet container	Servlet non-blocking I/O	spring-web: Servlet non-blocking I/O to Reactive Streams bridge

The following table describes server dependencies (also see supported versions):

Server name	Group id	Artifact name
Reactor Netty	io.projectreactor.netty	reactor-netty
Undertow	io.undertow	undertow-core
Tomcat	org.apache.tomcat.embed	tomcat-embed-core
Jetty	org.eclipse.jetty	jetty-server, jetty-servlet

The code snippets below show using the `HttpHandler` adapters with each server API:

Reactor Netty

Java

```
HttpHandler handler = ...
ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);
HttpServer.create().host(host).port(port).handle(adapter).bind().block();
```

Kotlin

```
val handler: HttpHandler = ...
val adapter = ReactorHttpHandlerAdapter(handler)
HttpServer.create().host(host).port(port).handle(adapter).bind().block()
```

Undertow

Java

```
HttpHandler handler = ...
UndertowHttpHandlerAdapter adapter = new UndertowHttpHandlerAdapter(handler);
Undertow server = Undertow.builder().addHttpListener(port,
host).setHandler(adapter).build();
server.start();
```

Kotlin

```
val handler: HttpHandler = ...
val adapter = UndertowHttpHandlerAdapter(handler)
val server = Undertow.builder().addHttpListener(port,
host).setHandler(adapter).build()
server.start()
```

Tomcat

Java

```
HttpHandler handler = ...
Servlet servlet = new TomcatHttpHandlerAdapter(handler);

Tomcat server = new Tomcat();
File base = new File(System.getProperty("java.io.tmpdir"));
Context rootContext = server.addContext("", base.getAbsolutePath());
Tomcat.addServlet(rootContext, "main", servlet);
rootContext.addServletMappingDecoded("/", "main");
server.setHost(host);
server.setPort(port);
server.start();
```

Kotlin

```
val handler: HttpHandler = ...
val servlet = TomcatHttpHandlerAdapter(handler)

val server = Tomcat()
val base = File(System.getProperty("java.io.tmpdir"))
val rootContext = server.addContext("", base.absolutePath)
Tomcat.addServlet(rootContext, "main", servlet)
rootContext.addServletMappingDecoded("/", "main")
server.host = host
server.setPort(port)
server.start()
```

Jetty

Java

```
HttpHandler handler = ...
Servlet servlet = new JettyHttpHandlerAdapter(handler);

Server server = new Server();
ServletContextHandler contextHandler = new ServletContextHandler(server, "");
contextHandler.addServlet(new ServletHolder(servlet), "/");
contextHandler.start();

ServerConnector connector = new ServerConnector(server);
connector.setHost(host);
connector.setPort(port);
server.addConnector(connector);
server.start();
```

Kotlin

```
val handler: HttpHandler = ...
val servlet = JettyHttpHandlerAdapter(handler)

val server = Server()
val contextHandler = ServletContextHandler(server, "")
contextHandler.addServlet(ServletHolder(servlet), "/")
contextHandler.start();

val connector = ServerConnector(server)
connector.host = host
connector.port = port
server.addConnector(connector)
server.start()
```

Servlet Container

To deploy as a WAR to any Servlet container, you can extend and include `AbstractReactiveWebInitializer` in the WAR. That class wraps an `HttpHandler` with `ServletHttpHandlerAdapter` and registers that as a `Servlet`.

`WebHandler` API

The `org.springframework.web.server` package builds on the `HttpHandler` contract to provide a general-purpose web API for processing requests through a chain of multiple `WebExceptionHandler`, multiple `WebFilter`, and a single `WebHandler` component. The chain can be put together with `WebHttpHandlerBuilder` by simply pointing to a Spring `ApplicationContext` where components are auto-detected, and/or by registering components with the builder.

While `HttpHandler` has a simple goal to abstract the use of different HTTP servers, the `WebHandler` API aims to provide a broader set of features commonly used in web applications such as:

- User session with attributes.

- Request attributes.
- Resolved `Locale` or `Principal` for the request.
- Access to parsed and cached form data.
- Abstractions for multipart data.
- and more..

Special bean types

The table below lists the components that `WebHttpHandlerBuilder` can auto-detect in a Spring ApplicationContext, or that can be registered directly with it:

Bean name	Bean type	Count	Description
<any>	`WebExceptionHandler`	0..N	Provide handling for exceptions from the chain of `WebFilter` instances and the target `WebHandler`. For more details, see Exceptions.
<any>	`WebFilter`	0..N	Apply interception style logic to before and after the rest of the filter chain and the target `WebHandler`. For more details, see Filters.
`webHandler`	`WebHandler`	1	The handler for the request.
`webSessionManager`	`WebSessionManager`	0..1	The manager for `WebSession` instances exposed through a method on `ServerWebExchange`. `DefaultWebSessionManager` by default.
`serverCodecConfigurer`	`ServerCodecConfigurer`	0..1	For access to `HttpMessageReader` instances for parsing form data and multipart data that is then exposed through methods on `ServerWebExchange`. `ServerCodecConfigurer.create()` by default.
`localeContextResolver`	`LocaleContextResolver`	0..1	The resolver for `LocaleContext` exposed through a method on `ServerWebExchange`. `AcceptHeaderLocaleContextResolver` by default.
`forwardedHeaderTransformer`	`ForwardedHeaderTransformer`	0..1	For processing forwarded type headers, either by extracting and removing them or by removing them only. Not used by default.

Form Data

`ServerWebExchange` exposes the following method for accessing form data:

Java

```
Mono<MultiValueMap<String, String>> getFormData();
```

Kotlin

```
suspend fun getFormData(): MultiValueMap<String, String>
```

The `DefaultServerWebExchange` uses the configured `HttpMessageReader` to parse form data (`application/x-www-form-urlencoded`) into a `MultiValueMap`. By default, `FormHttpMessageReader` is configured for use by the `ServerCodecConfigurer` bean (see the Web Handler API).

Multipart Data

Web MVC

`ServerWebExchange` exposes the following method for accessing multipart data:

Java

```
Mono<MultiValueMap<String, Part>> getMultipartData();
```

Kotlin

```
suspend fun getMultipartData(): MultiValueMap<String, Part>
```

The `DefaultServerWebExchange` uses the configured `HttpMessageReader<MultiValueMap<String, Part>>` to parse `multipart/form-data` content into a `MultiValueMap`. By default, this is the `DefaultPartHttpMessageReader`, which does not have any third-party dependencies. Alternatively, the `SynchronossPartHttpMessageReader` can be used, which is based on the Synchronoss NIO Multipart library. Both are configured through the `ServerCodecConfigurer` bean (see the Web Handler API).

To parse multipart data in streaming fashion, you can use the `Flux<PartEvent>` returned from the `PartEventHttpMessageReader` instead of using `@RequestPart`, as that implies `Map`-like access to individual parts by name and, hence, requires parsing multipart data in full. By contrast, you can use `@RequestBody` to decode the content to `Flux<PartEvent>` without collecting to a `MultiValueMap`.

Forwarded Headers

Web MVC

As a request goes through proxies (such as load balancers), the host, port, and scheme may change. That makes it a challenge, from a client perspective, to create links that point to the correct host, port, and scheme.

RFC 7239 defines the `Forwarded` HTTP header that proxies can use to provide information about the

original request. There are other non-standard headers, too, including `X-Forwarded-Host`, `X-Forwarded-Port`, `X-Forwarded-Proto`, `X-Forwarded-Ssl`, and `X-Forwarded-Prefix`.

`ForwardedHeaderTransformer` is a component that modifies the host, port, and scheme of the request, based on forwarded headers, and then removes those headers. If you declare it as a bean with the name `forwardedHeaderTransformer`, it will be detected and used.

There are security considerations for forwarded headers, since an application cannot know if the headers were added by a proxy, as intended, or by a malicious client. This is why a proxy at the boundary of trust should be configured to remove untrusted forwarded traffic coming from the outside. You can also configure the `ForwardedHeaderTransformer` with `removeOnly=true`, in which case it removes but does not use the headers.

In 5.1 `ForwardedHeaderFilter` was deprecated and superseded by `ForwardedHeaderTransformer` so forwarded headers can be processed earlier, before the exchange is created. If the filter is configured anyway, it is taken out of the list of filters, and `ForwardedHeaderTransformer` is used instead.

Filters

Web MVC

In the `WebHandler` API, you can use a `WebFilter` to apply interception-style logic before and after the rest of the processing chain of filters and the target `WebHandler`. When using the WebFlux Config, registering a `WebFilter` is as simple as declaring it as a Spring bean and (optionally) expressing precedence by using `@Order` on the bean declaration or by implementing `Ordered`.

CORS

Web MVC

Spring WebFlux provides fine-grained support for CORS configuration through annotations on controllers. However, when you use it with Spring Security, we advise relying on the built-in `CorsFilter`, which must be ordered ahead of Spring Security's chain of filters.

See the section on CORS and the CORS `WebFilter` for more details.

Exceptions

Web MVC

In the `WebHandler` API, you can use a `WebExceptionHandler` to handle exceptions from the chain of `WebFilter` instances and the target `WebHandler`. When using the WebFlux Config, registering a `WebExceptionHandler` is as simple as declaring it as a Spring bean and (optionally) expressing precedence by using `@Order` on the bean declaration or by implementing `Ordered`.

The following table describes the available `WebExceptionHandler` implementations:

Exception Handler	Description
`ResponseStatusExceptionHandler`	Provides handling for exceptions of type `ResponseStatusException` by setting the response to the HTTP status code of the exception.
`WebFluxResponseStatusExceptionHandler`	Extension of `ResponseStatusExceptionHandler` that can also determine the HTTP status code of a `@ResponseStatus` annotation on any exception. This handler is declared in the WebFlux Config.

Codecs

Web MVC

The `spring-web` and `spring-core` modules provide support for serializing and deserializing byte content to and from higher level objects through non-blocking I/O with Reactive Streams back pressure. The following describes this support:

- `Encoder` and `Decoder` are low level contracts to encode and decode content independent of HTTP.
- `HttpMessageReader` and `HttpMessageWriter` are contracts to encode and decode HTTP message content.
- An `Encoder` can be wrapped with `EncoderHttpMessageWriter` to adapt it for use in a web application, while a `Decoder` can be wrapped with `DecoderHttpMessageReader`.
- `DataBuffer` abstracts different byte buffer representations (e.g. Netty `ByteBuf`, `java.nio.ByteBuffer`, etc.) and is what all codecs work on. See Data Buffers and Codecs in the "Spring Core" section for more on this topic.

The `spring-core` module provides `byte[]`, `ByteBuffer`, `DataBuffer`, `Resource`, and `String` encoder and decoder implementations. The `spring-web` module provides Jackson JSON, Jackson Smile, JAXB2, Protocol Buffers and other encoders and decoders along with web-only HTTP message reader and writer implementations for form data, multipart content, server-sent events, and others.

`ClientCodecConfigurer` and `ServerCodecConfigurer` are typically used to configure and customize the codecs to use in an application. See the section on configuring HTTP message codecs.

Jackson JSON

JSON and binary JSON (Smile) are both supported when the Jackson library is present.

The `Jackson2Decoder` works as follows:

- Jackson's asynchronous, non-blocking parser is used to aggregate a stream of byte chunks into `TokenBuffer`'s each representing a JSON object.
- Each `TokenBuffer` is passed to Jackson's `ObjectMapper` to create a higher level object.
- When decoding to a single-value publisher (e.g. `Mono`), there is one `TokenBuffer`.
- When decoding to a multi-value publisher (e.g. `Flux`), each `TokenBuffer` is passed to the `ObjectMapper` as soon as enough bytes are received for a fully formed object. The input content can be a JSON array, or any line-delimited JSON format such as NDJSON, JSON Lines, or JSON

Text Sequences.

The `Jackson2Encoder` works as follows:

- For a single value publisher (e.g. `Mono`), simply serialize it through the `ObjectMapper`.
- For a multi-value publisher with `application/json`, by default collect the values with `Flux#collectToList()` and then serialize the resulting collection.
- For a multi-value publisher with a streaming media type such as `application/x-ndjson` or `application/stream+x-jackson-smile`, encode, write, and flush each value individually using a line-delimited JSON format. Other streaming media types may be registered with the encoder.
- For SSE the `Jackson2Encoder` is invoked per event and the output is flushed to ensure delivery without delay.

By default both `Jackson2Encoder` and `Jackson2Decoder` do not support elements of type `String`. Instead the default assumption is that a string or a sequence of strings represent serialized JSON content, to be rendered by the `CharSequenceEncoder`. If what you need is to render a JSON array from `Flux<String>`, use `Flux#collectToList()` and encode a `Mono<List<String>>`.

Form Data

`FormHttpMessageReader` and `FormHttpMessageWriter` support decoding and encoding `application/x-www-form-urlencoded` content.

On the server side where form content often needs to be accessed from multiple places, `ServerWebExchange` provides a dedicated `getFormData()` method that parses the content through `FormHttpMessageReader` and then caches the result for repeated access. See Form Data in the `WebHandler` API section.

Once `getFormData()` is used, the original raw content can no longer be read from the request body. For this reason, applications are expected to go through `ServerWebExchange` consistently for access to the cached form data versus reading from the raw request body.

Multipart

`MultipartHttpMessageReader` and `MultipartHttpMessageWriter` support decoding and encoding "multipart/form-data" content. In turn `MultipartHttpMessageReader` delegates to another `HttpMessageReader` for the actual parsing to a `Flux<Part>` and then simply collects the parts into a `MultiValueMap`. By default, the `DefaultPartHttpMessageReader` is used, but this can be changed through the `ServerCodecConfigurer`. For more information about the `DefaultPartHttpMessageReader`, refer to to the javadoc of `DefaultPartHttpMessageReader`.

On the server side where multipart form content may need to be accessed from multiple places, `ServerWebExchange` provides a dedicated `getMultipartData()` method that parses the content through `MultipartHttpMessageReader` and then caches the result for repeated access. See Multipart Data in the `WebHandler` API section.

Once `getMultipartData()` is used, the original raw content can no longer be read from the request body. For this reason applications have to consistently use `getMultipartData()` for repeated, map-

like access to parts, or otherwise rely on the `SynchronossPartHttpMessageReader` for a one-time access to `Flux<Part>`.

Limits

`Decoder` and `HttpMessageReader` implementations that buffer some or all of the input stream can be configured with a limit on the maximum number of bytes to buffer in memory. In some cases buffering occurs because input is aggregated and represented as a single object — for example, a controller method with `@RequestBody byte[]`, `x-www-form-urlencoded` data, and so on. Buffering can also occur with streaming, when splitting the input stream — for example, delimited text, a stream of JSON objects, and so on. For those streaming cases, the limit applies to the number of bytes associated with one object in the stream.

To configure buffer sizes, you can check if a given `Decoder` or `HttpMessageReader` exposes a `maxInMemorySize` property and if so the Javadoc will have details about default values. On the server side, `ServerCodecConfigurer` provides a single place from where to set all codecs, see HTTP message codecs. On the client side, the limit for all codecs can be changed in WebClient.Builder.

For Multipart parsing the `maxInMemorySize` property limits the size of non-file parts. For file parts, it determines the threshold at which the part is written to disk. For file parts written to disk, there is an additional `maxDiskUsagePerPart` property to limit the amount of disk space per part. There is also a `maxParts` property to limit the overall number of parts in a multipart request. To configure all three in WebFlux, you'll need to supply a pre-configured instance of `MultipartHttpMessageReader` to `ServerCodecConfigurer`.

Streaming

Web MVC

When streaming to the HTTP response (for example, `text/event-stream`, `application/x-ndjson`), it is important to send data periodically, in order to reliably detect a disconnected client sooner rather than later. Such a send could be a comment-only, empty SSE event or any other "no-op" data that would effectively serve as a heartbeat.

`DataBuffer`

`DataBuffer` is the representation for a byte buffer in WebFlux. The Spring Core part of this reference has more on that in the section on Data Buffers and Codecs. The key point to understand is that on some servers like Netty, byte buffers are pooled and reference counted, and must be released when consumed to avoid memory leaks.

WebFlux applications generally do not need to be concerned with such issues, unless they consume or produce data buffers directly, as opposed to relying on codecs to convert to and from higher level objects, or unless they choose to create custom codecs. For such cases please review the information in Data Buffers and Codecs, especially the section on Using DataBuffer.

Logging

Web MVC

`DEBUG` level logging in Spring WebFlux is designed to be compact, minimal, and human-friendly. It

focuses on high value bits of information that are useful over and over again vs others that are useful only when debugging a specific issue.

`TRACE` level logging generally follows the same principles as `DEBUG` (and for example also should not be a firehose) but can be used for debugging any issue. In addition, some log messages may show a different level of detail at `TRACE` vs `DEBUG`.

Good logging comes from the experience of using the logs. If you spot anything that does not meet the stated goals, please let us know.

Log Id

In WebFlux, a single request can be run over multiple threads and the thread ID is not useful for correlating log messages that belong to a specific request. This is why WebFlux log messages are prefixed with a request-specific ID by default.

On the server side, the log ID is stored in the `ServerWebExchange` attribute (`LOG_ID_ATTRIBUTE`), while a fully formatted prefix based on that ID is available from `ServerWebExchange#getLogPrefix()`. On the `WebClient` side, the log ID is stored in the `ClientRequest` attribute (`LOG_ID_ATTRIBUTE`) ,while a fully formatted prefix is available from `ClientRequest#logPrefix()`.

Sensitive Data

Web MVC

`DEBUG` and `TRACE` logging can log sensitive information. This is why form parameters and headers are masked by default and you must explicitly enable their logging in full.

The following example shows how to do so for server-side requests:

Java

```
@Configuration
@EnableWebFlux
class MyConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class MyConfig : WebFluxConfigurer {

    override fun configureHttpMessageCodecs(configurer: ServerCodecConfigurer) {
        configurer.defaultCodecs().enableLoggingRequestDetails(true)
    }
}
```

The following example shows how to do so for client-side requests:

Java

```
Consumer<ClientCodecConfigurer> consumer = configurer ->
        configurer.defaultCodecs().enableLoggingRequestDetails(true);

WebClient webClient = WebClient.builder()
        .exchangeStrategies(strategies -> strategies.codecs(consumer))
        .build();
```

Kotlin

```
val consumer: (ClientCodecConfigurer) -> Unit  = { configurer ->
configurer.defaultCodecs().enableLoggingRequestDetails(true) }

val webClient = WebClient.builder()
        .exchangeStrategies({ strategies -> strategies.codecs(consumer) })
        .build()
```

Appenders

Logging libraries such as SLF4J and Log4J 2 provide asynchronous loggers that avoid blocking. While those have their own drawbacks such as potentially dropping messages that could not be queued for logging, they are the best available options currently for use in a reactive, non-blocking application.

Custom codecs

Applications can register custom codecs for supporting additional media types, or specific behaviors that are not supported by the default codecs.

Some configuration options expressed by developers are enforced on default codecs. Custom codecs might want to get a chance to align with those preferences, like enforcing buffering limits or logging sensitive data.

The following example shows how to do so for client-side requests:

Java

```
WebClient webClient = WebClient.builder()
        .codecs(configurer -> {
                CustomDecoder decoder = new CustomDecoder();
                configurer.customCodecs().registerWithDefaultConfig(decoder);
        })
        .build();
```

Kotlin

```
val webClient = WebClient.builder()
        .codecs({ configurer ->
                val decoder = CustomDecoder()
                configurer.customCodecs().registerWithDefaultConfig(decoder)
         })
        .build()
```

6.1.3. DispatcherHandler

Web MVC

Spring WebFlux, similarly to Spring MVC, is designed around the front controller pattern, where a central `WebHandler`, the `DispatcherHandler`, provides a shared algorithm for request processing, while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

`DispatcherHandler` discovers the delegate components it needs from Spring configuration. It is also designed to be a Spring bean itself and implements `ApplicationContextAware` for access to the context in which it runs. If `DispatcherHandler` is declared with a bean name of `webHandler`, it is, in turn, discovered by `WebHttpHandlerBuilder`, which puts together a request-processing chain, as described in `WebHandler` API.

Spring configuration in a WebFlux application typically contains:

- `DispatcherHandler` with the bean name `webHandler`
- `WebFilter` and `WebExceptionHandler` beans
- `DispatcherHandler` special beans
- Others

The configuration is given to `WebHttpHandlerBuilder` to build the processing chain, as the following example shows:

Java

```
ApplicationContext context = ...
HttpHandler handler = WebHttpHandlerBuilder.applicationContext(context).build();
```

Kotlin

```
val context: ApplicationContext = ...
val handler = WebHttpHandlerBuilder.applicationContext(context).build()
```

The resulting `HttpHandler` is ready for use with a server adapter.

Special Bean Types

Web MVC

The `DispatcherHandler` delegates to special beans to process requests and render the appropriate responses. By "special beans," we mean Spring-managed `Object` instances that implement WebFlux framework contracts. Those usually come with built-in contracts, but you can customize their properties, extend them, or replace them.

The following table lists the special beans detected by the `DispatcherHandler`. Note that there are also some other beans detected at a lower level (see Special bean types in the Web Handler API).

Bean type	Explanation
`HandlerMapping`	Map a request to a handler. The mapping is based on some criteria, the details of which vary by `HandlerMapping` implementation — annotated controllers, simple URL pattern mappings, and others. The main `HandlerMapping` implementations are `RequestMappingHandlerMapping` for `@RequestMapping` annotated methods, `RouterFunctionMapping` for functional endpoint routes, and `SimpleUrlHandlerMapping` for explicit registrations of URI path patterns and `WebHandler` instances.
`HandlerAdapter`	Help the `DispatcherHandler` to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a `HandlerAdapter` is to shield the `DispatcherHandler` from such details.
`HandlerResultHandler`	Process the result from the handler invocation and finalize the response. See Result Handling.

WebFlux Config

Web MVC

Applications can declare the infrastructure beans (listed under Web Handler API and `DispatcherHandler`) that are required to process requests. However, in most cases, the WebFlux Config is the best starting point. It declares the required beans and provides a higher-level configuration callback API to customize it.

Spring Boot relies on the WebFlux config to configure Spring WebFlux and also provides many extra convenient options.

Processing

Web MVC

`DispatcherHandler` processes requests as follows:

- Each `HandlerMapping` is asked to find a matching handler, and the first match is used.
- If a handler is found, it is run through an appropriate `HandlerAdapter`, which exposes the return value from the execution as `HandlerResult`.
- The `HandlerResult` is given to an appropriate `HandlerResultHandler` to complete processing by writing to the response directly or by using a view to render.

Result Handling

The return value from the invocation of a handler, through a `HandlerAdapter`, is wrapped as a `HandlerResult`, along with some additional context, and passed to the first `HandlerResultHandler` that claims support for it. The following table shows the available `HandlerResultHandler` implementations, all of which are declared in the WebFlux Config:

Result Handler Type	Return Values	Default Order
`ResponseEntityResultHandler`	`ResponseEntity`, typically from `@Controller` instances.	0
`ServerResponseResultHandler`	`ServerResponse`, typically from functional endpoints.	0
`ResponseBodyResultHandler`	Handle return values from `@ResponseBody` methods or `@RestController` classes.	100
`ViewResolutionResultHandler`	`CharSequence`, `View`, Model, `Map`, Rendering, or any other `Object` is treated as a model attribute. See also View Resolution.	`Integer.MAX_VALUE`

Exceptions

Web MVC

`HandlerAdapter` implementations can handle internally exceptions from invoking a request handler, such as a controller method. However, an exception may be deferred if the request handler returns an asynchronous value.

A `HandlerAdapter` may expose its exception handling mechanism as a `DispatchExceptionHandler` set on the `HandlerResult` it returns. When that's set, `DispatcherHandler` will also apply it to the handling of the result.

A `HandlerAdapter` may also choose to implement `DispatchExceptionHandler`. Inn that case

`DispatcherHandler` will apply it to exceptions that arise before a handler is mapped, e.g. during handler mapping, or earlier, e.g. in a `WebFilter`.

See also Exceptions in the "Annotated Controller" section or Exceptions in the WebHandler API section.

View Resolution

Web MVC

View resolution enables rendering to a browser with an HTML template and a model without tying you to a specific view technology. In Spring WebFlux, view resolution is supported through a dedicated HandlerResultHandler that uses `ViewResolver` instances to map a String (representing a logical view name) to a `View` instance. The `View` is then used to render the response.

Handling

Web MVC

The `HandlerResult` passed into `ViewResolutionResultHandler` contains the return value from the handler and the model that contains attributes added during request handling. The return value is processed as one of the following:

- `String`, `CharSequence`: A logical view name to be resolved to a `View` through the list of configured `ViewResolver` implementations.
- `void`: Select a default view name based on the request path, minus the leading and trailing slash, and resolve it to a `View`. The same also happens when a view name was not provided (for example, model attribute was returned) or an async return value (for example, `Mono` completed empty).
- Rendering: API for view resolution scenarios. Explore the options in your IDE with code completion.
- `Model`, `Map`: Extra model attributes to be added to the model for the request.
- Any other: Any other return value (except for simple types, as determined by BeanUtils#isSimpleProperty) is treated as a model attribute to be added to the model. The attribute name is derived from the class name by using conventions, unless a handler method `@ModelAttribute` annotation is present.

The model can contain asynchronous, reactive types (for example, from Reactor or RxJava). Prior to rendering, `AbstractView` resolves such model attributes into concrete values and updates the model. Single-value reactive types are resolved to a single value or no value (if empty), while multi-value reactive types (for example, `Flux<T>`) are collected and resolved to `List<T>`.

To configure view resolution is as simple as adding a `ViewResolutionResultHandler` bean to your Spring configuration. WebFlux Config provides a dedicated configuration API for view resolution.

See View Technologies for more on the view technologies integrated with Spring WebFlux.

Redirecting

Web MVC

The special `redirect:` prefix in a view name lets you perform a redirect. The `UrlBasedViewResolver` (and sub-classes) recognize this as an instruction that a redirect is needed. The rest of the view name is the redirect URL.

The net effect is the same as if the controller had returned a `RedirectView` or `Rendering.redirectTo("abc").build()`, but now the controller itself can operate in terms of logical view names. A view name such as `redirect:/some/resource` is relative to the current application, while a view name such as `redirect:https://example.com/arbitrary/path` redirects to an absolute URL.

Content Negotiation

Web MVC

`ViewResolutionResultHandler` supports content negotiation. It compares the request media types with the media types supported by each selected `View`. The first `View` that supports the requested media type(s) is used.

In order to support media types such as JSON and XML, Spring WebFlux provides `HttpMessageWriterView`, which is a special `View` that renders through an HttpMessageWriter. Typically, you would configure these as default views through the WebFlux Configuration. Default views are always selected and used if they match the requested media type.

6.1.4. Annotated Controllers

Web MVC

Spring WebFlux provides an annotation-based programming model, where `@Controller` and `@RestController` components use annotations to express request mappings, request input, handle exceptions, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.

The following listing shows a basic example:

Java

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public String handle() {
        return "Hello WebFlux";
    }
}
```

Kotlin

```
@RestController
class HelloController {

    @GetMapping("/hello")
    fun handle() = "Hello WebFlux"
}
```

In the preceding example, the method returns a `String` to be written to the response body.

`@Controller`

Web MVC

You can define controller beans by using a standard Spring bean definition. The `@Controller` stereotype allows for auto-detection and is aligned with Spring general support for detecting `@Component` classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

To enable auto-detection of such `@Controller` beans, you can add component scanning to your Java configuration, as the following example shows:

Java

```
@Configuration
@ComponentScan("org.example.web") ①
public class WebConfig {

    // ...
}
```

① Scan the `org.example.web` package.

Kotlin

```
@Configuration
@ComponentScan("org.example.web") ①
class WebConfig {

    // ...
}
```

① Scan the `org.example.web` package.

`@RestController` is a composed annotation that is itself meta-annotated with `@Controller` and `@ResponseBody`, indicating a controller whose every method inherits the type-level `@ResponseBody` annotation and, therefore, writes directly to the response body versus view resolution and rendering with an HTML template.

AOP Proxies

Web MVC

In some cases, you may need to decorate a controller with an AOP proxy at runtime. One example is if you choose to have `@Transactional` annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is automatically the case with such annotations directly on the controller.

If the controller implements an interface, and needs AOP proxying, you may need to explicitly configure class-based proxying. For example, with `@EnableTransactionManagement` you can change to `@EnableTransactionManagement(proxyTargetClass = true)`, and with `<tx:annotation-driven/>` you can change to `<tx:annotation-driven proxy-target-class="true"/>`.

Keep in mind that as of 6.0, with interface proxying, Spring WebFlux no longer detects controllers based solely on a type-level `@RequestMapping` annotation on the interface. Please, enable class based proxying, or otherwise the interface must also have an `@Controller` annotation.

Request Mapping

Web MVC

The `@RequestMapping` annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

There are also HTTP method specific shortcut variants of `@RequestMapping`:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

The preceding annotations are Custom Annotations that are provided because, arguably, most controller methods should be mapped to a specific HTTP method versus using `@RequestMapping`, which, by default, matches to all HTTP methods. At the same time, a `@RequestMapping` is still needed at the class level to express shared mappings.

The following example uses type and method level mappings:

Java

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    public Person getPerson(@PathVariable Long id) {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public void add(@RequestBody Person person) {
        // ...
    }
}
```

Kotlin

```
@RestController
@RequestMapping("/persons")
class PersonController {

    @GetMapping("/{id}")
    fun getPerson(@PathVariable id: Long): Person {
        // ...
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    fun add(@RequestBody person: Person) {
        // ...
    }
}
```

URI Patterns

Web MVC

You can map requests by using glob patterns and wildcards:

Pattern	Description	Example
`?`	Matches one character	`"/pages/t?st.html"` matches `"/pages/test.html"` and `"/pages/t3st.html"`

Pattern	Description	Example

`*`	Matches zero or more characters within a path segment	`"/resources/*.png"` matches `"/resources/file.png"`
		
		`"/projects/*/versions"` matches `"/projects/spring/versions"` but does not match `"/projects/spring/boot/versions"`

`**`	Matches zero or more path segments until the end of the path	`"/resources/**"` matches `"/resources/file.png"` and `"/resources/images/file.png"` `"/resources/**/file.png"` is invalid as `**` is only allowed at the end of the path.
`{name}`	Matches a path segment and captures it as a variable named "name"	`"/projects/{project}/versions"` matches `"/projects/spring/versions"` and captures `project=spring`
`{name:[a-z]+}`	Matches the regexp `"[a-z]+"` as a path variable named "name"	`"/projects/{project:[a-z]+}/versions"` matches `"/projects/spring/versions"` but not `"/projects/spring1/versions"`
`{*path}`	Matches zero or more path segments until the end of the path and captures it as a variable named "path"	`"/resources/{*file}"` matches `"/resources/images/file.png"` and captures `file=/images/file.png`

Captured URI variables can be accessed with `@PathVariable`, as the following example shows:

Java

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
    // ...
}
```

Kotlin

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
    // ...
}
```

You can declare URI variables at the class and method levels, as the following example shows:

Java

```
@Controller
@RequestMapping("/owners/{ownerId}") ①
public class OwnerController {

    @GetMapping("/pets/{petId}") ②
    public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
        // ...
    }
}
```

① Class-level URI mapping.

② Method-level URI mapping.

Kotlin

```
@Controller
@RequestMapping("/owners/{ownerId}") ①
class OwnerController {

    @GetMapping("/pets/{petId}") ②
    fun findPet(@PathVariable ownerId: Long, @PathVariable petId: Long): Pet {
        // ...
    }
}
```

① Class-level URI mapping.

② Method-level URI mapping.

URI variables are automatically converted to the appropriate type or a `TypeMismatchException` is raised. Simple types (`int`, `long`, `Date`, and so on) are supported by default and you can register support for any other data type. See Type Conversion and `DataBinder`.

URI variables can be named explicitly (for example, `@PathVariable("customId")`), but you can leave that detail out if the names are the same and you compile your code with the `-parameters` compiler flag.

The syntax `{*varName}` declares a URI variable that matches zero or more remaining path segments. For example `/resources/{*path}` matches all files under `/resources/`, and the `"path"` variable captures the complete path under `/resources`.

The syntax `{varName:regex}` declares a URI variable with a regular expression that has the syntax: `{varName:regex}`. For example, given a URL of `/spring-web-3.0.5.jar`, the following method extracts the name, version, and file extension:

Java

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String ext) {
    // ...
}
```

Kotlin

```
@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")
fun handle(@PathVariable version: String, @PathVariable ext: String) {
    // ...
}
```

URI path patterns can also have embedded `${…}` placeholders that are resolved on startup through `PropertySourcesPlaceholderConfigurer` against local, system, environment, and other property sources. You can use this to, for example, parameterize a base URL based on some external configuration.

Spring WebFlux uses `PathPattern` and the `PathPatternParser` for URI path matching support. Both classes are located in `spring-web` and are expressly designed for use with HTTP URL paths in web applications where a large number of URI path patterns are matched at runtime.

Spring WebFlux does not support suffix pattern matching — unlike Spring MVC, where a mapping such as `/person` also matches to `/person.*`. For URL-based content negotiation, if needed, we recommend using a query parameter, which is simpler, more explicit, and less vulnerable to URL path based exploits.

Pattern Comparison

Web MVC

When multiple patterns match a URL, they must be compared to find the best match. This is done with `PathPattern.SPECIFICITY_COMPARATOR`, which looks for patterns that are more specific.

For every pattern, a score is computed, based on the number of URI variables and wildcards, where a URI variable scores lower than a wildcard. A pattern with a lower total score wins. If two patterns have the same score, the longer is chosen.

Catch-all patterns (for example, `**`, `{*varName}`) are excluded from the scoring and are always sorted last instead. If two patterns are both catch-all, the longer is chosen.

Consumable Media Types

Web MVC

You can narrow the request mapping based on the `Content-Type` of the request, as the following example shows:

Java

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
    // ...
}
```

Kotlin

```
@PostMapping("/pets", consumes = ["application/json"])
fun addPet(@RequestBody pet: Pet) {
    // ...
}
```

The consumes attribute also supports negation expressions — for example, `!text/plain` means any content type other than `text/plain`.

You can declare a shared `consumes` attribute at the class level. Unlike most other request mapping attributes, however, when used at the class level, a method-level `consumes` attribute overrides rather than extends the class-level declaration.

`MediaType` provides constants for commonly used media types — for example, `APPLICATION_JSON_VALUE` and `APPLICATION_XML_VALUE`.

Producible Media Types

Web MVC

You can narrow the request mapping based on the `Accept` request header and the list of content types that a controller method produces, as the following example shows:

Java

```
@GetMapping(path = "/pets/{petId}", produces = "application/json")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
    // ...
}
```

Kotlin

```
@GetMapping("/pets/{petId}", produces = ["application/json"])
@ResponseBody
fun getPet(@PathVariable String petId): Pet {
    // ...
}
```

The media type can specify a character set. Negated expressions are supported — for example,

`!text/plain` means any content type other than `text/plain`.

You can declare a shared `produces` attribute at the class level. Unlike most other request mapping attributes, however, when used at the class level, a method-level `produces` attribute overrides rather than extend the class level declaration.

`MediaType` provides constants for commonly used media types — e.g. `APPLICATION_JSON_VALUE`, `APPLICATION_XML_VALUE`.

Parameters and Headers

Web MVC

You can narrow request mappings based on query parameter conditions. You can test for the presence of a query parameter (`myParam`), for its absence (`!myParam`), or for a specific value (`myParam=myValue`). The following examples tests for a parameter with a value:

Java

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue") ①
public void findPet(@PathVariable String petId) {
    // ...
}
```

① Check that `myParam` equals `myValue`.

Kotlin

```
@GetMapping("/pets/{petId}", params = ["myParam=myValue"]) ①
fun findPet(@PathVariable petId: String) {
    // ...
}
```

① Check that `myParam` equals `myValue`.

You can also use the same with request header conditions, as the following example shows:

Java

```
@GetMapping(path = "/pets", headers = "myHeader=myValue") ①
public void findPet(@PathVariable String petId) {
    // ...
}
```

① Check that `myHeader` equals `myValue`.

Kotlin

```
@GetMapping("/pets", headers = ["myHeader=myValue"]) ①
fun findPet(@PathVariable petId: String) {
    // ...
}
```

① Check that `myHeader` equals `myValue`.

HTTP HEAD, OPTIONS

Web MVC

`@GetMapping` and `@RequestMapping(method=HttpMethod.GET)` support HTTP HEAD transparently for request mapping purposes. Controller methods need not change. A response wrapper, applied in the `HttpHandler` server adapter, ensures a `Content-Length` header is set to the number of bytes written without actually writing to the response.

By default, HTTP OPTIONS is handled by setting the `Allow` response header to the list of HTTP methods listed in all `@RequestMapping` methods with matching URL patterns.

For a `@RequestMapping` without HTTP method declarations, the `Allow` header is set to `GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS`. Controller methods should always declare the supported HTTP methods (for example, by using the HTTP method specific variants — `@GetMapping`, `@PostMapping`, and others).

You can explicitly map a `@RequestMapping` method to HTTP HEAD and HTTP OPTIONS, but that is not necessary in the common case.

Custom Annotations

Web MVC

Spring WebFlux supports the use of composed annotations for request mapping. Those are annotations that are themselves meta-annotated with `@RequestMapping` and composed to redeclare a subset (or all) of the `@RequestMapping` attributes with a narrower, more specific purpose.

`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, and `@PatchMapping` are examples of composed annotations. They are provided, because, arguably, most controller methods should be mapped to a specific HTTP method versus using `@RequestMapping`, which, by default, matches to all HTTP methods. If you need an example of composed annotations, look at how those are declared.

Spring WebFlux also supports custom request mapping attributes with custom request matching logic. This is a more advanced option that requires sub-classing `RequestMappingHandlerMapping` and overriding the `getCustomMethodCondition` method, where you can check the custom attribute and return your own `RequestCondition`.

Explicit Registrations

Web MVC

You can programmatically register Handler methods, which can be used for dynamic registrations

or for advanced cases, such as different instances of the same handler under different URLs. The following example shows how to do so:

Java

```
@Configuration
public class MyConfig {

    @Autowired
    public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler
handler) ①
            throws NoSuchMethodException {

        RequestMappingInfo info = RequestMappingInfo
                .paths("/user/{id}").methods(RequestMethod.GET).build(); ②

        Method method = UserHandler.class.getMethod("getUser", Long.class); ③

        mapping.registerMapping(info, handler, method); ④
    }

}
```

① Inject target handlers and the handler mapping for controllers.

② Prepare the request mapping metadata.

③ Get the handler method.

④ Add the registration.

Kotlin

```
@Configuration
class MyConfig {

    @Autowired
    fun setHandlerMapping(mapping: RequestMappingHandlerMapping, handler: UserHandler)
{ ①

        val info =
RequestMappingInfo.paths("/user/{id}").methods(RequestMethod.GET).build() ②

        val method = UserHandler::class.java.getMethod("getUser", Long::class.java) ③

        mapping.registerMapping(info, handler, method) ④
    }
}
```

① Inject target handlers and the handler mapping for controllers.

② Prepare the request mapping metadata.

③ Get the handler method.

④ Add the registration.

Handler Methods

Web MVC

`@RequestMapping` handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

Method Arguments

Web MVC

The following table shows the supported controller method arguments.

Reactive types (Reactor, RxJava, or other) are supported on arguments that require blocking I/O (for example, reading the request body) to be resolved. This is marked in the Description column. Reactive types are not expected on arguments that do not require blocking.

JDK 1.8's `java.util.Optional` is supported as a method argument in combination with annotations that have a `required` attribute (for example, `@RequestParam`, `@RequestHeader`, and others) and is equivalent to `required=false`.

Controller method argument	Description
`ServerWebExchange`	Access to the full `ServerWebExchange` — container for the HTTP request and response, request and session attributes, `checkNotModified` methods, and others.
`ServerHttpRequest`, `ServerHttpResponse`	Access to the HTTP request or response.
`WebSession`	Access to the session. This does not force the start of a new session unless attributes are added. Supports reactive types.
`java.security.Principal`	The currently authenticated user — possibly a specific `Principal` implementation class if known. Supports reactive types.
`org.springframework.http.HttpMethod`	The HTTP method of the request.
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available — in effect, the configured `LocaleResolver`/`LocaleContextResolver`.
`java.util.TimeZone` + `java.time.ZoneId`	The time zone associated with the current request, as determined by a `LocaleContextResolver`.
`@PathVariable`	For access to URI template variables. See URI Patterns.
`@MatrixVariable`	For access to name-value pairs in URI path segments. See Matrix Variables.

Controller method argument	Description
`@RequestParam`	For access to query parameters. Parameter values are converted to the declared method argument type. See `@RequestParam`. Note that use of `@RequestParam` is optional — for example, to set its attributes. See “Any other argument” later in this table.
`@RequestHeader`	For access to request headers. Header values are converted to the declared method argument type. See `@RequestHeader`.
`@CookieValue`	For access to cookies. Cookie values are converted to the declared method argument type. See `@CookieValue`.
`@RequestBody`	For access to the HTTP request body. Body content is converted to the declared method argument type by using `HttpMessageReader` instances. Supports reactive types. See `@RequestBody`.
`HttpEntity<B>`	For access to request headers and body. The body is converted with `HttpMessageReader` instances. Supports reactive types. See `HttpEntity`.
`@RequestPart`	For access to a part in a `multipart/form-data` request. Supports reactive types. See Multipart Content and Multipart Data.
`java.util.Map`, `org.springframework.ui.Model`, and `org.springframework.ui.ModelMap`.	For access to the model that is used in HTML controllers and is exposed to templates as part of view rendering.
`@ModelAttribute`	For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See `@ModelAttribute` as well as `Model` and `DataBinder`. Note that use of `@ModelAttribute` is optional — for example, to set its attributes. See “Any other argument” later in this table.
`Errors`, `BindingResult`	For access to errors from validation and data binding for a command object, i.e. a `@ModelAttribute` argument. An `Errors`, or `BindingResult` argument must be declared immediately after the validated method argument.
`SessionStatus` + class-level `@SessionAttributes`	For marking form processing complete, which triggers cleanup of session attributes declared through a class-level `@SessionAttributes` annotation. See `@SessionAttributes` for more details.
`UriComponentsBuilder`	For preparing a URL relative to the current request’s host, port, scheme, and context path. See URI Links.
`@SessionAttribute`	For access to any session attribute — in contrast to model attributes stored in the session as a result of a class-level `@SessionAttributes` declaration. See `@SessionAttribute` for more details.

Controller method argument	Description
`@RequestAttribute`	For access to request attributes. See `@RequestAttribute` for more details.
Any other argument	If a method argument is not matched to any of the above, it is, by default, resolved as a `@RequestParam` if it is a simple type, as determined by BeanUtils#isSimpleProperty, or as a `@ModelAttribute`, otherwise.

Return Values

Web MVC

The following table shows the supported controller method return values. Note that reactive types from libraries such as Reactor, RxJava, or other are generally supported for all return values.

Controller method return value	Description
`@ResponseBody`	The return value is encoded through `HttpMessageWriter` instances and written to the response. See `@ResponseBody`.
`HttpEntity<B>`, `ResponseEntity<B>`	The return value specifies the full response, including HTTP headers, and the body is encoded through `HttpMessageWriter` instances and written to the response. See `ResponseEntity`.
`HttpHeaders`	For returning a response with headers and no body.
`ErrorResponse`	To render an RFC 7807 error response with details in the body, see Error Responses
`ProblemDetail`	To render an RFC 7807 error response with details in the body, see Error Responses
`String`	A view name to be resolved with `ViewResolver` instances and used together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`View`	A `View` instance to use for rendering together with the implicit model — determined through command objects and `@ModelAttribute` methods. The handler method can also programmatically enrich the model by declaring a `Model` argument (described earlier).
`java.util.Map`, `org.springframework.ui.Model`	Attributes to be added to the implicit model, with the view name implicitly determined based on the request path.
`@ModelAttribute`	An attribute to be added to the model, with the view name implicitly determined based on the request path. Note that `@ModelAttribute` is optional. See “Any other return value” later in this table.

Controller method return value	Description
`Rendering`	An API for model and view rendering scenarios.
`void`	A method with a `void`, possibly asynchronous (for example, `Mono<Void>`), return type (or a `null` return value) is considered to have fully handled the response if it also has a `ServerHttpResponse`, a `ServerWebExchange` argument, or an `@ResponseStatus` annotation. The same is also true if the controller has made a positive ETag or `lastModified` timestamp check. See Controllers for details. If none of the above is true, a `void` return type can also indicate "no response body" for REST controllers or default view name selection for HTML controllers.
`Flux<ServerSentEvent>`, `Observable<ServerSentEvent>`, or other reactive type	Emit server-sent events. The `ServerSentEvent` wrapper can be omitted when only data needs to be written (however, `text/event-stream` must be requested or declared in the mapping through the `produces` attribute).
Other return values	If a return value remains unresolved in any other way, it is treated as a model attribute, unless it is a simple type as determined by BeanUtils#isSimpleProperty, in which case it remains unresolved.

Type Conversion

Web MVC

Some annotated controller method arguments that represent String-based request input (for example, `@RequestParam`, `@RequestHeader`, `@PathVariable`, `@MatrixVariable`, and `@CookieValue`) can require type conversion if the argument is declared as something other than `String`.

For such cases, type conversion is automatically applied based on the configured converters. By default, simple types (such as `int`, `long`, `Date`, and others) are supported. Type conversion can be customized through a `WebDataBinder` (see `DataBinder`) or by registering `Formatters` with the `FormattingConversionService` (see Spring Field Formatting).

A practical issue in type conversion is the treatment of an empty String source value. Such a value is treated as missing if it becomes `null` as a result of type conversion. This can be the case for `Long`, `UUID`, and other target types. If you want to allow `null` to be injected, either use the `required` flag on the argument annotation, or declare the argument as `@Nullable`.

Matrix Variables

Web MVC

RFC 3986 discusses name-value pairs in path segments. In Spring WebFlux, we refer to those as "matrix variables" based on an "old post" by Tim Berners-Lee, but they can be also be referred to as URI path parameters.

Matrix variables can appear in any path segment, with each variable separated by a semicolon and multiple values separated by commas — for example, `"/cars;color=red,green;year=2012"`. Multiple values can also be specified through repeated variable names — for example, `"color=red;color=green;color=blue"`.

Unlike Spring MVC, in WebFlux, the presence or absence of matrix variables in a URL does not affect request mappings. In other words, you are not required to use a URI variable to mask variable content. That said, if you want to access matrix variables from a controller method, you need to add a URI variable to the path segment where matrix variables are expected. The following example shows how to do so:

Java

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11
}
```

Kotlin

```
// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")
fun findPet(@PathVariable petId: String, @MatrixVariable q: Int) {

    // petId == 42
    // q == 11
}
```

Given that all path segments can contain matrix variables, you may sometimes need to disambiguate which path variable the matrix variable is expected to be in, as the following example shows:

Java

```
// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable(name="q", pathVar="ownerId") int q1,
        @MatrixVariable(name="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22
}
```

Kotlin

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(
        @MatrixVariable(name = "q", pathVar = "ownerId") q1: Int,
        @MatrixVariable(name = "q", pathVar = "petId") q2: Int) {

    // q1 == 11
    // q2 == 22
}
```

You can define a matrix variable may be defined as optional and specify a default value as the following example shows:

Java

```
// GET /pets/42

@GetMapping("/pets/{petId}")
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1
}
```

Kotlin

```
// GET /pets/42

@GetMapping("/pets/{petId}")
fun findPet(@MatrixVariable(required = false, defaultValue = "1") q: Int) {

    // q == 1
}
```

To get all matrix variables, use a `MultiValueMap`, as the following example shows:

Java

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
public void findPet(
        @MatrixVariable MultiValueMap<String, String> matrixVars,
        @MatrixVariable(pathVar="petId") MultiValueMap<String, String> petMatrixVars)
{

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 22, "s" : 23]
}
```

Kotlin

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")
fun findPet(
        @MatrixVariable matrixVars: MultiValueMap<String, String>,
        @MatrixVariable(pathVar="petId") petMatrixVars: MultiValueMap<String, String>)
{

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 22, "s" : 23]
}
```

`@RequestParam`

Web MVC

You can use the `@RequestParam` annotation to bind query parameters to a method argument in a controller. The following code snippet shows the usage:

Java

```
@Controller
@RequestMapping("/pets")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, Model model) { ①
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...
}
```

① Using `@RequestParam`.

Kotlin

```
import org.springframework.ui.set

@Controller
@RequestMapping("/pets")
class EditPetForm {

    // ...

    @GetMapping
    fun setupForm(@RequestParam("petId") petId: Int, model: Model): String { ①
        val pet = clinic.loadPet(petId)
        model["pet"] = pet
        return "petForm"
    }

    // ...
}
```

① Using `@RequestParam`.

The Servlet API "request parameter" concept conflates query parameters, form data, and multiparts into one. However, in WebFlux, each is accessed individually through `ServerWebExchange`. While `@RequestParam` binds to query parameters only, you can use data binding to apply query parameters, form data, and multiparts to a command object.

Method parameters that use the `@RequestParam` annotation are required by default, but you can specify that a method parameter is optional by setting the required flag of a `@RequestParam` to `false`

or by declaring the argument with a `java.util.Optional` wrapper.

Type conversion is applied automatically if the target method parameter type is not `String`. See Type Conversion.

When a `@RequestParam` annotation is declared on a `Map<String, String>` or `MultiValueMap<String, String>` argument, the map is populated with all query parameters.

Note that use of `@RequestParam` is optional — for example, to set its attributes. By default, any argument that is a simple value type (as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver is treated as if it were annotated with `@RequestParam`.

`@RequestHeader`

Web MVC

You can use the `@RequestHeader` annotation to bind a request header to a method argument in a controller.

The following example shows a request with headers:

```
Host                    localhost:8080
Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language         fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding         gzip,deflate
Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive              300
```

The following example gets the value of the `Accept-Encoding` and `Keep-Alive` headers:

Java

```
@GetMapping("/demo")
public void handle(
        @RequestHeader("Accept-Encoding") String encoding, ①
        @RequestHeader("Keep-Alive") long keepAlive) { ②
    //...
}
```

① Get the value of the `Accept-Encoding` header.

② Get the value of the `Keep-Alive` header.

Kotlin

```
@GetMapping("/demo")
fun handle(
        @RequestHeader("Accept-Encoding") encoding: String, ①
        @RequestHeader("Keep-Alive") keepAlive: Long) { ②
    //...
}
```

① Get the value of the `Accept-Encoding` header.

② Get the value of the `Keep-Alive` header.

Type conversion is applied automatically if the target method parameter type is not `String`. See Type Conversion.

When a `@RequestHeader` annotation is used on a `Map<String, String>`, `MultiValueMap<String, String>`, or `HttpHeaders` argument, the map is populated with all header values.

Built-in support is available for converting a comma-separated string into an array or collection of strings or other types known to the type conversion system. For example, a method parameter annotated with `@RequestHeader("Accept")` may be of type `String` but also of `String[]` or `List<String>`.

`@CookieValue`

Web MVC

You can use the `@CookieValue` annotation to bind the value of an HTTP cookie to a method argument in a controller.

The following example shows a request with a cookie:

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

The following code sample demonstrates how to get the cookie value:

Java

```
@GetMapping("/demo")
public void handle(@CookieValue("JSESSIONID") String cookie) { ①
    //...
}
```

① Get the cookie value.

Kotlin

```
@GetMapping("/demo")
fun handle(@CookieValue("JSESSIONID") cookie: String) { ①
    //...
}
```

① Get the cookie value.

Type conversion is applied automatically if the target method parameter type is not `String`. See Type Conversion.

`@ModelAttribute`

Web MVC

You can use the `@ModelAttribute` annotation on a method argument to access an attribute from the model or have it instantiated if not present. The model attribute is also overlaid with the values of query parameters and form fields whose names match to field names. This is referred to as data binding, and it saves you from having to deal with parsing and converting individual query parameters and form fields. The following example binds an instance of `Pet`:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) { } ①
```

① Bind an instance of `Pet`.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute pet: Pet): String { } ①
```

① Bind an instance of `Pet`.

The `Pet` instance in the preceding example is resolved as follows:

- From the model if already added through `Model`.
- From the HTTP session through `@SessionAttributes`.
- From the invocation of a default constructor.
- From the invocation of a "primary constructor" with arguments that match query parameters or form fields. Argument names are determined through JavaBeans `@ConstructorProperties` or through runtime-retained parameter names in the bytecode.

After the model attribute instance is obtained, data binding is applied. The `WebExchangeDataBinder` class matches names of query parameters and form fields to field names on the target `Object`. Matching fields are populated after type conversion is applied where necessary. For more on data binding (and validation), see Validation. For more on customizing data binding, see `DataBinder`.

Data binding can result in errors. By default, a `WebExchangeBindException` is raised, but, to check for such errors in the controller method, you can add a `BindingResult` argument immediately next to the `@ModelAttribute`, as the following example shows:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) { ①
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

① Adding a `BindingResult`.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@ModelAttribute("pet") pet: Pet, result: BindingResult): String { ①
    if (result.hasErrors()) {
        return "petForm"
    }
    // ...
}
```

① Adding a `BindingResult`.

You can automatically apply validation after data binding by adding the `jakarta.validation.Valid` annotation or Spring's `@Validated` annotation (see also Bean Validation and Spring validation). The following example uses the `@Valid` annotation:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult
result) { ①
    if (result.hasErrors()) {
        return "petForm";
    }
    // ...
}
```

① Using `@Valid` on a model attribute argument.

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") pet: Pet, result: BindingResult):
String { ①
    if (result.hasErrors()) {
        return "petForm"
    }
    // ...
}
```

① Using `@Valid` on a model attribute argument.

Spring WebFlux, unlike Spring MVC, supports reactive types in the model — for example, `Mono<Account>` or `io.reactivex.Single<Account>`. You can declare a `@ModelAttribute` argument with or without a reactive type wrapper, and it will be resolved accordingly, to the actual value if necessary. However, note that, to use a `BindingResult` argument, you must declare the `@ModelAttribute` argument before it without a reactive type wrapper, as shown earlier. Alternatively, you can handle any errors through the reactive type, as the following example shows:

Java

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public Mono<String> processSubmit(@Valid @ModelAttribute("pet") Mono<Pet> petMono) {
    return petMono
        .flatMap(pet -> {
            // ...
        })
        .onErrorResume(ex -> {
            // ...
        });
}
```

Kotlin

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
fun processSubmit(@Valid @ModelAttribute("pet") petMono: Mono<Pet>): Mono<String> {
    return petMono
            .flatMap { pet ->
                // ...
            }
            .onErrorResume{ ex ->
                // ...
            }
}
```

Note that use of `@ModelAttribute` is optional — for example, to set its attributes. By default, any argument that is not a simple value type(as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver is treated as if it were annotated with `@ModelAttribute`.

`@SessionAttributes`

Web MVC

`@SessionAttributes` is used to store model attributes in the `WebSession` between requests. It is a type-level annotation that declares session attributes used by a specific controller. This typically lists the names of model attributes or types of model attributes that should be transparently stored in the session for subsequent requests to access.

Consider the following example:

Java

```
@Controller
@SessionAttributes("pet") ①
public class EditPetForm {
    // ...
}
```

① Using the `@SessionAttributes` annotation.

Kotlin

```
@Controller
@SessionAttributes("pet") ①
class EditPetForm {
    // ...
}
```

① Using the `@SessionAttributes` annotation.

On the first request, when a model attribute with the name, `pet`, is added to the model, it is automatically promoted to and saved in the `WebSession`. It remains there until another controller method uses a `SessionStatus` method argument to clear the storage, as the following example shows:

Java

```
@Controller
@SessionAttributes("pet") ①
public class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    public String handle(Pet pet, BindingResult errors, SessionStatus status) { ②
        if (errors.hasErrors()) {
            // ...
        }
            status.setComplete();
            // ...
        }
    }
}
```

① Using the `@SessionAttributes` annotation.

② Using a `SessionStatus` variable.

Kotlin

```
@Controller
@SessionAttributes("pet") ①
class EditPetForm {

    // ...

    @PostMapping("/pets/{id}")
    fun handle(pet: Pet, errors: BindingResult, status: SessionStatus): String { ②
        if (errors.hasErrors()) {
            // ...
        }
        status.setComplete()
        // ...
    }
}
```

① Using the `@SessionAttributes` annotation.

② Using a `SessionStatus` variable.

`@SessionAttribute`

Web MVC

If you need access to pre-existing session attributes that are managed globally (that is, outside the controller — for example, by a filter) and may or may not be present, you can use the `@SessionAttribute` annotation on a method parameter, as the following example shows:

Java

```
@GetMapping("/")
public String handle(@SessionAttribute User user) { ①
    // ...
}
```

① Using `@SessionAttribute`.

Kotlin

```
@GetMapping("/")
fun handle(@SessionAttribute user: User): String { ①
    // ...
}
```

① Using `@SessionAttribute`.

For use cases that require adding or removing session attributes, consider injecting `WebSession` into the controller method.

For temporary storage of model attributes in the session as part of a controller workflow, consider using `SessionAttributes`, as described in `@SessionAttributes`.

`@RequestAttribute`

Web MVC

Similarly to `@SessionAttribute`, you can use the `@RequestAttribute` annotation to access pre-existing request attributes created earlier (for example, by a `WebFilter`), as the following example shows:

Java

```
@GetMapping("/")
public String handle(@RequestAttribute Client client) { ①
    // ...
}
```

① Using `@RequestAttribute`.

Kotlin

```
@GetMapping("/")
fun handle(@RequestAttribute client: Client): String { ①
    // ...
}
```

① Using `@RequestAttribute`.

Multipart Content

Web MVC

As explained in Multipart Data, `ServerWebExchange` provides access to multipart content. The best way to handle a file upload form (for example, from a browser) in a controller is through data binding to a command object, as the following example shows:

Java

```
class MyForm {

    private String name;

    private MultipartFile file;

    // ...

}

@Controller
public class FileUploadController {

    @PostMapping("/form")
    public String handleFormUpload(MyForm form, BindingResult errors) {
        // ...
    }

}
```

Kotlin

```
class MyForm(
        val name: String,
        val file: MultipartFile)

@Controller
class FileUploadController {

    @PostMapping("/form")
    fun handleFormUpload(form: MyForm, errors: BindingResult): String {
        // ...
    }

}
```

You can also submit multipart requests from non-browser clients in a RESTful service scenario. The following example uses a file along with JSON:

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
    "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

You can access individual parts with `@RequestPart`, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") Part metadata, ①
        @RequestPart("file-data") FilePart file) { ②
    // ...
}
```

① Using `@RequestPart` to get the metadata.

② Using `@RequestPart` to get the file.

Kotlin

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") Part metadata, ①
        @RequestPart("file-data") FilePart file): String { ②
    // ...
}
```

① Using `@RequestPart` to get the metadata.

② Using `@RequestPart` to get the file.

To deserialize the raw part content (for example, to JSON — similar to `@RequestBody`), you can declare a concrete target `Object`, instead of `Part`, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestPart("meta-data") MetaData metadata) { ①
    // ...
}
```

① Using `@RequestPart` to get the metadata.

Kotlin

```
@PostMapping("/")
fun handle(@RequestPart("meta-data") metadata: MetaData): String { ①
    // ...
}
```

① Using `@RequestPart` to get the metadata.

You can use `@RequestPart` in combination with `jakarta.validation.Valid` or Spring's `@Validated` annotation, which causes Standard Bean Validation to be applied. Validation errors lead to a `WebExchangeBindException` that results in a 400 (BAD_REQUEST) response. The exception contains a `BindingResult` with the error details and can also be handled in the controller method by declaring the argument with an async wrapper and then using error related operators:

Java

```
@PostMapping("/")
public String handle(@Valid @RequestPart("meta-data") Mono<MetaData> metadata) {
    // use one of the onError* operators...
}
```

Kotlin

```
@PostMapping("/")
fun handle(@Valid @RequestPart("meta-data") metadata: MetaData): String {
    // ...
}
```

To access all multipart data as a `MultiValueMap`, you can use `@RequestBody`, as the following example shows:

Java

```
@PostMapping("/")
public String handle(@RequestBody Mono<MultiValueMap<String, Part>> parts) { ①
    // ...
}
```

① Using `@RequestBody`.

Kotlin

```
@PostMapping("/")
fun handle(@RequestBody parts: MultiValueMap<String, Part>): String { ①
    // ...
}
```

① Using `@RequestBody`.

`PartEvent`

To access multipart data sequentially, in a streaming fashion, you can use `@RequestBody` with `Flux<PartEvent>` (or `Flow<PartEvent>` in Kotlin). Each part in a multipart HTTP message will produce at least one `PartEvent` containing both headers and a buffer with the contents of the part.

- Form fields will produce a **single** `FormPartEvent`, containing the value of the field.
- File uploads will produce **one or more** `FilePartEvent` objects, containing the filename used when uploading. If the file is large enough to be split across multiple buffers, the first `FilePartEvent` will be followed by subsequent events.

For example:

Java

```
@PostMapping("/")
public void handle(@RequestBody Flux<PartEvent> allPartsEvents) { ①
    allPartsEvents.windowUntil(PartEvent::isLast) ②
            .concatMap(p -> p.switchOnFirst((signal, partEvents) -> { ③
                if (signal.hasValue()) {
                    PartEvent event = signal.get();
                    if (event instanceof FormPartEvent formEvent) { ④
                        String value = formEvent.value();
                        // handle form field
                    }
                    else if (event instanceof FilePartEvent fileEvent) { ⑤
                        String filename = fileEvent.filename();
                        Flux<DataBuffer> contents =
partEvents.map(PartEvent::content); ⑥
                        // handle file upload
                    }
                    else {
                        return Mono.error(new RuntimeException("Unexpected event: " +
event));
                    }
                }
                else {
                    return partEvents; // either complete or error signal
                }
            }));
}
```

① Using `@RequestBody`.

② The final `PartEvent` for a particular part will have `isLast()` set to `true`, and can be followed by additional events belonging to subsequent parts. This makes the `isLast` property suitable as a predicate for the `Flux::windowUntil` operator, to split events from all parts into windows that each belong to a single part.

③ The `Flux::switchOnFirst` operator allows you to see whether you are handling a form field or file upload.

④ Handling the form field.

⑤ Handling the file upload.

⑥ The body contents must be completely consumed, relayed, or released to avoid memory leaks.

Kotlin

```
    @PostMapping("/")
    fun handle(@RequestBody allPartsEvents: Flux<PartEvent>) = { ①
      allPartsEvents.windowUntil(PartEvent::isLast) ②
          .concatMap {
              it.switchOnFirst { signal, partEvents -> ③
                  if (signal.hasValue()) {
                      val event = signal.get()
                      if (event is FormPartEvent) { ④
                          val value: String = event.value();
                          // handle form field
                      } else if (event is FilePartEvent) { ⑤
                          val filename: String = event.filename();
                          val contents: Flux<DataBuffer> =
partEvents.map(PartEvent::content); ⑥
                          // handle file upload
                      } else {
                          return Mono.error(RuntimeException("Unexpected event: " +
event));
                      }
                  } else {
                      return partEvents; // either complete or error signal
                  }
              }
          }
}
```

① Using `@RequestBody`.

② The final `PartEvent` for a particular part will have `isLast()` set to `true`, and can be followed by additional events belonging to subsequent parts. This makes the `isLast` property suitable as a predicate for the `Flux::windowUntil` operator, to split events from all parts into windows that each belong to a single part.

③ The `Flux::switchOnFirst` operator allows you to see whether you are handling a form field or file upload.

④ Handling the form field.

⑤ Handling the file upload.

⑥ The body contents must be completely consumed, relayed, or released to avoid memory leaks.

Received part events can also be relayed to another service by using the `WebClient`. See Multipart Data.

`@RequestBody`

Web MVC

You can use the `@RequestBody` annotation to have the request body read and deserialized into an `Object` through an HttpMessageReader. The following example uses a `@RequestBody` argument:

Java

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@RequestBody account: Account) {
    // ...
}
```

Unlike Spring MVC, in WebFlux, the `@RequestBody` method argument supports reactive types and fully non-blocking reading and (client-to-server) streaming.

Java

```
@PostMapping("/accounts")
public void handle(@RequestBody Mono<Account> account) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@RequestBody accounts: Flow<Account>) {
    // ...
}
```

You can use the HTTP message codecs option of the WebFlux Config to configure or customize message readers.

You can use `@RequestBody` in combination with `jakarta.validation.Valid` or Spring's `@Validated` annotation, which causes Standard Bean Validation to be applied. Validation errors cause a `WebExchangeBindException`, which results in a 400 (BAD_REQUEST) response. The exception contains a `BindingResult` with error details and can be handled in the controller method by declaring the argument with an async wrapper and then using error related operators:

Java

```
@PostMapping("/accounts")
public void handle(@Valid @RequestBody Mono<Account> account) {
    // use one of the onError* operators...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(@Valid @RequestBody account: Mono<Account>) {
    // ...
}
```

`HttpEntity`

Web MVC

`HttpEntity` is more or less identical to using `@RequestBody` but is based on a container object that exposes request headers and the body. The following example uses an `HttpEntity`:

Java

```
@PostMapping("/accounts")
public void handle(HttpEntity<Account> entity) {
    // ...
}
```

Kotlin

```
@PostMapping("/accounts")
fun handle(entity: HttpEntity<Account>) {
    // ...
}
```

`@ResponseBody`

Web MVC

You can use the `@ResponseBody` annotation on a method to have the return serialized to the response body through an HttpMessageWriter. The following example shows how to do so:

Java

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account handle() {
    // ...
}
```

Kotlin

```
@GetMapping("/accounts/{id}")
@ResponseBody
fun handle(): Account {
    // ...
}
```

`@ResponseBody` is also supported at the class level, in which case it is inherited by all controller methods. This is the effect of `@RestController`, which is nothing more than a meta-annotation marked with `@Controller` and `@ResponseBody`.

`@ResponseBody` supports reactive types, which means you can return Reactor or RxJava types and have the asynchronous values they produce rendered to the response. For additional details, see Streaming and JSON rendering.

You can combine `@ResponseBody` methods with JSON serialization views. See Jackson JSON for details.

You can use the HTTP message codecs option of the WebFlux Config to configure or customize message writing.

`ResponseEntity`

Web MVC

`ResponseEntity` is like `@ResponseBody` but with status and headers. For example:

Java

```
@GetMapping("/something")
public ResponseEntity<String> handle() {
    String body = ... ;
    String etag = ... ;
    return ResponseEntity.ok().eTag(etag).body(body);
}
```

Kotlin

```
@GetMapping("/something")
fun handle(): ResponseEntity<String> {
    val body: String = ...
    val etag: String = ...
    return ResponseEntity.ok().eTag(etag).build(body)
}
```

WebFlux supports using a single value reactive type to produce the `ResponseEntity` asynchronously, and/or single and multi-value reactive types for the body. This allows a variety of async responses with `ResponseEntity` as follows:

- `ResponseEntity<Mono<T>>` or `ResponseEntity<Flux<T>>` make the response status and headers known immediately while the body is provided asynchronously at a later point. Use `Mono` if the body consists of 0..1 values or `Flux` if it can produce multiple values.
- `Mono<ResponseEntity<T>>` provides all three — response status, headers, and body, asynchronously at a later point. This allows the response status and headers to vary depending on the outcome of asynchronous request handling.
- `Mono<ResponseEntity<Mono<T>>>` or `Mono<ResponseEntity<Flux<T>>>` are yet another possible, albeit less common alternative. They provide the response status and headers asynchronously first and then the response body, also asynchronously, second.

Jackson JSON

Spring offers support for the Jackson JSON library.

JSON Views

Web MVC

Spring WebFlux provides built-in support for Jackson's Serialization Views, which allows rendering only a subset of all fields in an `Object`. To use it with `@ResponseBody` or `ResponseEntity` controller methods, you can use Jackson's `@JsonView` annotation to activate a serialization view class, as the following example shows:

Java

```
@RestController
public class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView.class)
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }
}

public class User {

    public interface WithoutPasswordView {};
    public interface WithPasswordView extends WithoutPasswordView {};

    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    @JsonView(WithoutPasswordView.class)
    public String getUsername() {
        return this.username;
    }

    @JsonView(WithPasswordView.class)
    public String getPassword() {
        return this.password;
    }
}
```

Kotlin

```
@RestController
class UserController {

    @GetMapping("/user")
    @JsonView(User.WithoutPasswordView::class)
    fun getUser(): User {
        return User("eric", "7!jd#h23")
    }
}

class User(
        @JsonView(WithoutPasswordView::class) val username: String,
        @JsonView(WithPasswordView::class) val password: String
) {
    interface WithoutPasswordView
    interface WithPasswordView : WithoutPasswordView
}
```

`@JsonView` allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate multiple views.

`Model`

Web MVC

You can use the `@ModelAttribute` annotation:

- On a method argument in `@RequestMapping` methods to create or access an Object from the model and to bind it to the request through a `WebDataBinder`.
- As a method-level annotation in `@Controller` or `@ControllerAdvice` classes, helping to initialize the model prior to any `@RequestMapping` method invocation.
- On a `@RequestMapping` method to mark its return value as a model attribute.

This section discusses `@ModelAttribute` methods, or the second item from the preceding list. A controller can have any number of `@ModelAttribute` methods. All such methods are invoked before `@RequestMapping` methods in the same controller. A `@ModelAttribute` method can also be shared across controllers through `@ControllerAdvice`. See the section on Controller Advice for more details.

`@ModelAttribute` methods have flexible method signatures. They support many of the same arguments as `@RequestMapping` methods (except for `@ModelAttribute` itself and anything related to the request body).

The following example uses a `@ModelAttribute` method:

Java

```
@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountRepository.findAccount(number));
    // add more ...
}
```

Kotlin

```
@ModelAttribute
fun populateModel(@RequestParam number: String, model: Model) {
    model.addAttribute(accountRepository.findAccount(number))
    // add more ...
}
```

The following example adds one attribute only:

Java

```
@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountRepository.findAccount(number);
}
```

Kotlin

```
@ModelAttribute
fun addAccount(@RequestParam number: String): Account {
    return accountRepository.findAccount(number);
}
```

When a name is not explicitly specified, a default name is chosen based on the type, as explained in the javadoc for `Conventions`. You can always assign an explicit name by using the overloaded `addAttribute` method or through the name attribute on `@ModelAttribute` (for a return value).

Spring WebFlux, unlike Spring MVC, explicitly supports reactive types in the model (for example, `Mono<Account>` or `io.reactivex.Single<Account>`). Such asynchronous model attributes can be transparently resolved (and the model updated) to their actual values at the time of `@RequestMapping` invocation, provided a `@ModelAttribute` argument is declared without a wrapper, as the following example shows:

Java

```
@ModelAttribute
public void addAccount(@RequestParam String number) {
    Mono<Account> accountMono = accountRepository.findAccount(number);
    model.addAttribute("account", accountMono);
}

@PostMapping("/accounts")
public String handle(@ModelAttribute Account account, BindingResult errors) {
    // ...
}
```

Kotlin

```
import org.springframework.ui.set

@ModelAttribute
fun addAccount(@RequestParam number: String) {
    val accountMono: Mono<Account> = accountRepository.findAccount(number)
    model["account"] = accountMono
}

@PostMapping("/accounts")
fun handle(@ModelAttribute account: Account, errors: BindingResult): String {
    // ...
}
```

In addition, any model attributes that have a reactive type wrapper are resolved to their actual values (and the model updated) just prior to view rendering.

You can also use `@ModelAttribute` as a method-level annotation on `@RequestMapping` methods, in which case the return value of the `@RequestMapping` method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a `String` that would otherwise be interpreted as a view name. `@ModelAttribute` can also help to customize the model attribute name, as the following example shows:

Java

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
public Account handle() {
    // ...
    return account;
}
```

Kotlin

```
@GetMapping("/accounts/{id}")
@ModelAttribute("myAccount")
fun handle(): Account {
    // ...
    return account
}
```

`DataBinder`

Web MVC

`@Controller` or `@ControllerAdvice` classes can have `@InitBinder` methods, to initialize instances of `WebDataBinder`. Those, in turn, are used to:

- Bind request parameters (that is, form data or query) to a model object.
- Convert `String`-based request values (such as request parameters, path variables, headers, cookies, and others) to the target type of controller method arguments.
- Format model object values as `String` values when rendering HTML forms.

`@InitBinder` methods can register controller-specific `java.beans.PropertyEditor` or Spring `Converter` and `Formatter` components. In addition, you can use the WebFlux Java configuration to register `Converter` and `Formatter` types in a globally shared `FormattingConversionService`.

`@InitBinder` methods support many of the same arguments that `@RequestMapping` methods do, except for `@ModelAttribute` (command object) arguments. Typically, they are declared with a `WebDataBinder` argument, for registrations, and a `void` return value. The following example uses the `@InitBinder` annotation:

Java

```
@Controller
public class FormController {

    @InitBinder ①
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat,
false));
    }

    // ...
}
```

① Using the `@InitBinder` annotation.

Kotlin

```
@Controller
class FormController {

    @InitBinder ①
    fun initBinder(binder: WebDataBinder) {
        val dateFormat = SimpleDateFormat("yyyy-MM-dd")
        dateFormat.isLenient = false
        binder.registerCustomEditor(Date::class.java, CustomDateEditor(dateFormat,
false))
    }

    // ...
}
```

Alternatively, when using a `Formatter`-based setup through a shared `FormattingConversionService`, you could re-use the same approach and register controller-specific `Formatter` instances, as the following example shows:

Java

```
@Controller
public class FormController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd")); ①
    }

    // ...
}
```

① Adding a custom formatter (a `DateFormatter`, in this case).

Kotlin

```
@Controller
class FormController {

    @InitBinder
    fun initBinder(binder: WebDataBinder) {
        binder.addCustomFormatter(DateFormatter("yyyy-MM-dd")) ①
    }

    // ...
}
```

① Adding a custom formatter (a `DateFormatter`, in this case).

Model Design

Web MVC

In the context of web applications, *data binding* involves the binding of HTTP request parameters (that is, form data or query parameters) to properties in a model object and its nested objects.

Only `public` properties following the JavaBeans naming conventions are exposed for data binding — for example, `public String getFirstName()` and `public void setFirstName(String)` methods for a `firstName` property.

The model object, and its nested object graph, is also sometimes referred to as a *command object*, *form-backing object*, or *POJO* (Plain Old Java Object).

By default, Spring permits binding to all public properties in the model object graph. This means you need to carefully consider what public properties the model has, since a client could target any public property path, even some that are not expected to be targeted for a given use case.

For example, given an HTTP form data endpoint, a malicious client could supply values for properties that exist in the model object graph but are not part of the HTML form presented in the browser. This could lead to data being set on the model object and any of its nested objects, that is not expected to be updated.

The recommended approach is to use a *dedicated model object* that exposes only properties that are relevant for the form submission. For example, on a form for changing a user's email address, the model object should declare a minimum set of properties such as in the following `ChangeEmailForm`.

```
public class ChangeEmailForm {

    private String oldEmailAddress;
    private String newEmailAddress;

    public void setOldEmailAddress(String oldEmailAddress) {
        this.oldEmailAddress = oldEmailAddress;
    }

    public String getOldEmailAddress() {
        return this.oldEmailAddress;
    }

    public void setNewEmailAddress(String newEmailAddress) {
        this.newEmailAddress = newEmailAddress;
    }

    public String getNewEmailAddress() {
        return this.newEmailAddress;
    }

}
```

If you cannot or do not want to use a *dedicated model object* for each data binding use case, you **must** limit the properties that are allowed for data binding. Ideally, you can achieve this by registering *allowed field patterns* via the `setAllowedFields()` method on `WebDataBinder`.

For example, to register allowed field patterns in your application, you can implement an `@InitBinder` method in a `@Controller` or `@ControllerAdvice` component as shown below:

```
@Controller
public class ChangeEmailController {

    @InitBinder
    void initBinder(WebDataBinder binder) {
        binder.setAllowedFields("oldEmailAddress", "newEmailAddress");
    }

    // @RequestMapping methods, etc.

}
```

In addition to registering allowed patterns, it is also possible to register *disallowed field patterns* via the `setDisallowedFields()` method in `DataBinder` and its subclasses. Please note, however, that an "allow list" is safer than a "deny list". Consequently, `setAllowedFields()` should be favored over `setDisallowedFields()`.

Note that matching against allowed field patterns is case-sensitive; whereas, matching against disallowed field patterns is case-insensitive. In addition, a field matching a disallowed pattern will not be accepted even if it also happens to match a pattern in the allowed list.

It is extremely important to properly configure allowed and disallowed field patterns when exposing your domain model directly for data binding purposes. Otherwise, it is a big security risk.

Furthermore, it is strongly recommended that you do **not** use types from your domain model such as JPA or Hibernate entities as the model object in data binding scenarios.

Exceptions

Web MVC

`@Controller` and @ControllerAdvice classes can have `@ExceptionHandler` methods to handle exceptions from controller methods. The following example includes such a handler method:

Java

```
@Controller
public class SimpleController {

    // ...

    @ExceptionHandler ①
    public ResponseEntity<String> handle(IOException ex) {
        // ...
    }
}
```

① Declaring an `@ExceptionHandler`.

Kotlin

```
@Controller
class SimpleController {

    // ...

    @ExceptionHandler ①
    fun handle(ex: IOException): ResponseEntity<String> {
        // ...
    }
}
```

① Declaring an `@ExceptionHandler`.

The exception can match against a top-level exception being propagated (that is, a direct `IOException` being thrown) or against the immediate cause within a top-level wrapper exception (for example, an `IOException` wrapped inside an `IllegalStateException`).

For matching exception types, preferably declare the target exception as a method argument, as shown in the preceding example. Alternatively, the annotation declaration can narrow the exception types to match. We generally recommend being as specific as possible in the argument signature and to declare your primary root exception mappings on a `@ControllerAdvice` prioritized with a corresponding order. See the MVC section for details.

An `@ExceptionHandler` method in WebFlux supports the same method arguments and return values as a `@RequestMapping` method, with the exception of request body- and `@ModelAttribute`-related method arguments.

Support for `@ExceptionHandler` methods in Spring WebFlux is provided by the `HandlerAdapter` for `@RequestMapping` methods. See `DispatcherHandler` for more detail.

Method Arguments

Web MVC

`@ExceptionHandler` methods support the same method arguments as `@RequestMapping` methods, except the request body might have been consumed already.

Return Values

Web MVC

`@ExceptionHandler` methods support the same return values as `@RequestMapping` methods.

Controller Advice

Web MVC

Typically, the `@ExceptionHandler`, `@InitBinder`, and `@ModelAttribute` methods apply within the `@Controller` class (or class hierarchy) in which they are declared. If you want such methods to apply more globally (across controllers), you can declare them in a class annotated with `@ControllerAdvice` or `@RestControllerAdvice`.

`@ControllerAdvice` is annotated with `@Component`, which means that such classes can be registered as Spring beans through component scanning. `@RestControllerAdvice` is a composed annotation that is annotated with both `@ControllerAdvice` and `@ResponseBody`, which essentially means `@ExceptionHandler` methods are rendered to the response body through message conversion (versus view resolution or template rendering).

On startup, the infrastructure classes for `@RequestMapping` and `@ExceptionHandler` methods detect Spring beans annotated with `@ControllerAdvice` and then apply their methods at runtime. Global `@ExceptionHandler` methods (from a `@ControllerAdvice`) are applied *after* local ones (from the `@Controller`). By contrast, global `@ModelAttribute` and `@InitBinder` methods are applied *before* local ones.

By default, `@ControllerAdvice` methods apply to every request (that is, all controllers), but you can narrow that down to a subset of controllers by using attributes on the annotation, as the following example shows:

Java

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class,
AbstractController.class})
public class ExampleAdvice3 {}
```

Kotlin

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = [RestController::class])
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = [ControllerInterface::class,
AbstractController::class])
public class ExampleAdvice3 {}
```

The selectors in the preceding example are evaluated at runtime and may negatively impact performance if used extensively. See the `@ControllerAdvice` javadoc for more details.

6.1.5. Functional Endpoints

Web MVC

Spring WebFlux includes WebFlux.fn, a lightweight functional programming model in which functions are used to route and handle requests and contracts are designed for immutability. It is an alternative to the annotation-based programming model but otherwise runs on the same Reactive Core foundation.

Overview

Web MVC

In WebFlux.fn, an HTTP request is handled with a `HandlerFunction`: a function that takes `ServerRequest` and returns a delayed `ServerResponse` (i.e. `Mono<ServerResponse>`). Both the request and the response object have immutable contracts that offer JDK 8-friendly access to the HTTP request and response. `HandlerFunction` is the equivalent of the body of a `@RequestMapping` method in the annotation-based programming model.

Incoming requests are routed to a handler function with a `RouterFunction`: a function that takes `ServerRequest` and returns a delayed `HandlerFunction` (i.e. `Mono<HandlerFunction>`). When the router function matches, a handler function is returned; otherwise an empty Mono. `RouterFunction` is the equivalent of a `@RequestMapping` annotation, but with the major difference that router functions provide not just data, but also behavior.

`RouterFunctions.route()` provides a router builder that facilitates the creation of routers, as the following example shows:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> route = route() ①
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson)
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople)
    .POST("/person", handler::createPerson)
    .build();


public class PersonHandler {

    // ...

    public Mono<ServerResponse> listPeople(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        // ...
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) {
        // ...
    }
}
```

① Create router using `route()`.

Kotlin

```
val repository: PersonRepository = ...
val handler = PersonHandler(repository)

val route = coRouter { ①
    accept(APPLICATION_JSON).nest {
        GET("/person/{id}", handler::getPerson)
        GET("/person", handler::listPeople)
    }
    POST("/person", handler::createPerson)
}


class PersonHandler(private val repository: PersonRepository) {

    // ...

    suspend fun listPeople(request: ServerRequest): ServerResponse {
        // ...
    }

    suspend fun createPerson(request: ServerRequest): ServerResponse {
        // ...
    }

    suspend fun getPerson(request: ServerRequest): ServerResponse {
        // ...
    }
}
```

① Create router using Coroutines router DSL; a Reactive alternative is also available via `router { }`.

One way to run a `RouterFunction` is to turn it into an `HttpHandler` and install it through one of the built-in server adapters:

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

Most applications can run through the WebFlux Java configuration, see Running a Server.

HandlerFunction

Web MVC

`ServerRequest` and `ServerResponse` are immutable interfaces that offer JDK 8-friendly access to the HTTP request and response. Both request and response provide Reactive Streams back pressure against the body streams. The request body is represented with a Reactor `Flux` or `Mono`. The response body is represented with any Reactive Streams `Publisher`, including `Flux` and `Mono`. For more on that, see Reactive Libraries.

ServerRequest

`ServerRequest` provides access to the HTTP method, URI, headers, and query parameters, while access to the body is provided through the `body` methods.

The following example extracts the request body to a `Mono<String>`:

Java

```
Mono<String> string = request.bodyToMono(String.class);
```

Kotlin

```
val string = request.awaitBody<String>()
```

The following example extracts the body to a `Flux<Person>` (or a `Flow<Person>` in Kotlin), where `Person` objects are decoded from some serialized form, such as JSON or XML:

Java

```
Flux<Person> people = request.bodyToFlux(Person.class);
```

Kotlin

```
val people = request.bodyToFlow<Person>()
```

The preceding examples are shortcuts that use the more general `ServerRequest.body(BodyExtractor)`, which accepts the `BodyExtractor` functional strategy interface. The utility class `BodyExtractors` provides access to a number of instances. For example, the preceding examples can also be written as follows:

Java

```
Mono<String> string = request.body(BodyExtractors.toMono(String.class));
Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class));
```

Kotlin

```
    val string = request.body(BodyExtractors.toMono(String::class.java)).awaitSingle()
    val people = request.body(BodyExtractors.toFlux(Person::class.java)).asFlow()
```

The following example shows how to access form data:

Java

```
Mono<MultiValueMap<String, String>> map = request.formData();
```

Kotlin

```
val map = request.awaitFormData()
```

The following example shows how to access multipart data as a map:

Java

```
Mono<MultiValueMap<String, Part>> map = request.multipartData();
```

Kotlin

```
val map = request.awaitMultipartData()
```

The following example shows how to access multipart data, one at a time, in streaming fashion:

Java

```
Flux<PartEvent> allPartEvents = request.bodyToFlux(PartEvent.class);
allPartsEvents.windowUntil(PartEvent::isLast)
      .concatMap(p -> p.switchOnFirst((signal, partEvents) -> {
          if (signal.hasValue()) {
              PartEvent event = signal.get();
              if (event instanceof FormPartEvent formEvent) {
                  String value = formEvent.value();
                  // handle form field
              }
              else if (event instanceof FilePartEvent fileEvent) {
                  String filename = fileEvent.filename();
                  Flux<DataBuffer> contents = partEvents.map(PartEvent::content);
                  // handle file upload
              }
              else {
                  return Mono.error(new RuntimeException("Unexpected event: " +
event));
              }
          }
          else {
              return partEvents; // either complete or error signal
          }
      }));
```

Kotlin

```
val parts = request.bodyToFlux<PartEvent>()
allPartsEvents.windowUntil(PartEvent::isLast)
    .concatMap {
        it.switchOnFirst { signal, partEvents ->
            if (signal.hasValue()) {
                val event = signal.get()
                if (event is FormPartEvent) {
                    val value: String = event.value();
                    // handle form field
                } else if (event is FilePartEvent) {
                    val filename: String = event.filename();
                    val contents: Flux<DataBuffer> =
partEvents.map(PartEvent::content);
                    // handle file upload
                } else {
                    return Mono.error(RuntimeException("Unexpected event: " + event));
                }
            } else {
                return partEvents; // either complete or error signal
            }
        }
    }
}
```

Note that the body contents of the `PartEvent` objects must be completely consumed, relayed, or released to avoid memory leaks.

ServerResponse

`ServerResponse` provides access to the HTTP response and, since it is immutable, you can use a `build` method to create it. You can use the builder to set the response status, to add response headers, or to provide a body. The following example creates a 200 (OK) response with JSON content:

Java

```
Mono<Person> person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).body(person,
Person.class);
```

Kotlin

```
val person: Person = ...
ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyValue(person)
```

The following example shows how to build a 201 (CREATED) response with a `Location` header and no body:

Java

```
URI location = ...
ServerResponse.created(location).build();
```

Kotlin

```
val location: URI = ...
ServerResponse.created(location).build()
```

Depending on the codec used, it is possible to pass hint parameters to customize how the body is serialized or deserialized. For example, to specify a Jackson JSON view:

Java

```
ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT,
MyJacksonView.class).body(...);
```

Kotlin

```
ServerResponse.ok().hint(Jackson2CodecSupport.JSON_VIEW_HINT,
MyJacksonView::class.java).body(...)
```

Handler Classes

We can write a handler function as a lambda, as the following example shows:

Java

```
HandlerFunction<ServerResponse> helloWorld =
  request -> ServerResponse.ok().bodyValue("Hello World");
```

Kotlin

```
val helloWorld = HandlerFunction<ServerResponse> {
ServerResponse.ok().bodyValue("Hello World") }
```

That is convenient, but in an application we need multiple functions, and multiple inline lambda's can get messy. Therefore, it is useful to group related handler functions together into a handler class, which has a similar role as `@Controller` in an annotation-based application. For example, the following class exposes a reactive `Person` repository:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;

public class PersonHandler {

    private final PersonRepository repository;

    public PersonHandler(PersonRepository repository) {
        this.repository = repository;
    }

    public Mono<ServerResponse> listPeople(ServerRequest request) { ①
        Flux<Person> people = repository.allPeople();
        return ok().contentType(APPLICATION_JSON).body(people, Person.class);
    }

    public Mono<ServerResponse> createPerson(ServerRequest request) { ②
        Mono<Person> person = request.bodyToMono(Person.class);
        return ok().build(repository.savePerson(person));
    }

    public Mono<ServerResponse> getPerson(ServerRequest request) { ③
        int personId = Integer.valueOf(request.pathVariable("id"));
        return repository.getPerson(personId)
            .flatMap(person -> ok().contentType(APPLICATION_JSON).bodyValue(person))
            .switchIfEmpty(ServerResponse.notFound().build());
    }
}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `createPerson` is a handler function that stores a new `Person` contained in the request body. Note that `PersonRepository.savePerson(Person)` returns `Mono<Void>`: an empty `Mono` that emits a completion signal when the person has been read from the request and stored. So we use the `build(Publisher<Void>)` method to send a response when that completion signal is received (that is, when the `Person` has been saved).

③ `getPerson` is a handler function that returns a single person, identified by the `id` path variable. We retrieve that `Person` from the repository and create a JSON response, if it is found. If it is not found, we use `switchIfEmpty(Mono<T>)` to return a 404 Not Found response.

Kotlin

```
class PersonHandler(private val repository: PersonRepository) {

    suspend fun listPeople(request: ServerRequest): ServerResponse { ①
        val people: Flow<Person> = repository.allPeople()
        return ok().contentType(APPLICATION_JSON).bodyAndAwait(people);
    }

    suspend fun createPerson(request: ServerRequest): ServerResponse { ②
        val person = request.awaitBody<Person>()
        repository.savePerson(person)
        return ok().buildAndAwait()
    }

    suspend fun getPerson(request: ServerRequest): ServerResponse { ③
        val personId = request.pathVariable("id").toInt()
        return repository.getPerson(personId)?.let {
ok().contentType(APPLICATION_JSON).bodyValueAndAwait(it) }
                ?: ServerResponse.notFound().buildAndAwait()

    }
}
```

① `listPeople` is a handler function that returns all `Person` objects found in the repository as JSON.

② `createPerson` is a handler function that stores a new `Person` contained in the request body. Note that `PersonRepository.savePerson(Person)` is a suspending function with no return type.

③ `getPerson` is a handler function that returns a single person, identified by the `id` path variable. We retrieve that `Person` from the repository and create a JSON response, if it is found. If it is not found, we return a 404 Not Found response.

Validation

A functional endpoint can use Spring's validation facilities to apply validation to the request body. For example, given a custom Spring Validator implementation for a `Person`:

Java

```
public class PersonHandler {

    private final Validator validator = new PersonValidator(); ①

    // ...

    public Mono<ServerResponse> createPerson(ServerRequest request) {
        Mono<Person> person =
request.bodyToMono(Person.class).doOnNext(this::validate); ②
        return ok().build(repository.savePerson(person));
    }

    private void validate(Person person) {
        Errors errors = new BeanPropertyBindingResult(person, "person");
        validator.validate(person, errors);
        if (errors.hasErrors()) {
            throw new ServerWebInputException(errors.toString()); ③
        }
    }
}
```

① Create `Validator` instance.

② Apply validation.

③ Raise exception for a 400 response.

Kotlin

```
class PersonHandler(private val repository: PersonRepository) {

    private val validator = PersonValidator() ①

    // ...

    suspend fun createPerson(request: ServerRequest): ServerResponse {
        val person = request.awaitBody<Person>()
        validate(person) ②
        repository.savePerson(person)
        return ok().buildAndAwait()
    }

    private fun validate(person: Person) {
        val errors: Errors = BeanPropertyBindingResult(person, "person");
        validator.validate(person, errors);
        if (errors.hasErrors()) {
            throw ServerWebInputException(errors.toString()) ③
        }
    }
}
```

① Create `Validator` instance.

② Apply validation.

③ Raise exception for a 400 response.

Handlers can also use the standard bean validation API (JSR-303) by creating and injecting a global `Validator` instance based on `LocalValidatorFactoryBean`. See Spring Validation.

`RouterFunction`

Web MVC

Router functions are used to route the requests to the corresponding `HandlerFunction`. Typically, you do not write router functions yourself, but rather use a method on the `RouterFunctions` utility class to create one. `RouterFunctions.route()` (no parameters) provides you with a fluent builder for creating a router function, whereas `RouterFunctions.route(RequestPredicate, HandlerFunction)` offers a direct way to create a router.

Generally, it is recommended to use the `route()` builder, as it provides convenient short-cuts for typical mapping scenarios without requiring hard-to-discover static imports. For instance, the router function builder offers the method `GET(String, HandlerFunction)` to create a mapping for GET requests; and `POST(String, HandlerFunction)` for POSTs.

Besides HTTP method-based mapping, the route builder offers a way to introduce additional predicates when mapping to requests. For each HTTP method there is an overloaded variant that takes a `RequestPredicate` as a parameter, though which additional constraints can be expressed.

Predicates

You can write your own `RequestPredicate`, but the `RequestPredicates` utility class offers commonly used implementations, based on the request path, HTTP method, content-type, and so on. The following example uses a request predicate to create a constraint based on the `Accept` header:

Java

```
RouterFunction<ServerResponse> route = RouterFunctions.route()
    .GET("/hello-world", accept(MediaType.TEXT_PLAIN),
        request -> ServerResponse.ok().bodyValue("Hello World")).build();
```

Kotlin

```
val route = coRouter {
    GET("/hello-world", accept(TEXT_PLAIN)) {
        ServerResponse.ok().bodyValueAndAwait("Hello World")
    }
}
```

You can compose multiple request predicates together by using:

- `RequestPredicate.and(RequestPredicate)` — both must match.
- `RequestPredicate.or(RequestPredicate)` — either can match.

Many of the predicates from `RequestPredicates` are composed. For example, `RequestPredicates.GET(String)` is composed from `RequestPredicates.method(HttpMethod)` and `RequestPredicates.path(String)`. The example shown above also uses two request predicates, as the builder uses `RequestPredicates.GET` internally, and composes that with the `accept` predicate.

Routes

Router functions are evaluated in order: if the first route does not match, the second is evaluated, and so on. Therefore, it makes sense to declare more specific routes before general ones. This is also important when registering router functions as Spring beans, as will be described later. Note that this behavior is different from the annotation-based programming model, where the "most specific" controller method is picked automatically.

When using the router function builder, all defined routes are composed into one `RouterFunction` that is returned from `build()`. There are also other ways to compose multiple router functions together:

- `add(RouterFunction)` on the `RouterFunctions.route()` builder
- `RouterFunction.and(RouterFunction)`
- `RouterFunction.andRoute(RequestPredicate, HandlerFunction)` — shortcut for `RouterFunction.and()` with nested `RouterFunctions.route()`.

The following example shows the composition of four routes:

Java

```
import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.RequestPredicates.*;

PersonRepository repository = ...
PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> otherRoute = ...

RouterFunction<ServerResponse> route = route()
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ①
    .GET("/person", accept(APPLICATION_JSON), handler::listPeople) ②
    .POST("/person", handler::createPerson) ③
    .add(otherRoute) ④
    .build();
```

① `GET /person/{id}` with an `Accept` header that matches JSON is routed to `PersonHandler.getPerson`

② `GET /person` with an `Accept` header that matches JSON is routed to `PersonHandler.listPeople`

③ `POST /person` with no additional predicates is mapped to `PersonHandler.createPerson`, and

④ `otherRoute` is a router function that is created elsewhere, and added to the route built.

Kotlin

```
import org.springframework.http.MediaType.APPLICATION_JSON

val repository: PersonRepository = ...
val handler = PersonHandler(repository);

val otherRoute: RouterFunction<ServerResponse> = coRouter {  }

val route = coRouter {
    GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson) ①
    GET("/person", accept(APPLICATION_JSON), handler::listPeople) ②
    POST("/person", handler::createPerson) ③
}.and(otherRoute) ④
```

① `GET /person/{id}` with an `Accept` header that matches JSON is routed to `PersonHandler.getPerson`

② `GET /person` with an `Accept` header that matches JSON is routed to `PersonHandler.listPeople`

③ `POST /person` with no additional predicates is mapped to `PersonHandler.createPerson`, and

④ `otherRoute` is a router function that is created elsewhere, and added to the route built.

Nested Routes

It is common for a group of router functions to have a shared predicate, for instance a shared path. In the example above, the shared predicate would be a path predicate that matches `/person`, used by three of the routes. When using annotations, you would remove this duplication by using a type-level `@RequestMapping` annotation that maps to `/person`. In WebFlux.fn, path predicates can be

shared through the `path` method on the router function builder. For instance, the last few lines of the example above can be improved in the following way by using nested routes:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", builder -> builder ①
        .GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        .GET(accept(APPLICATION_JSON), handler::listPeople)
        .POST(handler::createPerson))
    .build();
```

① Note that second parameter of `path` is a consumer that takes the router builder.

Kotlin

```
val route = coRouter { ①
    "/person".nest {
        GET("/{id}", accept(APPLICATION_JSON), handler::getPerson)
        GET(accept(APPLICATION_JSON), handler::listPeople)
        POST(handler::createPerson)
    }
}
```

① Create router using Coroutines router DSL; a Reactive alternative is also available via `router { }`.

Though path-based nesting is the most common, you can nest on any kind of predicate by using the `nest` method on the builder. The above still contains some duplication in the form of the shared `Accept`-header predicate. We can further improve by using the `nest` method together with `accept`:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople))
        .POST(handler::createPerson))
    .build();
```

Kotlin

```
val route = coRouter {
    "/person".nest {
        accept(APPLICATION_JSON).nest {
            GET("/{id}", handler::getPerson)
            GET(handler::listPeople)
            POST(handler::createPerson)
        }
    }
}
```

Running a Server

Web MVC

How do you run a router function in an HTTP server? A simple option is to convert a router function to an `HttpHandler` by using one of the following:

- `RouterFunctions.toHttpHandler(RouterFunction)`
- `RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)`

You can then use the returned `HttpHandler` with a number of server adapters by following HttpHandler for server-specific instructions.

A more typical option, also used by Spring Boot, is to run with a `DispatcherHandler`-based setup through the WebFlux Config, which uses Spring configuration to declare the components required to process requests. The WebFlux Java configuration declares the following infrastructure components to support functional endpoints:

- `RouterFunctionMapping`: Detects one or more `RouterFunction<?>` beans in the Spring configuration, orders them, combines them through `RouterFunction.andOther`, and routes requests to the resulting composed `RouterFunction`.
- `HandlerFunctionAdapter`: Simple adapter that lets `DispatcherHandler` invoke a `HandlerFunction` that was mapped to a request.
- `ServerResponseResultHandler`: Handles the result from the invocation of a `HandlerFunction` by invoking the `writeTo` method of the `ServerResponse`.

The preceding components let functional endpoints fit within the `DispatcherHandler` request processing lifecycle and also (potentially) run side by side with annotated controllers, if any are declared. It is also how functional endpoints are enabled by the Spring Boot WebFlux starter.

The following example shows a WebFlux Java configuration (see DispatcherHandler for how to run it):

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Bean
    public RouterFunction<?> routerFunctionA() {
        // ...
    }

    @Bean
    public RouterFunction<?> routerFunctionB() {
        // ...
    }

    // ...

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        // configure message conversion...
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        // configure CORS...
    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // configure view resolution for HTML rendering...
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    @Bean
    fun routerFunctionA(): RouterFunction<*> {
        // ...
    }

    @Bean
    fun routerFunctionB(): RouterFunction<*> {
        // ...
    }

    // ...

    override fun configureHttpMessageCodecs(configurer: ServerCodecConfigurer) {
        // configure message conversion...
    }

    override fun addCorsMappings(registry: CorsRegistry) {
        // configure CORS...
    }

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        // configure view resolution for HTML rendering...
    }
}
```

Filtering Handler Functions

Web MVC

You can filter handler functions by using the `before`, `after`, or `filter` methods on the routing function builder. With annotations, you can achieve similar functionality by using `@ControllerAdvice`, a `ServletFilter`, or both. The filter will apply to all routes that are built by the builder. This means that filters defined in nested routes do not apply to "top-level" routes. For instance, consider the following example:

Java

```
RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople)
            .before(request -> ServerRequest.from(request) ①
                .header("X-RequestHeader", "Value")
                .build()))
        .POST(handler::createPerson))
    .after((request, response) -> logResponse(response)) ②
    .build();
```

① The `before` filter that adds a custom request header is only applied to the two GET routes.

② The `after` filter that logs the response is applied to all routes, including the nested ones.

Kotlin

```
val route = router {
    "/person".nest {
        GET("/{id}", handler::getPerson)
        GET("", handler::listPeople)
        before { ①
            ServerRequest.from(it)
                    .header("X-RequestHeader", "Value").build()
        }
        POST(handler::createPerson)
        after { _, response -> ②
            logResponse(response)
        }
    }
}
```

① The `before` filter that adds a custom request header is only applied to the two GET routes.

② The `after` filter that logs the response is applied to all routes, including the nested ones.

The `filter` method on the router builder takes a `HandlerFilterFunction`: a function that takes a `ServerRequest` and `HandlerFunction` and returns a `ServerResponse`. The handler function parameter represents the next element in the chain. This is typically the handler that is routed to, but it can also be another filter if multiple are applied.

Now we can add a simple security filter to our route, assuming that we have a `SecurityManager` that can determine whether a particular path is allowed. The following example shows how to do so:

Java

```
SecurityManager securityManager = ...

RouterFunction<ServerResponse> route = route()
    .path("/person", b1 -> b1
        .nest(accept(APPLICATION_JSON), b2 -> b2
            .GET("/{id}", handler::getPerson)
            .GET(handler::listPeople))
        .POST(handler::createPerson))
    .filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    })
    .build();
```

Kotlin

```
val securityManager: SecurityManager = ...

val route = router {
        ("/person" and accept(APPLICATION_JSON)).nest {
            GET("/{id}", handler::getPerson)
            GET("", handler::listPeople)
            POST(handler::createPerson)
            filter { request, next ->
                if (securityManager.allowAccessTo(request.path())) {
                    next(request)
                }
                else {
                    status(UNAUTHORIZED).build();
                }
            }
        }
    }
```

The preceding example demonstrates that invoking the `next.handle(ServerRequest)` is optional. We only let the handler function be run when access is allowed.

Besides using the `filter` method on the router function builder, it is possible to apply a filter to an existing router function via `RouterFunction.filter(HandlerFilterFunction)`.

CORS support for functional endpoints is provided through a dedicated `CorsWebFilter`.

6.1.6. URI Links

Web MVC

This section describes various options available in the Spring Framework to prepare URIs.

UriComponents

Spring MVC and Spring WebFlux

`UriComponentsBuilder` helps to build URI's from URI templates with variables, as the following example shows:

Java

```
UriComponents uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")  ①
        .queryParam("q", "{q}")  ②
        .encode() ③
        .build(); ④

URI uri = uriComponents.expand("Westin", "123").toUri();  ⑤
```

① Static factory method with a URI template.

② Add or replace URI components.

③ Request to have the URI template and URI variables encoded.

④ Build a `UriComponents`.

⑤ Expand variables and obtain the `URI`.

Kotlin

```
val uriComponents = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")  ①
        .queryParam("q", "{q}")  ②
        .encode() ③
        .build() ④

val uri = uriComponents.expand("Westin", "123").toUri()  ⑤
```

① Static factory method with a URI template.

② Add or replace URI components.

③ Request to have the URI template and URI variables encoded.

④ Build a `UriComponents`.

⑤ Expand variables and obtain the `URI`.

The preceding example can be consolidated into one chain and shortened with `buildAndExpand`, as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri();
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("Westin", "123")
        .toUri()
```

You can shorten it further by going directly to a URI (which implies encoding), as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123")
```

You can shorten it further still with a full URI template, as the following example shows:

Java

```
URI uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123");
```

Kotlin

```
val uri = UriComponentsBuilder
        .fromUriString("https://example.com/hotels/{hotel}?q={q}")
        .build("Westin", "123")
```

UriBuilder

Spring MVC and Spring WebFlux

`UriComponentsBuilder` implements `UriBuilder`. You can create a `UriBuilder`, in turn, with a `UriBuilderFactory`. Together, `UriBuilderFactory` and `UriBuilder` provide a pluggable mechanism to build URIs from URI templates, based on shared configuration, such as a base URL, encoding preferences, and other details.

You can configure `RestTemplate` and `WebClient` with a `UriBuilderFactory` to customize the preparation of URIs. `DefaultUriBuilderFactory` is a default implementation of `UriBuilderFactory` that uses `UriComponentsBuilder` internally and exposes shared configuration options.

The following example shows how to configure a `RestTemplate`:

Java

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);
```

Kotlin

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode

val baseUrl = "https://example.org"
val factory = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val restTemplate = RestTemplate()
restTemplate.uriTemplateHandler = factory
```

The following example configures a `WebClient`:

Java

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "https://example.org";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

Kotlin

```
// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode

val baseUrl = "https://example.org"
val factory = DefaultUriBuilderFactory(baseUrl)
factory.encodingMode = EncodingMode.TEMPLATE_AND_VALUES

val client = WebClient.builder().uriBuilderFactory(factory).build()
```

In addition, you can also use `DefaultUriBuilderFactory` directly. It is similar to using `UriComponentsBuilder` but, instead of static factory methods, it is an actual instance that holds configuration and preferences, as the following example shows:

Java

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123");
```

Kotlin

```
val baseUrl = "https://example.com"
val uriBuilderFactory = DefaultUriBuilderFactory(baseUrl)

val uri = uriBuilderFactory.uriString("/hotels/{hotel}")
        .queryParam("q", "{q}")
        .build("Westin", "123")
```

URI Encoding

Spring MVC and Spring WebFlux

`UriComponentsBuilder` exposes encoding options at two levels:

- UriComponentsBuilder#encode(): Pre-encodes the URI template first and then strictly encodes

URI variables when expanded.

- UriComponents#encode(): Encodes URI components *after* URI variables are expanded.

Both options replace non-ASCII and illegal characters with escaped octets. However, the first option also replaces characters with reserved meaning that appear in URI variables.

Consider ";", which is legal in a path but has reserved meaning. The first option replaces ";" with "%3B" in URI variables but not in the URI template. By contrast, the second option never replaces ";", since it is a legal character in a path.

For most cases, the first option is likely to give the expected result, because it treats URI variables as opaque data to be fully encoded, while the second option is useful if URI variables do intentionally contain reserved characters. The second option is also useful when not expanding URI variables at all since that will also encode anything that incidentally looks like a URI variable.

The following example uses the first option:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri();

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .encode()
        .buildAndExpand("New York", "foo+bar")
        .toUri()

// Result is "/hotel%20list/New%20York?q=foo%2Bbar"
```

You can shorten the preceding example by going directly to the URI (which implies encoding), as the following example shows:

Java

```
URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar");
```

Kotlin

```
val uri = UriComponentsBuilder.fromPath("/hotel list/{city}")
        .queryParam("q", "{q}")
        .build("New York", "foo+bar")
```

You can shorten it further still with a full URI template, as the following example shows:

Java

```
URI uri = UriComponentsBuilder.fromUriString("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar");
```

Kotlin

```
val uri = UriComponentsBuilder.fromUriString("/hotel list/{city}?q={q}")
        .build("New York", "foo+bar")
```

The `WebClient` and the `RestTemplate` expand and encode URI templates internally through the `UriBuilderFactory` strategy. Both can be configured with a custom strategy, as the following example shows:

Java

```
String baseUrl = "https://example.com";
DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)
factory.setEncodingMode(EncodingMode.TEMPLATE_AND_VALUES);

// Customize the RestTemplate..
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..
WebClient client = WebClient.builder().uriBuilderFactory(factory).build();
```

Kotlin

```
val baseUrl = "https://example.com"
val factory = DefaultUriBuilderFactory(baseUrl).apply {
    encodingMode = EncodingMode.TEMPLATE_AND_VALUES
}

// Customize the RestTemplate..
val restTemplate = RestTemplate().apply {
    uriTemplateHandler = factory
}

// Customize the WebClient..
val client = WebClient.builder().uriBuilderFactory(factory).build()
```

The `DefaultUriBuilderFactory` implementation uses `UriComponentsBuilder` internally to expand and encode URI templates. As a factory, it provides a single place to configure the approach to encoding, based on one of the below encoding modes:

- `TEMPLATE_AND_VALUES`: Uses `UriComponentsBuilder#encode()`, corresponding to the first option in the earlier list, to pre-encode the URI template and strictly encode URI variables when expanded.
- `VALUES_ONLY`: Does not encode the URI template and, instead, applies strict encoding to URI variables through `UriUtils#encodeUriVariables` prior to expanding them into the template.
- `URI_COMPONENT`: Uses `UriComponents#encode()`, corresponding to the second option in the earlier list, to encode URI component value *after* URI variables are expanded.
- `NONE`: No encoding is applied.

The `RestTemplate` is set to `EncodingMode.URI_COMPONENT` for historic reasons and for backwards compatibility. The `WebClient` relies on the default value in `DefaultUriBuilderFactory`, which was changed from `EncodingMode.URI_COMPONENT` in 5.0.x to `EncodingMode.TEMPLATE_AND_VALUES` in 5.1.

6.1.7. CORS

Web MVC

Spring WebFlux lets you handle CORS (Cross-Origin Resource Sharing). This section describes how to do so.

Introduction

Web MVC

For security reasons, browsers prohibit AJAX calls to resources outside the current origin. For example, you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials — for example, withdrawing money from your account!

Cross-Origin Resource Sharing (CORS) is a W3C specification implemented by most browsers that

lets you specify what kind of cross-domain requests are authorized, rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

Processing

Web MVC

The CORS specification distinguishes between preflight, simple, and actual requests. To learn how CORS works, you can read this article, among many others, or see the specification for more details.

Spring WebFlux `HandlerMapping` implementations provide built-in support for CORS. After successfully mapping a request to a handler, a `HandlerMapping` checks the CORS configuration for the given request and handler and takes further actions. Preflight requests are handled directly, while simple and actual CORS requests are intercepted, validated, and have the required CORS response headers set.

In order to enable cross-origin requests (that is, the `Origin` header is present and differs from the host of the request), you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the responses of simple and actual CORS requests and, consequently, browsers reject them.

Each `HandlerMapping` can be configured individually with URL pattern-based `CorsConfiguration` mappings. In most cases, applications use the WebFlux Java configuration to declare such mappings, which results in a single, global map passed to all `HandlerMapping` implementations.

You can combine global CORS configuration at the `HandlerMapping` level with more fine-grained, handler-level CORS configuration. For example, annotated controllers can use class- or method-level `@CrossOrigin` annotations (other handlers can implement `CorsConfigurationSource`).

The rules for combining global and local configuration are generally additive — for example, all global and all local origins. For those attributes where only a single value can be accepted, such as `allowCredentials` and `maxAge`, the local overrides the global value. See `CorsConfiguration#combine(CorsConfiguration)` for more details.

To learn more from the source or to make advanced customizations, see:

- `CorsConfiguration`
- `CorsProcessor` and `DefaultCorsProcessor`
- `AbstractHandlerMapping`

`@CrossOrigin`

Web MVC

The `@CrossOrigin` annotation enables cross-origin requests on annotated controller methods, as the following example shows:

Java

```
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@RestController
@RequestMapping("/account")
class AccountController {

    @CrossOrigin
    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

By default, `@CrossOrigin` allows:

- All origins.
- All headers.
- All HTTP methods to which the controller method is mapped.

`allowCredentials` is not enabled by default, since that establishes a trust level that exposes sensitive user-specific information (such as cookies and CSRF tokens) and should be used only where appropriate. When it is enabled either `allowOrigins` must be set to one or more specific domain (but not the special value `"*"`) or alternatively the `allowOriginPatterns` property may be used to match to a dynamic set of origins.

`maxAge` is set to 30 minutes.

`@CrossOrigin` is supported at the class level, too, and inherited by all methods. The following example specifies a certain domain and sets `maxAge` to an hour:

Java

```
@CrossOrigin(origins = "https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
public class AccountController {

    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

Kotlin

```
@CrossOrigin("https://domain2.com", maxAge = 3600)
@RestController
@RequestMapping("/account")
class AccountController {

    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

You can use `@CrossOrigin` at both the class and the method level, as the following example shows:

Java

```
@CrossOrigin(maxAge = 3600) ①
@RestController
@RequestMapping("/account")
public class AccountController {

    @CrossOrigin("https://domain2.com") ②
    @GetMapping("/{id}")
    public Mono<Account> retrieve(@PathVariable Long id) {
        // ...
    }

    @DeleteMapping("/{id}")
    public Mono<Void> remove(@PathVariable Long id) {
        // ...
    }
}
```

① Using `@CrossOrigin` at the class level.

② Using `@CrossOrigin` at the method level.

Kotlin

```
@CrossOrigin(maxAge = 3600) ①
@RestController
@RequestMapping("/account")
class AccountController {

    @CrossOrigin("https://domain2.com") ②
    @GetMapping("/{id}")
    suspend fun retrieve(@PathVariable id: Long): Account {
        // ...
    }

    @DeleteMapping("/{id}")
    suspend fun remove(@PathVariable id: Long) {
        // ...
    }
}
```

① Using `@CrossOrigin` at the class level.

② Using `@CrossOrigin` at the method level.

Global Configuration

Web MVC

In addition to fine-grained, controller method-level configuration, you probably want to define some global CORS configuration, too. You can set URL-based `CorsConfiguration` mappings

individually on any `HandlerMapping`. Most applications, however, use the WebFlux Java configuration to do that.

By default global configuration enables the following:

- All origins.
- All headers.
- `GET`, `HEAD`, and `POST` methods.

`allowedCredentials` is not enabled by default, since that establishes a trust level that exposes sensitive user-specific information(such as cookies and CSRF tokens) and should be used only where appropriate. When it is enabled either `allowOrigins` must be set to one or more specific domain (but not the special value `"*"`) or alternatively the `allowOriginPatterns` property may be used to match to a dynamic set of origins.

`maxAge` is set to 30 minutes.

To enable CORS in the WebFlux Java configuration, you can use the `CorsRegistry` callback, as the following example shows:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/api/**")
            .allowedOrigins("https://domain2.com")
            .allowedMethods("PUT", "DELETE")
            .allowedHeaders("header1", "header2", "header3")
            .exposedHeaders("header1", "header2")
            .allowCredentials(true).maxAge(3600);

        // Add more mappings...
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addCorsMappings(registry: CorsRegistry) {

        registry.addMapping("/api/**")
                .allowedOrigins("https://domain2.com")
                .allowedMethods("PUT", "DELETE")
                .allowedHeaders("header1", "header2", "header3")
                .exposedHeaders("header1", "header2")
                .allowCredentials(true).maxAge(3600)

        // Add more mappings...
    }
}
```

CORS `WebFilter`

Web MVC

You can apply CORS support through the built-in `CorsWebFilter`, which is a good fit with functional endpoints.

If you try to use the `CorsFilter` with Spring Security, keep in mind that Spring Security has built-in support for CORS.

To configure the filter, you can declare a `CorsWebFilter` bean and pass a `CorsConfigurationSource` to its constructor, as the following example shows:

Java

```
@Bean
CorsWebFilter corsFilter() {

    CorsConfiguration config = new CorsConfiguration();

    // Possibly...
    // config.applyPermitDefaultValues()

    config.setAllowCredentials(true);
    config.addAllowedOrigin("https://domain1.com");
    config.addAllowedHeader("*");
    config.addAllowedMethod("*");

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);

    return new CorsWebFilter(source);
}
```

Kotlin

```
@Bean
fun corsFilter(): CorsWebFilter {

    val config = CorsConfiguration()

    // Possibly...
    // config.applyPermitDefaultValues()

    config.allowCredentials = true
    config.addAllowedOrigin("https://domain1.com")
    config.addAllowedHeader("*")
    config.addAllowedMethod("*")

    val source = UrlBasedCorsConfigurationSource().apply {
        registerCorsConfiguration("/**", config)
    }
    return CorsWebFilter(source)
}
```

6.1.8. Error Responses

Web MVC

A common requirement for REST services is to include details in the body of error responses. The Spring Framework supports the "Problem Details for HTTP APIs" specification, RFC 7807.

The following are the main abstractions for this support:

- `ProblemDetail` — representation for an RFC 7807 problem detail; a simple container for both standard fields defined in the spec, and for non-standard ones.
- `ErrorResponse` — contract to expose HTTP error response details including HTTP status, response headers, and a body in the format of RFC 7807; this allows exceptions to encapsulate and expose the details of how they map to an HTTP response. All Spring WebFlux exceptions implement this.
- `ErrorResponseException` — basic `ErrorResponse` implementation that others can use as a convenient base class.
- `ResponseEntityExceptionHandler` — convenient base class for an @ControllerAdvice that handles all Spring WebFlux exceptions, and any `ErrorResponseException`, and renders an error response with a body.

Render

Web MVC

You can return `ProblemDetail` or `ErrorResponse` from any `@ExceptionHandler` or from any `@RequestMapping` method to render an RFC 7807 response. This is processed as follows:

- The `status` property of `ProblemDetail` determines the HTTP status.
- The `instance` property of `ProblemDetail` is set from the current URL path, if not already set.
- For content negotiation, the Jackson `HttpMessageConverter` prefers "application/problem+json" over "application/json" when rendering a `ProblemDetail`, and also falls back on it if no compatible media type is found.

To enable RFC 7807 responses for Spring WebFlux exceptions and for any `ErrorResponseException`, extend `ResponseEntityExceptionHandler` and declare it as an @ControllerAdvice in Spring configuration. The handler has an `@ExceptionHandler` method that handles any `ErrorResponse` exception, which includes all built-in web exceptions. You can add more exception handling methods, and use a protected method to map any exception to a `ProblemDetail`.

Non-Standard Fields

Web MVC

You can extend an RFC 7807 response with non-standard fields in one of two ways.

One, insert into the "properties" `Map` of `ProblemDetail`. When using the Jackson library, the Spring Framework registers `ProblemDetailJacksonMixin` that ensures this "properties" `Map` is unwrapped and rendered as top level JSON properties in the response, and likewise any unknown property during deserialization is inserted into this `Map`.

You can also extend `ProblemDetail` to add dedicated non-standard properties. The copy constructor in `ProblemDetail` allows a subclass to make it easy to be created from an existing `ProblemDetail`. This could be done centrally, e.g. from an `@ControllerAdvice` such as `ResponseEntityExceptionHandler` that re-creates the `ProblemDetail` of an exception into a subclass with the additional non-standard fields.

Internationalization

Web MVC

It is a common requirement to internationalize error response details, and good practice to customize the problem details for Spring WebFlux exceptions. This is supported as follows:

- Each `ErrorResponse` exposes a message code and arguments to resolve the "detail" field through a MessageSource. The actual message code value is parameterized with placeholders, e.g. `"HTTP method {0} not supported"` to be expanded from the arguments.
- Each `ErrorResponse` also exposes a message code to resolve the "title" field.
- `ResponseEntityExceptionHandler` uses the message code and arguments to resolve the "detail" and the "title" fields.

By default, the message code for the "detail" field is "problemDetail." + the fully qualified exception class name. Some exceptions may expose additional message codes in which case a suffix is added to the default message code. The table below lists message arguments and codes for Spring WebFlux exceptions:

Exception	Message Code	Message Code Arguments
`UnsupportedMediaTypeStatusException`	(default)	`{0}` the media type that is not supported, `{1}` list of supported media types
`UnsupportedMediaTypeStatusException`	(default) + ".parseError"	
`MissingRequestValueException`	(default)	`{0}` a label for the value (e.g. "request header", "cookie value", ...), `{1}` the value name
`UnsatisfiedRequestParameterException`	(default)	`{0}` the list of parameter conditions
`WebExchangeBindException`	(default)	`{0}` the list of global errors, `{1}` the list of field errors. Message codes and arguments for each error within the `BindingResult` are also resolved via `MessageSource`.
`NotAcceptableStatusException`	(default)	`{0}` list of supported media types
`NotAcceptableStatusException`	(default) + ".parseError"	
`ServerErrorException`	(default)	`{0}` the failure reason provided to the class constructor
`MethodNotAllowedException`	(default)	`{0}` the current HTTP method, `{1}` the list of supported HTTP methods

By default, the message code for the "title" field is "problemDetail.title." + the fully qualified exception class name.

Client Handling

Web MVC

A client application can catch `WebClientResponseException`, when using the `WebClient`, or `RestClientResponseException` when using the `RestTemplate`, and use their `getResponseBodyAs` methods to decode the error response body to any target type such as `ProblemDetail`, or a subclass of `ProblemDetail`.

6.1.9. Web Security

Web MVC

The Spring Security project provides support for protecting web applications from malicious exploits. See the Spring Security reference documentation, including:

- WebFlux Security
- WebFlux Testing Support
- CSRF protection
- Security Response Headers

6.1.10. HTTP Caching

Web MVC

HTTP caching can significantly improve the performance of a web application. HTTP caching revolves around the `Cache-Control` response header and subsequent conditional request headers, such as `Last-Modified` and `ETag`. `Cache-Control` advises private (for example, browser) and public (for example, proxy) caches how to cache and re-use responses. An `ETag` header is used to make a conditional request that may result in a 304 (NOT_MODIFIED) without a body, if the content has not changed. `ETag` can be seen as a more sophisticated successor to the `Last-Modified` header.

This section describes the HTTP caching related options available in Spring WebFlux.

`CacheControl`

Web MVC

`CacheControl` provides support for configuring settings related to the `Cache-Control` header and is accepted as an argument in a number of places:

- Controllers
- Static Resources

While RFC 7234 describes all possible directives for the `Cache-Control` response header, the `CacheControl` type takes a use case-oriented approach that focuses on the common scenarios, as the following example shows:

Java

```
// Cache for an hour - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10,
TimeUnit.DAYS).noTransform().cachePublic();
```

Kotlin

```
// Cache for an hour - "Cache-Control: max-age=3600"
val ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS)

// Prevent caching - "Cache-Control: no-store"
val ccNoStore = CacheControl.noStore()

// Cache for ten days in public and private caches,
// public caches should not transform the response
// "Cache-Control: max-age=864000, public, no-transform"
val ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic()
```

Controllers

Web MVC

Controllers can add explicit support for HTTP caching. We recommend doing so, since the `lastModified` or `ETag` value for a resource needs to be calculated before it can be compared against conditional request headers. A controller can add an `ETag` and `Cache-Control` settings to a `ResponseEntity`, as the following example shows:

Java

```
@GetMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book);
}
```

Kotlin

```
@GetMapping("/book/{id}")
fun showBook(@PathVariable id: Long): ResponseEntity<Book> {

    val book = findBook(id)
    val version = book.getVersion()

    return ResponseEntity
            .ok()
            .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
            .eTag(version) // lastModified is also available
            .body(book)
}
```

The preceding example sends a 304 (NOT_MODIFIED) response with an empty body if the comparison to the conditional request headers indicates the content has not changed. Otherwise, the `ETag` and `Cache-Control` headers are added to the response.

You can also make the check against conditional request headers in the controller, as the following example shows:

Java

```
@RequestMapping
public String myHandleMethod(ServerWebExchange exchange, Model model) {

    long eTag = ... ①

    if (exchange.checkNotModified(eTag)) {
        return null; ②
    }

    model.addAttribute(...); ③
    return "myViewName";
}
```

① Application-specific calculation.

② Response has been set to 304 (NOT_MODIFIED). No further processing.

③ Continue with request processing.

Kotlin

```
@RequestMapping
fun myHandleMethod(exchange: ServerWebExchange, model: Model): String? {

    val eTag: Long = ... ①

    if (exchange.checkNotModified(eTag)) {
        return null②
    }

    model.addAttribute(...) ③
    return "myViewName"
}
```

① Application-specific calculation.

② Response has been set to 304 (NOT_MODIFIED). No further processing.

③ Continue with request processing.

There are three variants for checking conditional requests against `eTag` values, `lastModified` values, or both. For conditional `GET` and `HEAD` requests, you can set the response to 304 (NOT_MODIFIED). For conditional `POST`, `PUT`, and `DELETE`, you can instead set the response to 412 (PRECONDITION_FAILED) to prevent concurrent modification.

Static Resources

Web MVC

You should serve static resources with a `Cache-Control` and conditional response headers for optimal performance. See the section on configuring Static Resources.

6.1.11. View Technologies

Web MVC

The use of view technologies in Spring WebFlux is pluggable. Whether you decide to use Thymeleaf, FreeMarker, or some other view technology is primarily a matter of a configuration change. This chapter covers the view technologies integrated with Spring WebFlux. We assume you are already familiar with View Resolution.

Thymeleaf

Web MVC

Thymeleaf is a modern server-side Java template engine that emphasizes natural HTML templates that can be previewed in a browser by double-clicking, which is very helpful for independent work on UI templates (for example, by a designer) without the need for a running server. Thymeleaf offers an extensive set of features, and it is actively developed and maintained. For a more complete introduction, see the Thymeleaf project home page.

The Thymeleaf integration with Spring WebFlux is managed by the Thymeleaf project. The configuration involves a few bean declarations, such as `SpringResourceTemplateResolver`, `SpringWebFluxTemplateEngine`, and `ThymeleafReactiveViewResolver`. For more details, see Thymeleaf+Spring and the WebFlux integration announcement.

FreeMarker

Web MVC

Apache FreeMarker is a template engine for generating any kind of text output from HTML to email and others. The Spring Framework has built-in integration for using Spring WebFlux with FreeMarker templates.

View Configuration

Web MVC

The following example shows how to configure FreeMarker as a view technology:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure FreeMarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates/freemarker");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()
    }

    // Configure FreeMarker...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("classpath:/templates/freemarker")
    }
}
```

Your templates need to be stored in the directory specified by the `FreeMarkerConfigurer`, shown in the preceding example. Given the preceding configuration, if your controller returns the view name, `welcome`, the resolver looks for the `classpath:/templates/freemarker/welcome.ftl` template.

FreeMarker Configuration

Web MVC

You can pass FreeMarker 'Settings' and 'SharedVariables' directly to the FreeMarker `Configuration` object (which is managed by Spring) by setting the appropriate bean properties on the `FreeMarkerConfigurer` bean. The `freemarkerSettings` property requires a `java.util.Properties` object,

and the `freemarkerVariables` property requires a `java.util.Map`. The following example shows how to use a `FreeMarkerConfigurer`:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // ...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        Map<String, Object> variables = new HashMap<>();
        variables.put("xml_escape", new XmlEscape());

        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        configurer.setFreemarkerVariables(variables);
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    // ...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("classpath:/templates")
        setFreemarkerVariables(mapOf("xml_escape" to XmlEscape()))
    }
}
```

See the FreeMarker documentation for details of settings and variables as they apply to the `Configuration` object.

Form Handling

Web MVC

Spring provides a tag library for use in JSPs that contains, among others, a `<spring:bind/>` element. This element primarily lets forms display values from form-backing objects and show the results of failed validations from a `Validator` in the web or business tier. Spring also has support for the same functionality in FreeMarker, with additional convenience macros for generating form input elements themselves.

The Bind Macros

Web MVC

A standard set of macros are maintained within the `spring-webflux.jar` file for FreeMarker, so they are always available to a suitably configured application.

Some of the macros defined in the Spring templating libraries are considered internal (private), but no such scoping exists in the macro definitions, making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to directly call from within your templates. If you wish to view the macro code directly, the file is called `spring.ftl` and is in the `org.springframework.web.reactive.result.view.freemarker` package.

For additional details on binding support, see Simple Binding for Spring MVC.

Form Macros

For details on Spring's form macro support for FreeMarker templates, consult the following sections of the Spring MVC documentation.

- Input Macros
- Input Fields
- Selection Fields
- HTML Escaping

Script Views

Web MVC

The Spring Framework has a built-in integration for using Spring WebFlux with any templating library that can run on top of the JSR-223 Java scripting engine. The following table shows the templating libraries that we have tested on different script engines:

Scripting Library	Scripting Engine
Handlebars	Nashorn
Mustache	Nashorn
React	Nashorn
EJS	Nashorn
ERB	JRuby
String templates	Jython
Kotlin Script templating	Kotlin

The basic rule for integrating any other script engine is that it must implement the `ScriptEngine` and `Invocable` interfaces.

Requirements

Web MVC

You need to have the script engine on your classpath, the details of which vary by script engine:

- The Nashorn JavaScript engine is provided with Java 8+. Using the latest update release available is highly recommended.
- JRuby should be added as a dependency for Ruby support.
- Jython should be added as a dependency for Python support.
- `org.jetbrains.kotlin:kotlin-script-util` dependency and a `META-INF/services/javax.script.ScriptEngineFactory` file containing a `org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory` line should be added for Kotlin script support. See this example for more detail.

You need to have the script templating library. One way to do that for JavaScript is through WebJars.

Script Templates

Web MVC

You can declare a `ScriptTemplateConfigurer` bean to specify the script engine to use, the script files to load, what function to call to render templates, and so on. The following example uses Mustache templates and the Nashorn JavaScript engine:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("mustache.js");
        configurer.setRenderObject("Mustache");
        configurer.setRenderFunction("render");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.scriptTemplate()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply {
        engineName = "nashorn"
        setScripts("mustache.js")
        renderObject = "Mustache"
        renderFunction = "render"
    }
}
```

The `render` function is called with the following parameters:

- `String template`: The template content
- `Map model`: The view model
- `RenderingContext renderingContext`: The `RenderingContext` that gives access to the application context, the locale, the template loader, and the URL (since 5.0)

`Mustache.render()` is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you can provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them and requires a polyfill in order to emulate some browser facilities not available in the server-side script engine. The following example shows how to set a custom render function:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.scriptTemplate();
    }

    @Bean
    public ScriptTemplateConfigurer configurer() {
        ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();
        configurer.setEngineName("nashorn");
        configurer.setScripts("polyfill.js", "handlebars.js", "render.js");
        configurer.setRenderFunction("render");
        configurer.setSharedEngine(false);
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.scriptTemplate()
    }

    @Bean
    fun configurer() = ScriptTemplateConfigurer().apply {
        engineName = "nashorn"
        setScripts("polyfill.js", "handlebars.js", "render.js")
        renderFunction = "render"
        isSharedEngine = false
    }
}
```

Setting the `sharedEngine` property to `false` is required when using non-thread-safe script engines with templating libraries not designed for concurrency, such as Handlebars or React running on Nashorn. In that case, Java SE 8 update 60 is required, due to this bug, but it is generally recommended to use a recent Java SE patch release in any case.

`polyfill.js` defines only the `window` object needed by Handlebars to run properly, as the following snippet shows:

```
var window = {};
```

This basic `render.js` implementation compiles the template before using it. A production ready implementation should also store and reused cached templates or pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example). The following example shows how compile a template:

```
function render(template, model) {
    var compiledTemplate = Handlebars.compile(template);
    return compiledTemplate(model);
}
```

Check out the Spring Framework unit tests, Java, and resources, for more configuration examples.

JSON and XML

Web MVC

For Content Negotiation purposes, it is useful to be able to alternate between rendering a model with an HTML template or as other formats (such as JSON or XML), depending on the content type requested by the client. To support doing so, Spring WebFlux provides the `HttpMessageWriterView`, which you can use to plug in any of the available Codecs from `spring-web`, such as `Jackson2JsonEncoder`, `Jackson2SmileEncoder`, or `Jaxb2XmlEncoder`.

Unlike other view technologies, `HttpMessageWriterView` does not require a `ViewResolver` but is instead configured as a default view. You can configure one or more such default views, wrapping different `HttpMessageWriter` instances or `Encoder` instances. The one that matches the requested content type is used at runtime.

In most cases, a model contains multiple attributes. To determine which one to serialize, you can configure `HttpMessageWriterView` with the name of the model attribute to use for rendering. If the model contains only one attribute, that one is used.

6.1.12. WebFlux Config

Web MVC

The WebFlux Java configuration declares the components that are required to process requests with annotated controllers or functional endpoints, and it offers an API to customize the configuration. That means you do not need to understand the underlying beans created by the Java configuration. However, if you want to understand them, you can see them in `WebFluxConfigurationSupport` or read more about what they are in Special Bean Types.

For more advanced customizations, not available in the configuration API, you can gain full control over the configuration through the Advanced Configuration Mode.

Enabling WebFlux Config

Web MVC

You can use the `@EnableWebFlux` annotation in your Java config, as the following example shows:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig {
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig
```

The preceding example registers a number of Spring WebFlux infrastructure beans and adapts to dependencies available on the classpath — for JSON, XML, and others.

WebFlux config API

Web MVC

In your Java configuration, you can implement the `WebFluxConfigurer` interface, as the following example shows:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    // Implement configuration methods...
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    // Implement configuration methods...
}
```

Conversion, formatting

Web MVC

By default, formatters for various number and date types are installed, along with support for customization via `@NumberFormat` and `@DateTimeFormat` on fields.

To register custom formatters and converters in Java config, use the following:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // ...
    }

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addFormatters(registry: FormatterRegistry) {
        // ...
    }
}
```

By default Spring WebFlux considers the request Locale when parsing and formatting date values. This works for forms where dates are represented as Strings with "input" form fields. For "date" and "time" form fields, however, browsers use a fixed format defined in the HTML spec. For such cases date and time formatting can be customized as follows:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        DateTimeFormatterRegistrar registrar = new DateTimeFormatterRegistrar();
        registrar.setUseIsoFormat(true);
        registrar.registerFormatters(registry);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addFormatters(registry: FormatterRegistry) {
        val registrar = DateTimeFormatterRegistrar()
        registrar.setUseIsoFormat(true)
        registrar.registerFormatters(registry)
    }
}
```

See `FormatterRegistrar` SPI and the `FormattingConversionServiceFactoryBean` for more information on when to use `FormatterRegistrar` implementations.

Validation

Web MVC

By default, if Bean Validation is present on the classpath (for example, the Hibernate Validator), the `LocalValidatorFactoryBean` is registered as a global validator for use with `@Valid` and `@Validated` on `@Controller` method arguments.

In your Java configuration, you can customize the global `Validator` instance, as the following example shows:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public Validator getValidator() {
        // ...
    }

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun getValidator(): Validator {
        // ...
    }

}
```

Note that you can also register `Validator` implementations locally, as the following example shows:

Java

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

Kotlin

```
@Controller
class MyController {

    @InitBinder
    protected fun initBinder(binder: WebDataBinder) {
        binder.addValidators(FooValidator())
    }
}
```

If you need to have a `LocalValidatorFactoryBean` injected somewhere, create a bean and mark it with `@Primary` in order to avoid conflict with the one declared in the MVC config.

Content Type Resolvers

Web MVC

You can configure how Spring WebFlux determines the requested media types for `@Controller` instances from the request. By default, only the `Accept` header is checked, but you can also enable a query parameter-based strategy.

The following example shows how to customize the requested content type resolution:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureContentTypeResolver(RequestedContentTypeResolverBuilder
builder) {
        // ...
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureContentTypeResolver(builder:
RequestedContentTypeResolverBuilder) {
        // ...
    }
}
```

HTTP message codecs

Web MVC

The following example shows how to customize how the request and response body are read and written:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {
        configurer.defaultCodecs().maxInMemorySize(512 * 1024);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureHttpMessageCodecs(configurer: ServerCodecConfigurer) {
        // ...
    }
}
```

`ServerCodecConfigurer` provides a set of default readers and writers. You can use it to add more readers and writers, customize the default ones, or replace the default ones completely.

For Jackson JSON and XML, consider using `Jackson2ObjectMapperBuilder`, which customizes Jackson's default properties with the following ones:

- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled.
- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled.

It also automatically registers the following well-known modules if they are detected on the classpath:

- `jackson-datatype-joda`: Support for Joda-Time types.
- `jackson-datatype-jsr310`: Support for Java 8 Date and Time API types.
- `jackson-datatype-jdk8`: Support for other Java 8 types, such as `Optional`.
- `jackson-module-kotlin`: Support for Kotlin classes and data classes.

View Resolvers

Web MVC

The following example shows how to configure view resolution:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        // ...
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        // ...
    }
}
```

The `ViewResolverRegistry` has shortcuts for view technologies with which the Spring Framework integrates. The following example uses FreeMarker (which also requires configuring the underlying FreeMarker view technology):

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();
    }

    // Configure Freemarker...

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates");
        return configurer;
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()
    }

    // Configure Freemarker...

    @Bean
    fun freeMarkerConfigurer() = FreeMarkerConfigurer().apply {
        setTemplateLoaderPath("classpath:/templates")
    }
}
```

You can also plug in any `ViewResolver` implementation, as the following example shows:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        ViewResolver resolver = ... ;
        registry.viewResolver(resolver);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        val resolver: ViewResolver = ...
        registry.viewResolver(resolver
    }
}
```

To support Content Negotiation and rendering other formats through view resolution (besides HTML), you can configure one or more default views based on the `HttpMessageWriterView` implementation, which accepts any of the available Codecs from `spring-web`. The following example shows how to do so:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {


    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.freeMarker();

        Jackson2JsonEncoder encoder = new Jackson2JsonEncoder();
        registry.defaultViews(new HttpMessageWriterView(encoder));
    }

    // ...
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {


    override fun configureViewResolvers(registry: ViewResolverRegistry) {
        registry.freeMarker()

        val encoder = Jackson2JsonEncoder()
        registry.defaultViews(HttpMessageWriterView(encoder))
    }

    // ...
}
```

See View Technologies for more on the view technologies that are integrated with Spring WebFlux.

Static Resources

Web MVC

This option provides a convenient way to serve static resources from a list of `Resource`-based locations.

In the next example, given a request that starts with `/resources`, the relative path is used to find and serve static resources relative to `/static` on the classpath. Resources are served with a one-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The `Last-Modified` header is also evaluated and, if present, a `304` status code is returned. The following listing shows the example:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public", "classpath:/static/")
                .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS));
    }

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addResourceHandlers(registry: ResourceHandlerRegistry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public", "classpath:/static/")
                .setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS))
    }
}
```

See also HTTP caching support for static resources.

The resource handler also supports a chain of `ResourceResolver` implementations and `ResourceTransformer` implementations, which can be used to create a toolchain for working with optimized resources.

You can use the `VersionResourceResolver` for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other information. A `ContentVersionStrategy` (MD5 hash) is a good choice with some notable exceptions (such as JavaScript resources used with a module loader).

The following example shows how to use `VersionResourceResolver` in your Java configuration:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)
                .addResolver(new
VersionResourceResolver().addContentVersionStrategy("/**"));
    }

}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    override fun addResourceHandlers(registry: ResourceHandlerRegistry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public/")
                .resourceChain(true)

.addResolver(VersionResourceResolver().addContentVersionStrategy("/**"))
    }

}
```

You can use `ResourceUrlProvider` to rewrite URLs and apply the full chain of resolvers and transformers (for example, to insert versions). The WebFlux configuration provides a `ResourceUrlProvider` so that it can be injected into others.

Unlike Spring MVC, at present, in WebFlux, there is no way to transparently rewrite static resource URLs, since there are no view technologies that can make use of a non-blocking chain of resolvers and transformers. When serving only local resources, the workaround is to use `ResourceUrlProvider` directly (for example, through a custom element) and block.

Note that, when using both `EncodedResourceResolver` (for example, Gzip, Brotli encoded) and `VersionedResourceResolver`, they must be registered in that order, to ensure content-based versions are always computed reliably based on the unencoded file.

For WebJars, versioned URLs like `/webjars/jquery/1.2.0/jquery.min.js` are the recommended and most efficient way to use them. The related resource location is configured out of the box with Spring Boot (or can be configured manually via `ResourceHandlerRegistry`) and does not require to add the `org.webjars:webjars-locator-core` dependency.

Version-less URLs like `/webjars/jquery/jquery.min.js` are supported through the `WebJarsResourceResolver` which is automatically registered when the `org.webjars:webjars-locator-core` library is present on the classpath, at the cost of a classpath scanning that could slow down application startup. The resolver can re-write URLs to include the version of the jar and can also match against incoming URLs without versions — for example, from `/webjars/jquery/jquery.min.js` to `/webjars/jquery/1.2.0/jquery.min.js`.

The Java configuration based on `ResourceHandlerRegistry` provides further options for fine-grained control, e.g. last-modified behavior and optimized resource resolution.

Path Matching

Web MVC

You can customize options related to path matching. For details on the individual options, see the `PathMatchConfigurer` javadoc. The following example shows how to use `PathMatchConfigurer`:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseCaseSensitiveMatch(true)
            .addPathPrefix("/api",
HandlerTypePredicate.forAnnotation(RestController.class));
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    @Override
    fun configurePathMatch(configurer: PathMatchConfigurer) {
        configurer
            .setUseCaseSensitiveMatch(true)
            .addPathPrefix("/api",
HandlerTypePredicate.forAnnotation(RestController::class.java))
    }
}
```

Spring WebFlux relies on a parsed representation of the request path called `RequestPath` for access to decoded path segment values, with semicolon content removed (that is, path or matrix variables). That means, unlike in Spring MVC, you need not indicate whether to decode the request path nor whether to remove semicolon content for path matching purposes.

Spring WebFlux also does not support suffix pattern matching, unlike in Spring MVC, where we are also recommend moving away from reliance on it.

WebSocketService

The WebFlux Java config declares of a `WebSocketHandlerAdapter` bean which provides support for the invocation of WebSocket handlers. That means all that remains to do in order to handle a WebSocket handshake request is to map a `WebSocketHandler` to a URL via `SimpleUrlHandlerMapping`.

In some cases it may be necessary to create the `WebSocketHandlerAdapter` bean with a provided

`WebSocketService` service which allows configuring WebSocket server properties. For example:

Java

```
@Configuration
@EnableWebFlux
public class WebConfig implements WebFluxConfigurer {

    @Override
    public WebSocketService getWebSocketService() {
        TomcatRequestUpgradeStrategy strategy = new TomcatRequestUpgradeStrategy();
        strategy.setMaxSessionIdleTimeout(0L);
        return new HandshakeWebSocketService(strategy);
    }
}
```

Kotlin

```
@Configuration
@EnableWebFlux
class WebConfig : WebFluxConfigurer {

    @Override
    fun webSocketService(): WebSocketService {
        val strategy = TomcatRequestUpgradeStrategy().apply {
            setMaxSessionIdleTimeout(0L)
        }
        return HandshakeWebSocketService(strategy)
    }
}
```

Advanced Configuration Mode

Web MVC

`@EnableWebFlux` imports `DelegatingWebFluxConfiguration` that:

- Provides default Spring configuration for WebFlux applications
- detects and delegates to `WebFluxConfigurer` implementations to customize that configuration.

For advanced mode, you can remove `@EnableWebFlux` and extend directly from `DelegatingWebFluxConfiguration` instead of implementing `WebFluxConfigurer`, as the following example shows:

Java

```
@Configuration
public class WebConfig extends DelegatingWebFluxConfiguration {

    // ...
}
```

Kotlin

```
@Configuration
class WebConfig : DelegatingWebFluxConfiguration {

    // ...
}
```

You can keep existing methods in `WebConfig`, but you can now also override bean declarations from the base class and still have any number of other `WebMvcConfigurer` implementations on the classpath.

6.1.13. HTTP/2

Web MVC

HTTP/2 is supported with Reactor Netty, Tomcat, Jetty, and Undertow. However, there are considerations related to server configuration. For more details, see the HTTP/2 wiki page.

6.2. WebClient

Spring WebFlux includes a client to perform HTTP requests with. `WebClient` has a functional, fluent API based on Reactor, see Reactive Libraries, which enables declarative composition of asynchronous logic without the need to deal with threads or concurrency. It is fully non-blocking, it supports streaming, and relies on the same codecs that are also used to encode and decode request and response content on the server side.

`WebClient` needs an HTTP client library to perform requests with. There is built-in support for the following:

- Reactor Netty
- JDK HttpClient
- Jetty Reactive HttpClient
- Apache HttpComponents
- Others can be plugged via `ClientHttpConnector`.

6.2.1. Configuration

The simplest way to create a `WebClient` is through one of the static factory methods:

- `WebClient.create()`
- `WebClient.create(String baseUrl)`

You can also use `WebClient.builder()` with further options:

- `uriBuilderFactory`: Customized `UriBuilderFactory` to use as a base URL.
- `defaultUriVariables`: default values to use when expanding URI templates.
- `defaultHeader`: Headers for every request.
- `defaultCookie`: Cookies for every request.
- `defaultRequest`: `Consumer` to customize every request.
- `filter`: Client filter for every request.
- `exchangeStrategies`: HTTP message reader/writer customizations.
- `clientConnector`: HTTP client library settings.

For example:

Java

```
WebClient client = WebClient.builder()
        .codecs(configurer -> ... )
        .build();
```

Kotlin

```
val webClient = WebClient.builder()
        .codecs { configurer -> ... }
        .build()
```

Once built, a `WebClient` is immutable. However, you can clone it and build a modified copy as follows:

Java

```
WebClient client1 = WebClient.builder()
        .filter(filterA).filter(filterB).build();

WebClient client2 = client1.mutate()
        .filter(filterC).filter(filterD).build();

// client1 has filterA, filterB

// client2 has filterA, filterB, filterC, filterD
```

Kotlin

```
val client1 = WebClient.builder()
        .filter(filterA).filter(filterB).build()

val client2 = client1.mutate()
        .filter(filterC).filter(filterD).build()

// client1 has filterA, filterB

// client2 has filterA, filterB, filterC, filterD
```

MaxInMemorySize

Codecs have limits for buffering data in memory to avoid application memory issues. By default those are set to 256KB. If that's not enough you'll get the following error:

```
org.springframework.core.io.buffer.DataBufferLimitException: Exceeded limit on max
bytes to buffer
```

To change the limit for default codecs, use the following:

Java

```
WebClient webClient = WebClient.builder()
        .codecs(configurer -> configurer.defaultCodecs().maxInMemorySize(2 * 1024 *
1024))
        .build();
```

Kotlin

```
val webClient = WebClient.builder()
        .codecs { configurer -> configurer.defaultCodecs().maxInMemorySize(2 * 1024 *
1024) }
        .build()
```

Reactor Netty

To customize Reactor Netty settings, provide a pre-configured `HttpClient`:

Java

```
HttpClient httpClient = HttpClient.create().secure(sslSpec -> ...);

WebClient webClient = WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(httpClient))
        .build();
```

Kotlin

```
val httpClient = HttpClient.create().secure { ... }

val webClient = WebClient.builder()
    .clientConnector(ReactorClientHttpConnector(httpClient))
    .build()
```

Resources

By default, `HttpClient` participates in the global Reactor Netty resources held in `reactor.netty.http.HttpResources`, including event loop threads and a connection pool. This is the recommended mode, since fixed, shared resources are preferred for event loop concurrency. In this mode global resources remain active until the process exits.

If the server is timed with the process, there is typically no need for an explicit shutdown. However, if the server can start or stop in-process (for example, a Spring MVC application deployed as a WAR), you can declare a Spring-managed bean of type `ReactorResourceFactory` with `globalResources=true` (the default) to ensure that the Reactor Netty global resources are shut down when the Spring `ApplicationContext` is closed, as the following example shows:

Java

```
@Bean
public ReactorResourceFactory reactorResourceFactory() {
    return new ReactorResourceFactory();
}
```

Kotlin

```
@Bean
fun reactorResourceFactory() = ReactorResourceFactory()
```

You can also choose not to participate in the global Reactor Netty resources. However, in this mode, the burden is on you to ensure that all Reactor Netty client and server instances use shared resources, as the following example shows:

Java

```
@Bean
public ReactorResourceFactory resourceFactory() {
    ReactorResourceFactory factory = new ReactorResourceFactory();
    factory.setUseGlobalResources(false); ①
    return factory;
}

@Bean
public WebClient webClient() {

    Function<HttpClient, HttpClient> mapper = client -> {
        // Further customizations...
    };

    ClientHttpConnector connector =
            new ReactorClientHttpConnector(resourceFactory(), mapper); ②

    return WebClient.builder().clientConnector(connector).build(); ③
}
```

① Create resources independent of global ones.

② Use the `ReactorClientHttpConnector` constructor with resource factory.

③ Plug the connector into the `WebClient.Builder`.

Kotlin

```
@Bean
fun resourceFactory() = ReactorResourceFactory().apply {
    isUseGlobalResources = false ①
}

@Bean
fun webClient(): WebClient {

    val mapper: (HttpClient) -> HttpClient = {
        // Further customizations...
    }

    val connector = ReactorClientHttpConnector(resourceFactory(), mapper) ②

    return WebClient.builder().clientConnector(connector).build() ③
}
```

① Create resources independent of global ones.

② Use the `ReactorClientHttpConnector` constructor with resource factory.

③ Plug the connector into the `WebClient.Builder`.

Timeouts

To configure a connection timeout:

Java

```
import io.netty.channel.ChannelOption;

HttpClient httpClient = HttpClient.create()
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000);

WebClient webClient = WebClient.builder()
        .clientConnector(new ReactorClientHttpConnector(httpClient))
        .build();
```

Kotlin

```
import io.netty.channel.ChannelOption

val httpClient = HttpClient.create()
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 10000);

val webClient = WebClient.builder()
        .clientConnector(ReactorClientHttpConnector(httpClient))
        .build();
```

To configure a read or write timeout:

Java

```
import io.netty.handler.timeout.ReadTimeoutHandler;
import io.netty.handler.timeout.WriteTimeoutHandler;

HttpClient httpClient = HttpClient.create()
        .doOnConnected(conn -> conn
                .addHandlerLast(new ReadTimeoutHandler(10))
                .addHandlerLast(new WriteTimeoutHandler(10)));

// Create WebClient...
```

Kotlin

```
import io.netty.handler.timeout.ReadTimeoutHandler
import io.netty.handler.timeout.WriteTimeoutHandler

val httpClient = HttpClient.create()
        .doOnConnected { conn -> conn
                .addHandlerLast(ReadTimeoutHandler(10))
                .addHandlerLast(WriteTimeoutHandler(10))
        }

// Create WebClient...
```

To configure a response timeout for all requests:

Java

```
HttpClient httpClient = HttpClient.create()
        .responseTimeout(Duration.ofSeconds(2));

// Create WebClient...
```

Kotlin

```
val httpClient = HttpClient.create()
        .responseTimeout(Duration.ofSeconds(2));

// Create WebClient...
```

To configure a response timeout for a specific request:

Java

```
WebClient.create().get()
        .uri("https://example.org/path")
        .httpRequest(httpRequest -> {
            HttpClientRequest reactorRequest = httpRequest.getNativeRequest();
            reactorRequest.responseTimeout(Duration.ofSeconds(2));
        })
        .retrieve()
        .bodyToMono(String.class);
```

Kotlin

```
WebClient.create().get()
        .uri("https://example.org/path")
        .httpRequest { httpRequest: ClientHttpRequest ->
            val reactorRequest = httpRequest.getNativeRequest<HttpClientRequest>()
            reactorRequest.responseTimeout(Duration.ofSeconds(2))
        }
        .retrieve()
        .bodyToMono(String::class.java)
```

JDK HttpClient

The following example shows how to customize the JDK `HttpClient`:

Java

```
HttpClient httpClient = HttpClient.newBuilder()
    .followRedirects(Redirect.NORMAL)
    .connectTimeout(Duration.ofSeconds(20))
    .build();

ClientHttpConnector connector =
        new JdkClientHttpConnector(httpClient, new DefaultDataBufferFactory());

WebClient webClient = WebClient.builder().clientConnector(connector).build();
```

Kotlin

```
val httpClient = HttpClient.newBuilder()
    .followRedirects(Redirect.NORMAL)
    .connectTimeout(Duration.ofSeconds(20))
    .build()

val connector = JdkClientHttpConnector(httpClient, DefaultDataBufferFactory())

val webClient = WebClient.builder().clientConnector(connector).build()
```

Jetty

The following example shows how to customize Jetty `HttpClient` settings:

Java

```
HttpClient httpClient = new HttpClient();
httpClient.setCookieStore(...);

WebClient webClient = WebClient.builder()
        .clientConnector(new JettyClientHttpConnector(httpClient))
        .build();
```

Kotlin

```
val httpClient = HttpClient()
httpClient.cookieStore = ...

val webClient = WebClient.builder()
        .clientConnector(JettyClientHttpConnector(httpClient))
        .build();
```

By default, `HttpClient` creates its own resources (`Executor`, `ByteBufferPool`, `Scheduler`), which remain active until the process exits or `stop()` is called.

You can share resources between multiple instances of the Jetty client (and server) and ensure that the resources are shut down when the Spring `ApplicationContext` is closed by declaring a Spring-managed bean of type `JettyResourceFactory`, as the following example shows:

Java

```
@Bean
public JettyResourceFactory resourceFactory() {
    return new JettyResourceFactory();
}

@Bean
public WebClient webClient() {

    HttpClient httpClient = new HttpClient();
    // Further customizations...

    ClientHttpConnector connector =
            new JettyClientHttpConnector(httpClient, resourceFactory()); ①

    return WebClient.builder().clientConnector(connector).build(); ②
}
```

① Use the `JettyClientHttpConnector` constructor with resource factory.

② Plug the connector into the `WebClient.Builder`.

Kotlin

```
@Bean
fun resourceFactory() = JettyResourceFactory()

@Bean
fun webClient(): WebClient {

    val httpClient = HttpClient()
    // Further customizations...

    val connector = JettyClientHttpConnector(httpClient, resourceFactory()) ①

    return WebClient.builder().clientConnector(connector).build() ②
}
```

① Use the `JettyClientHttpConnector` constructor with resource factory.

② Plug the connector into the `WebClient.Builder`.

HttpComponents

The following example shows how to customize Apache HttpComponents `HttpClient` settings:

Java

```
HttpAsyncClientBuilder clientBuilder = HttpAsyncClients.custom();
clientBuilder.setDefaultRequestConfig(...);
CloseableHttpAsyncClient client = clientBuilder.build();

ClientHttpConnector connector = new HttpComponentsClientHttpConnector(client);

WebClient webClient = WebClient.builder().clientConnector(connector).build();
```

Kotlin

```
val client = HttpAsyncClients.custom().apply {
    setDefaultRequestConfig(...)
}.build()
val connector = HttpComponentsClientHttpConnector(client)
val webClient = WebClient.builder().clientConnector(connector).build()
```

6.2.2. `retrieve()`

The `retrieve()` method can be used to declare how to extract the response. For example:

Java

```
WebClient client = WebClient.create("https://example.org");

Mono<ResponseEntity<Person>> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .toEntity(Person.class);
```

Kotlin

```
val client = WebClient.create("https://example.org")

val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .toEntity<Person>().awaitSingle()
```

Or to get only the body:

Java

```
WebClient client = WebClient.create("https://example.org");

Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .bodyToMono(Person.class);
```

Kotlin

```
val client = WebClient.create("https://example.org")

val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .awaitBody<Person>()
```

To get a stream of decoded objects:

Java

```
Flux<Quote> result = client.get()
        .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve()
        .bodyToFlux(Quote.class);
```

Kotlin

```
val result = client.get()
        .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
        .retrieve()
        .bodyToFlow<Quote>()
```

By default, 4xx or 5xx responses result in an `WebClientResponseException`, including sub-classes for specific HTTP status codes. To customize the handling of error responses, use `onStatus` handlers as follows:

Java

```
Mono<Person> result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError, response -> ...)
        .onStatus(HttpStatus::is5xxServerError, response -> ...)
        .bodyToMono(Person.class);
```

Kotlin

```
val result = client.get()
        .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
        .retrieve()
        .onStatus(HttpStatus::is4xxClientError) { ... }
        .onStatus(HttpStatus::is5xxServerError) { ... }
        .awaitBody<Person>()
```

6.2.3. Exchange

The `exchangeToMono()` and `exchangeToFlux()` methods (or `awaitExchange { }` and `exchangeToFlow { }` in Kotlin) are useful for more advanced cases that require more control, such as to decode the response differently depending on the response status:

Java

```
Mono<Person> entityMono = client.get()
        .uri("/persons/1")
        .accept(MediaType.APPLICATION_JSON)
        .exchangeToMono(response -> {
            if (response.statusCode().equals(HttpStatus.OK)) {
                return response.bodyToMono(Person.class);
            }
            else {
                // Turn to error
                return response.createError();
            }
        });
```

Kotlin

```
val entity = client.get()
  .uri("/persons/1")
  .accept(MediaType.APPLICATION_JSON)
  .awaitExchange {
        if (response.statusCode() == HttpStatus.OK) {
             return response.awaitBody<Person>()
        }
        else {
             throw response.createExceptionAndAwait()
        }
  }
```

When using the above, after the returned `Mono` or `Flux` completes, the response body is checked and if not consumed it is released to prevent memory and connection leaks. Therefore the response cannot be decoded further downstream. It is up to the provided function to declare how to decode the response if needed.

6.2.4. Request Body

The request body can be encoded from any asynchronous type handled by `ReactiveAdapterRegistry`, like `Mono` or Kotlin Coroutines `Deferred` as the following example shows:

Java

```
Mono<Person> personMono = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body(personMono, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val personDeferred: Deferred<Person> = ...

client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body<Person>(personDeferred)
        .retrieve()
        .awaitBody<Unit>()
```

You can also have a stream of objects be encoded, as the following example shows:

Java

```
Flux<Person> personFlux = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_STREAM_JSON)
        .body(personFlux, Person.class)
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val people: Flow<Person> = ...

client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .body(people)
        .retrieve()
        .awaitBody<Unit>()
```

Alternatively, if you have the actual value, you can use the `bodyValue` shortcut method, as the following example shows:

Java

```
Person person = ... ;

Mono<Void> result = client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .bodyValue(person)
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val person: Person = ...

client.post()
        .uri("/persons/{id}", id)
        .contentType(MediaType.APPLICATION_JSON)
        .bodyValue(person)
        .retrieve()
        .awaitBody<Unit>()
```

Form Data

To send form data, you can provide a `MultiValueMap<String, String>` as the body. Note that the content is automatically set to `application/x-www-form-urlencoded` by the `FormHttpMessageWriter`. The following example shows how to use `MultiValueMap<String, String>`:

Java

```
MultiValueMap<String, String> formData = ... ;

Mono<Void> result = client.post()
        .uri("/path", id)
        .bodyValue(formData)
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val formData: MultiValueMap<String, String> = ...

client.post()
        .uri("/path", id)
        .bodyValue(formData)
        .retrieve()
        .awaitBody<Unit>()
```

You can also supply form data in-line by using `BodyInserters`, as the following example shows:

Java

```
import static org.springframework.web.reactive.function.BodyInserters.*;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(fromFormData("k1", "v1").with("k2", "v2"))
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
import org.springframework.web.reactive.function.BodyInserters.*

client.post()
        .uri("/path", id)
        .body(fromFormData("k1", "v1").with("k2", "v2"))
        .retrieve()
        .awaitBody<Unit>()
```

Multipart Data

To send multipart data, you need to provide a `MultiValueMap<String, ?>` whose values are either `Object` instances that represent part content or `HttpEntity` instances that represent the content and headers for a part. `MultipartBodyBuilder` provides a convenient API to prepare a multipart request. The following example shows how to create a `MultiValueMap<String, ?>`:

Java

```
MultipartBodyBuilder builder = new MultipartBodyBuilder();
builder.part("fieldPart", "fieldValue");
builder.part("filePart1", new FileSystemResource("...logo.png"));
builder.part("jsonPart", new Person("Jason"));
builder.part("myPart", part); // Part from a server request

MultiValueMap<String, HttpEntity<?>> parts = builder.build();
```

Kotlin

```
val builder = MultipartBodyBuilder().apply {
    part("fieldPart", "fieldValue")
    part("filePart1", FileSystemResource("...logo.png"))
    part("jsonPart", Person("Jason"))
    part("myPart", part) // Part from a server request
}

val parts = builder.build()
```

In most cases, you do not have to specify the `Content-Type` for each part. The content type is determined automatically based on the `HttpMessageWriter` chosen to serialize it or, in the case of a `Resource`, based on the file extension. If necessary, you can explicitly provide the `MediaType` to use for each part through one of the overloaded builder `part` methods.

Once a `MultiValueMap` is prepared, the easiest way to pass it to the `WebClient` is through the `body` method, as the following example shows:

Java

```
MultipartBodyBuilder builder = ...;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(builder.build())
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
val builder: MultipartBodyBuilder = ...

client.post()
        .uri("/path", id)
        .body(builder.build())
        .retrieve()
        .awaitBody<Unit>()
```

If the `MultiValueMap` contains at least one non-`String` value, which could also represent regular form data (that is, `application/x-www-form-urlencoded`), you need not set the `Content-Type` to `multipart/form-data`. This is always the case when using `MultipartBodyBuilder`, which ensures an `HttpEntity` wrapper.

As an alternative to `MultipartBodyBuilder`, you can also provide multipart content, inline-style, through the built-in `BodyInserters`, as the following example shows:

Java

```
import static org.springframework.web.reactive.function.BodyInserters.*;

Mono<Void> result = client.post()
        .uri("/path", id)
        .body(fromMultipartData("fieldPart", "value").with("filePart", resource))
        .retrieve()
        .bodyToMono(Void.class);
```

Kotlin

```
import org.springframework.web.reactive.function.BodyInserters.*

client.post()
        .uri("/path", id)
        .body(fromMultipartData("fieldPart", "value").with("filePart", resource))
        .retrieve()
        .awaitBody<Unit>()
```

`PartEvent`

To stream multipart data sequentially, you can provide multipart content through `PartEvent` objects.

- Form fields can be created via `FormPartEvent::create`.
- File uploads can be created via `FilePartEvent::create`.

You can concatenate the streams returned from methods via `Flux::concat`, and create a request for the `WebClient`.

For instance, this sample will POST a multipart form containing a form field and a file.

Java

```
Resource resource = ...
Mono<String> result = webClient
    .post()
    .uri("https://example.com")
    .body(Flux.concat(
            FormPartEvent.create("field", "field value"),
            FilePartEvent.create("file", resource)
    ), PartEvent.class)
    .retrieve()
    .bodyToMono(String.class);
```

Kotlin

```
var resource: Resource = ...
var result: Mono<String> = webClient
    .post()
    .uri("https://example.com")
    .body(
        Flux.concat(
            FormPartEvent.create("field", "field value"),
            FilePartEvent.create("file", resource)
        )
    )
    .retrieve()
    .bodyToMono()
```

On the server side, `PartEvent` objects that are received via `@RequestBody` or `ServerRequest::bodyToFlux(PartEvent.class)` can be relayed to another service via the `WebClient`.

6.2.5. Filters

You can register a client filter (`ExchangeFilterFunction`) through the `WebClient.Builder` in order to intercept and modify requests, as the following example shows:

Java

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {

            ClientRequest filtered = ClientRequest.from(request)
                    .header("foo", "bar")
                    .build();

            return next.exchange(filtered);
        })
        .build();
```

Kotlin

```
val client = WebClient.builder()
        .filter { request, next ->

            val filtered = ClientRequest.from(request)
                    .header("foo", "bar")
                    .build()

            next.exchange(filtered)
        }
        .build()
```

This can be used for cross-cutting concerns, such as authentication. The following example uses a filter for basic authentication through a static factory method:

Java

```
import static
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
cation;

WebClient client = WebClient.builder()
        .filter(basicAuthentication("user", "password"))
        .build();
```

Kotlin

```
import
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
cation

val client = WebClient.builder()
        .filter(basicAuthentication("user", "password"))
        .build()
```

Filters can be added or removed by mutating an existing `WebClient` instance, resulting in a new `WebClient` instance that does not affect the original one. For example:

Java

```
import static
org.springframework.web.reactive.function.client.ExchangeFilterFunctions.basicAuthenti
cation;

WebClient client = webClient.mutate()
        .filters(filterList -> {
            filterList.add(0, basicAuthentication("user", "password"));
        })
        .build();
```

Kotlin

```
val client = webClient.mutate()
        .filters { it.add(0, basicAuthentication("user", "password")) }
        .build()
```

`WebClient` is a thin facade around the chain of filters followed by an `ExchangeFunction`. It provides a workflow to make requests, to encode to and from higher level objects, and it helps to ensure that response content is always consumed. When filters handle the response in some way, extra care must be taken to always consume its content or to otherwise propagate it downstream to the `WebClient` which will ensure the same. Below is a filter that handles the `UNAUTHORIZED` status code but ensures that any response content, whether expected or not, is released:

Java

```
public ExchangeFilterFunction renewTokenFilter() {
    return (request, next) -> next.exchange(request).flatMap(response -> {
        if (response.statusCode().value() == HttpStatus.UNAUTHORIZED.value()) {
            return response.releaseBody()
                    .then(renewToken())
                    .flatMap(token -> {
                        ClientRequest newRequest =
ClientRequest.from(request).build();
                        return next.exchange(newRequest);
                    });
        } else {
            return Mono.just(response);
        }
    });
}
```

Kotlin

```
fun renewTokenFilter(): ExchangeFilterFunction? {
    return ExchangeFilterFunction { request: ClientRequest?, next: ExchangeFunction ->
        next.exchange(request!!).flatMap { response: ClientResponse ->
            if (response.statusCode().value() == HttpStatus.UNAUTHORIZED.value()) {
                return@flatMap response.releaseBody()
                        .then(renewToken())
                        .flatMap { token: String? ->
                            val newRequest = ClientRequest.from(request).build()
                            next.exchange(newRequest)
                        }
            } else {
                return@flatMap Mono.just(response)
            }
        }
    }
}
```

6.2.6. Attributes

You can add attributes to a request. This is convenient if you want to pass information through the filter chain and influence the behavior of filters for a given request. For example:

Java

```
WebClient client = WebClient.builder()
        .filter((request, next) -> {
            Optional<Object> usr = request.attribute("myAttribute");
            // ...
        })
        .build();

client.get().uri("https://example.org/")
        .attribute("myAttribute", "...")
        .retrieve()
        .bodyToMono(Void.class);

    }
```

Kotlin

```
val client = WebClient.builder()
        .filter { request, _ ->
            val usr = request.attributes()["myAttribute"];
            // ...
        }
        .build()

    client.get().uri("https://example.org/")
            .attribute("myAttribute", "...")
            .retrieve()
            .awaitBody<Unit>()
```

Note that you can configure a `defaultRequest` callback globally at the `WebClient.Builder` level which lets you insert attributes into all requests, which could be used for example in a Spring MVC application to populate request attributes based on `ThreadLocal` data.

6.2.7. Context

Attributes provide a convenient way to pass information to the filter chain but they only influence the current request. If you want to pass information that propagates to additional requests that are nested, e.g. via `flatMap`, or executed after, e.g. via `concatMap`, then you'll need to use the Reactor `Context`.

The Reactor `Context` needs to be populated at the end of a reactive chain in order to apply to all operations. For example:

Java

```
WebClient client = WebClient.builder()
        .filter((request, next) ->
                Mono.deferContextual(contextView -> {
                    String value = contextView.get("foo");
                    // ...
                }))
        .build();

client.get().uri("https://example.org/")
        .retrieve()
        .bodyToMono(String.class)
        .flatMap(body -> {
                // perform nested request (context propagates automatically)...
        })
        .contextWrite(context -> context.put("foo", ...));
```

6.2.8. Synchronous Use

`WebClient` can be used in synchronous style by blocking at the end for the result:

Java

```
Person person = client.get().uri("/person/{id}", i).retrieve()
    .bodyToMono(Person.class)
    .block();

List<Person> persons = client.get().uri("/persons").retrieve()
    .bodyToFlux(Person.class)
    .collectList()
    .block();
```

Kotlin

```
val person = runBlocking {
    client.get().uri("/person/{id}", i).retrieve()
            .awaitBody<Person>()
}

val persons = runBlocking {
    client.get().uri("/persons").retrieve()
            .bodyToFlow<Person>()
            .toList()
}
```

However if multiple calls need to be made, it's more efficient to avoid blocking on each response individually, and instead wait for the combined result:

Java

```
Mono<Person> personMono = client.get().uri("/person/{id}", personId)
        .retrieve().bodyToMono(Person.class);

Mono<List<Hobby>> hobbiesMono = client.get().uri("/person/{id}/hobbies", personId)
        .retrieve().bodyToFlux(Hobby.class).collectList();

Map<String, Object> data = Mono.zip(personMono, hobbiesMono, (person, hobbies) -> {
            Map<String, String> map = new LinkedHashMap<>();
            map.put("person", person);
            map.put("hobbies", hobbies);
            return map;
        })
        .block();
```

Kotlin

```
val data = runBlocking {
        val personDeferred = async {
            client.get().uri("/person/{id}", personId)
                    .retrieve().awaitBody<Person>()
        }

        val hobbiesDeferred = async {
            client.get().uri("/person/{id}/hobbies", personId)
                    .retrieve().bodyToFlow<Hobby>().toList()
        }

        mapOf("person" to personDeferred.await(), "hobbies" to
hobbiesDeferred.await())
    }
```

The above is merely one example. There are lots of other patterns and operators for putting together a reactive pipeline that makes many remote calls, potentially some nested, inter-dependent, without ever blocking until the end.

With `Flux` or `Mono`, you should never have to block in a Spring MVC or Spring WebFlux controller. Simply return the resulting reactive type from the controller method. The same principle apply to Kotlin Coroutines and Spring WebFlux, just use suspending function or return `Flow` in your controller method .

6.2.9. Testing

To test code that uses the `WebClient`, you can use a mock web server, such as the OkHttp MockWebServer. To see an example of its use, check out `WebClientIntegrationTests` in the Spring Framework test suite or the `static-server` sample in the OkHttp repository.

6.3. HTTP Interface Client

The Spring Frameworks lets you define an HTTP service as a Java interface with HTTP exchange methods. You can then generate a proxy that implements this interface and performs the exchanges. This helps to simplify HTTP remote access and provides additional flexibility for to choose an API style such as synchronous or reactive.

See REST Endpoints for details.

6.4. WebSockets

Same as in the Servlet stack

This part of the reference documentation covers support for reactive-stack WebSocket messaging.

= Introduction to WebSocket

The WebSocket protocol, RFC 6455, provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection. It is a different TCP protocol from HTTP but is designed to work over HTTP, using ports 80 and 443 and allowing re-use of existing firewall rules.

A WebSocket interaction begins with an HTTP request that uses the HTTP `Upgrade` header to upgrade or, in this case, to switch to the WebSocket protocol. The following example shows such an interaction:

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket ①
Connection: Upgrade ②
Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

① The `Upgrade` header.

② Using the `Upgrade` connection.

Instead of the usual 200 status code, a server with WebSocket support returns output similar to the following:

```
HTTP/1.1 101 Switching Protocols ①
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

① Protocol switch

After a successful handshake, the TCP socket underlying the HTTP upgrade request remains open for both the client and the server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. See RFC 6455, the WebSocket chapter of HTML5, or any of the many introductions and tutorials on the Web.

Note that, if a WebSocket server is running behind a web server (e.g. nginx), you likely need to configure it to pass WebSocket upgrade requests on to the WebSocket server. Likewise, if the application runs in a cloud environment, check the instructions of the cloud provider related to WebSocket support.

== HTTP Versus WebSocket

Even though WebSocket is designed to be HTTP-compatible and starts with an HTTP request, it is important to understand that the two protocols lead to very different architectures and application programming models.

In HTTP and REST, an application is modeled as many URLs. To interact with the application, clients access those URLs, request-response style. Servers route requests to the appropriate handler based on the HTTP URL, method, and headers.

By contrast, in WebSockets, there is usually only one URL for the initial connect. Subsequently, all application messages flow on that same TCP connection. This points to an entirely different asynchronous, event-driven, messaging architecture.

WebSocket is also a low-level transport protocol, which, unlike HTTP, does not prescribe any semantics to the content of messages. That means that there is no way to route or process a message unless the client and the server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (for example, STOMP), through the `Sec-WebSocket-Protocol` header on the HTTP handshake request. In the absence of that, they need to come up with their own conventions.

== When to Use WebSockets

WebSockets can make a web page be dynamic and interactive. However, in many cases, a combination of Ajax and HTTP streaming or long polling can provide a simple and effective solution.

For example, news, mail, and social feeds need to update dynamically, but it may be perfectly okay to do so every few minutes. Collaboration, games, and financial apps, on the other hand, need to be much closer to real-time.

Latency alone is not a deciding factor. If the volume of messages is relatively low (for example, monitoring network failures) HTTP streaming or polling can provide an effective solution. It is the combination of low latency, high frequency, and high volume that make the best case for the use of WebSocket.

Keep in mind also that over the Internet, restrictive proxies that are outside of your control may preclude WebSocket interactions, either because they are not configured to pass on the `Upgrade` header or because they close long-lived connections that appear idle. This means that the use of WebSocket for internal applications within the firewall is a more straightforward decision than it is for public facing applications.

6.4.1. WebSocket API

Same as in the Servlet stack

The Spring Framework provides a WebSocket API that you can use to write client- and server-side applications that handle WebSocket messages.

Server

Same as in the Servlet stack

To create a WebSocket server, you can first create a `WebSocketHandler`. The following example shows how to do so:

Java

```
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketSession;

public class MyWebSocketHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        // ...
    }
}
```

Kotlin

```
import org.springframework.web.reactive.socket.WebSocketHandler
import org.springframework.web.reactive.socket.WebSocketSession

class MyWebSocketHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {
        // ...
    }
}
```

Then you can map it to a URL:

Java

```
@Configuration
class WebConfig {

    @Bean
    public HandlerMapping handlerMapping() {
        Map<String, WebSocketHandler> map = new HashMap<>();
        map.put("/path", new MyWebSocketHandler());
        int order = -1; // before annotated controllers

        return new SimpleUrlHandlerMapping(map, order);
    }
}
```

Kotlin

```
@Configuration
class WebConfig {

    @Bean
    fun handlerMapping(): HandlerMapping {
        val map = mapOf("/path" to MyWebSocketHandler())
        val order = -1 // before annotated controllers

        return SimpleUrlHandlerMapping(map, order)
    }
}
```

If using the WebFlux Config there is nothing further to do, or otherwise if not using the WebFlux config you'll need to declare a `WebSocketHandlerAdapter` as shown below:

Java

```
@Configuration
class WebConfig {

    // ...

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter();
    }
}
```

Kotlin

```
@Configuration
class WebConfig {

    // ...

    @Bean
    fun handlerAdapter() =  WebSocketHandlerAdapter()
}
```

`WebSocketHandler`

The `handle` method of `WebSocketHandler` takes `WebSocketSession` and returns `Mono<Void>` to indicate when application handling of the session is complete. The session is handled through two streams, one for inbound and one for outbound messages. The following table describes the two methods that handle the streams:

`WebSocketSession` method	Description
`Flux<WebSocketMessage> receive()`	Provides access to the inbound message stream and completes when the connection is closed.
`Mono<Void> send(Publisher<WebSocketMessage>)`	Takes a source for outgoing messages, writes the messages, and returns a `Mono<Void>` that completes when the source completes and writing is done.

A `WebSocketHandler` must compose the inbound and outbound streams into a unified flow and return a `Mono<Void>` that reflects the completion of that flow. Depending on application requirements, the unified flow completes when:

- Either the inbound or the outbound message stream completes.
- The inbound stream completes (that is, the connection closed), while the outbound stream is infinite.
- At a chosen point, through the `close` method of `WebSocketSession`.

When inbound and outbound message streams are composed together, there is no need to check if the connection is open, since Reactive Streams signals end activity. The inbound stream receives a completion or error signal, and the outbound stream receives a cancellation signal.

The most basic implementation of a handler is one that handles the inbound stream. The following example shows such an implementation:

Java

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {
        return session.receive()            ①
                .doOnNext(message -> {
                    // ...                  ②
                })
                .concatMap(message -> {
                    // ...                  ③
                })
                .then();                    ④
    }
}
```

① Access the stream of inbound messages.

② Do something with each message.

③ Perform nested asynchronous operations that use the message content.

④ Return a `Mono<Void>` that completes when receiving completes.

Kotlin

```
class ExampleHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {
        return session.receive()            ①
                .doOnNext {
                    // ...                  ②
                }
                .concatMap {
                    // ...                  ③
                }
                .then()                     ④
    }
}
```

① Access the stream of inbound messages.

② Do something with each message.

③ Perform nested asynchronous operations that use the message content.

④ Return a `Mono<Void>` that completes when receiving completes.

For nested, asynchronous operations, you may need to call `message.retain()` on underlying servers that use pooled data buffers (for example, Netty). Otherwise, the data buffer may be released before you have had a chance to read the data. For more background, see Data Buffers and Codecs.

The following implementation combines the inbound and outbound streams:

Java

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Flux<WebSocketMessage> output = session.receive()              ①
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .map(value -> session.textMessage("Echo " + value));    ②

        return session.send(output);                                    ③
    }
}
```

① Handle the inbound message stream.

② Create the outbound message, producing a combined flow.

③ Return a `Mono<Void>` that does not complete while we continue to receive.

Kotlin

```
class ExampleHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {

        val output = session.receive()                          ①
                .doOnNext {
                    // ...
                }
                .concatMap {
                    // ...
                }
                .map { session.textMessage("Echo $it") }        ②

        return session.send(output)                             ③
    }
}
```

① Handle the inbound message stream.

② Create the outbound message, producing a combined flow.

③ Return a `Mono<Void>` that does not complete while we continue to receive.

Inbound and outbound streams can be independent and be joined only for completion, as the following example shows:

Java

```
class ExampleHandler implements WebSocketHandler {

    @Override
    public Mono<Void> handle(WebSocketSession session) {

        Mono<Void> input = session.receive()                                ①
                .doOnNext(message -> {
                    // ...
                })
                .concatMap(message -> {
                    // ...
                })
                .then();

        Flux<String> source = ... ;
        Mono<Void> output = session.send(source.map(session::textMessage)); ②

        return Mono.zip(input, output).then();                              ③
    }
}
```

① Handle inbound message stream.

② Send outgoing messages.

③ Join the streams and return a `Mono<Void>` that completes when either stream ends.

Kotlin

```
class ExampleHandler : WebSocketHandler {

    override fun handle(session: WebSocketSession): Mono<Void> {

        val input = session.receive()                                       ①
                .doOnNext {
                    // ...
                }
                .concatMap {
                    // ...
                }
                .then()

        val source: Flux<String> = ...
        val output = session.send(source.map(session::textMessage))         ②

        return Mono.zip(input, output).then()                               ③
    }
}
```

① Handle inbound message stream.

② Send outgoing messages.

③ Join the streams and return a `Mono<Void>` that completes when either stream ends.

`DataBuffer`

`DataBuffer` is the representation for a byte buffer in WebFlux. The Spring Core part of the reference has more on that in the section on Data Buffers and Codecs. The key point to understand is that on some servers like Netty, byte buffers are pooled and reference counted, and must be released when consumed to avoid memory leaks.

When running on Netty, applications must use `DataBufferUtils.retain(dataBuffer)` if they wish to hold on input data buffers in order to ensure they are not released, and subsequently use `DataBufferUtils.release(dataBuffer)` when the buffers are consumed.

Handshake

Same as in the Servlet stack

`WebSocketHandlerAdapter` delegates to a `WebSocketService`. By default, that is an instance of `HandshakeWebSocketService`, which performs basic checks on the WebSocket request and then uses `RequestUpgradeStrategy` for the server in use. Currently, there is built-in support for Reactor Netty, Tomcat, Jetty, and Undertow.

`HandshakeWebSocketService` exposes a `sessionAttributePredicate` property that allows setting a `Predicate<String>` to extract attributes from the `WebSession` and insert them into the attributes of the `WebSocketSession`.

Server Configuration

Same as in the Servlet stack

The `RequestUpgradeStrategy` for each server exposes configuration specific to the underlying WebSocket server engine. When using the WebFlux Java config you can customize such properties as shown in the corresponding section of the WebFlux Config, or otherwise if not using the WebFlux config, use the below:

Java

```
@Configuration
class WebConfig {

    @Bean
    public WebSocketHandlerAdapter handlerAdapter() {
        return new WebSocketHandlerAdapter(webSocketService());
    }

    @Bean
    public WebSocketService webSocketService() {
        TomcatRequestUpgradeStrategy strategy = new TomcatRequestUpgradeStrategy();
        strategy.setMaxSessionIdleTimeout(0L);
        return new HandshakeWebSocketService(strategy);
    }
}
```

Kotlin

```
@Configuration
class WebConfig {

    @Bean
    fun handlerAdapter() =
            WebSocketHandlerAdapter(webSocketService())

    @Bean
    fun webSocketService(): WebSocketService {
        val strategy = TomcatRequestUpgradeStrategy().apply {
            setMaxSessionIdleTimeout(0L)
        }
        return HandshakeWebSocketService(strategy)
    }
}
```

Check the upgrade strategy for your server to see what options are available. Currently, only Tomcat and Jetty expose such options.

CORS

Same as in the Servlet stack

The easiest way to configure CORS and restrict access to a WebSocket endpoint is to have your `WebSocketHandler` implement `CorsConfigurationSource` and return a `CorsConfiguration` with allowed origins, headers, and other details. If you cannot do that, you can also set the `corsConfigurations` property on the `SimpleUrlHandler` to specify CORS settings by URL pattern. If both are specified, they are combined by using the `combine` method on `CorsConfiguration`.

Client

Spring WebFlux provides a `WebSocketClient` abstraction with implementations for Reactor Netty, Tomcat, Jetty, Undertow, and standard Java (that is, JSR-356).

The Tomcat client is effectively an extension of the standard Java one with some extra functionality in the `WebSocketSession` handling to take advantage of the Tomcat-specific API to suspend receiving messages for back pressure.

To start a WebSocket session, you can create an instance of the client and use its `execute` methods:

Java

```
WebSocketClient client = new ReactorNettyWebSocketClient();

URI url = new URI("ws://localhost:8080/path");
client.execute(url, session ->
        session.receive()
                .doOnNext(System.out::println)
                .then());
```

Kotlin

```
val client = ReactorNettyWebSocketClient()

        val url = URI("ws://localhost:8080/path")
        client.execute(url) { session ->
            session.receive()
                    .doOnNext(::println)
            .then()
        }
```

Some clients, such as Jetty, implement `Lifecycle` and need to be stopped and started before you can use them. All clients have constructor options related to configuration of the underlying WebSocket client.

6.5. Testing

Same in Spring MVC

The `spring-test` module provides mock implementations of `ServerHttpRequest`, `ServerHttpResponse`, and `ServerWebExchange`. See Spring Web Reactive for a discussion of mock objects.

`WebTestClient` builds on these mock request and response objects to provide support for testing WebFlux applications without an HTTP server. You can use the `WebTestClient` for end-to-end integration tests, too.

6.6. RSocket

This section describes Spring Framework's support for the RSocket protocol.

6.6.1. Overview

RSocket is an application protocol for multiplexed, duplex communication over TCP, WebSocket, and other byte stream transports, using one of the following interaction models:

- `Request-Response` — send one message and receive one back.
- `Request-Stream` — send one message and receive a stream of messages back.
- `Channel` — send streams of messages in both directions.
- `Fire-and-Forget` — send a one-way message.

Once the initial connection is made, the "client" vs "server" distinction is lost as both sides become symmetrical and each side can initiate one of the above interactions. This is why in the protocol calls the participating sides "requester" and "responder" while the above interactions are called "request streams" or simply "requests".

These are the key features and benefits of the RSocket protocol:

- Reactive Streams semantics across network boundary — for streaming requests such as `Request-Stream` and `Channel`, back pressure signals travel between requester and responder, allowing a requester to slow down a responder at the source, hence reducing reliance on network layer congestion control, and the need for buffering at the network level or at any level.
- Request throttling — this feature is named "Leasing" after the `LEASE` frame that can be sent from each end to limit the total number of requests allowed by other end for a given time. Leases are renewed periodically.
- Session resumption — this is designed for loss of connectivity and requires some state to be maintained. The state management is transparent for applications, and works well in combination with back pressure which can stop a producer when possible and reduce the amount of state required.
- Fragmentation and re-assembly of large messages.
- Keepalive (heartbeats).

RSocket has implementations in multiple languages. The Java library is built on Project Reactor, and Reactor Netty for the transport. That means signals from Reactive Streams Publishers in your application propagate transparently through RSocket across the network.

The Protocol

One of the benefits of RSocket is that it has well defined behavior on the wire and an easy to read specification along with some protocol extensions. Therefore it is a good idea to read the spec, independent of language implementations and higher level framework APIs. This section provides a succinct overview to establish some context.

Connecting

Initially a client connects to a server via some low level streaming transport such as TCP or WebSocket and sends a `SETUP` frame to the server to set parameters for the connection.

The server may reject the `SETUP` frame, but generally after it is sent (for the client) and received (for the server), both sides can begin to make requests, unless `SETUP` indicates use of leasing semantics to limit the number of requests, in which case both sides must wait for a `LEASE` frame from the other end to permit making requests.

Making Requests

Once a connection is established, both sides may initiate a request through one of the frames `REQUEST_RESPONSE`, `REQUEST_STREAM`, `REQUEST_CHANNEL`, or `REQUEST_FNF`. Each of those frames carries one message from the requester to the responder.

The responder may then return `PAYLOAD` frames with response messages, and in the case of `REQUEST_CHANNEL` the requester may also send `PAYLOAD` frames with more request messages.

When a request involves a stream of messages such as `Request-Stream` and `Channel`, the responder must respect demand signals from the requester. Demand is expressed as a number of messages. Initial demand is specified in `REQUEST_STREAM` and `REQUEST_CHANNEL` frames. Subsequent demand is signaled via `REQUEST_N` frames.

Each side may also send metadata notifications, via the `METADATA_PUSH` frame, that do not pertain to any individual request but rather to the connection as a whole.

Message Format

RSocket messages contain data and metadata. Metadata can be used to send a route, a security token, etc. Data and metadata can be formatted differently. Mime types for each are declared in the `SETUP` frame and apply to all requests on a given connection.

While all messages can have metadata, typically metadata such as a route are per-request and therefore only included in the first message on a request, i.e. with one of the frames `REQUEST_RESPONSE`, `REQUEST_STREAM`, `REQUEST_CHANNEL`, or `REQUEST_FNF`.

Protocol extensions define common metadata formats for use in applications:

- Composite Metadata-- multiple, independently formatted metadata entries.
- Routing — the route for a request.

Java Implementation

The Java implementation for RSocket is built on Project Reactor. The transports for TCP and WebSocket are built on Reactor Netty. As a Reactive Streams library, Reactor simplifies the job of implementing the protocol. For applications it is a natural fit to use `Flux` and `Mono` with declarative operators and transparent back pressure support.

The API in RSocket Java is intentionally minimal and basic. It focuses on protocol features and leaves the application programming model (e.g. RPC codegen vs other) as a higher level,

independent concern.

The main contract io.rsocket.RSocket models the four request interaction types with `Mono` representing a promise for a single message, `Flux` a stream of messages, and `io.rsocket.Payload` the actual message with access to data and metadata as byte buffers. The `RSocket` contract is used symmetrically. For requesting, the application is given an `RSocket` to perform requests with. For responding, the application implements `RSocket` to handle requests.

This is not meant to be a thorough introduction. For the most part, Spring applications will not have to use its API directly. However it may be important to see or experiment with RSocket independent of Spring. The RSocket Java repository contains a number of sample apps that demonstrate its API and protocol features.

Spring Support

The `spring-messaging` module contains the following:

- RSocketRequester — fluent API to make requests through an `io.rsocket.RSocket` with data and metadata encoding/decoding.
- Annotated Responders — `@MessageMapping` annotated handler methods for responding.

The `spring-web` module contains `Encoder` and `Decoder` implementations such as Jackson CBOR/JSON, and Protobuf that RSocket applications will likely need. It also contains the `PathPatternParser` that can be plugged in for efficient route matching.

Spring Boot 2.2 supports standing up an RSocket server over TCP or WebSocket, including the option to expose RSocket over WebSocket in a WebFlux server. There is also client support and auto-configuration for an `RSocketRequester.Builder` and `RSocketStrategies`. See the RSocket section in the Spring Boot reference for more details.

Spring Security 5.2 provides RSocket support.

Spring Integration 5.2 provides inbound and outbound gateways to interact with RSocket clients and servers. See the Spring Integration Reference Manual for more details.

Spring Cloud Gateway supports RSocket connections.

6.6.2. RSocketRequester

`RSocketRequester` provides a fluent API to perform RSocket requests, accepting and returning objects for data and metadata instead of low level data buffers. It can be used symmetrically, to make requests from clients and to make requests from servers.

Client Requester

To obtain an `RSocketRequester` on the client side is to connect to a server which involves sending an RSocket `SETUP` frame with connection settings. `RSocketRequester` provides a builder that helps to prepare an `io.rsocket.core.RSocketConnector` including connection settings for the `SETUP` frame.

This is the most basic way to connect with default settings:

Java

```
RSocketRequester requester = RSocketRequester.builder().tcp("localhost", 7000);

URI url = URI.create("https://example.org:8080/rsocket");
RSocketRequester requester = RSocketRequester.builder().webSocket(url);
```

Kotlin

```
val requester = RSocketRequester.builder().tcp("localhost", 7000)

URI url = URI.create("https://example.org:8080/rsocket");
val requester = RSocketRequester.builder().webSocket(url)
```

The above does not connect immediately. When requests are made, a shared connection is established transparently and used.

Connection Setup

`RSocketRequester.Builder` provides the following to customize the initial `SETUP` frame:

- `dataMimeType(MimeType)` — set the mime type for data on the connection.
- `metadataMimeType(MimeType)` — set the mime type for metadata on the connection.
- `setupData(Object)` — data to include in the `SETUP`.
- `setupRoute(String, Object…)` — route in the metadata to include in the `SETUP`.
- `setupMetadata(Object, MimeType)` — other metadata to include in the `SETUP`.

For data, the default mime type is derived from the first configured `Decoder`. For metadata, the default mime type is composite metadata which allows multiple metadata value and mime type pairs per request. Typically both don't need to be changed.

Data and metadata in the `SETUP` frame is optional. On the server side, @ConnectMapping methods can be used to handle the start of a connection and the content of the `SETUP` frame. Metadata may be used for connection level security.

Strategies

`RSocketRequester.Builder` accepts `RSocketStrategies` to configure the requester. You'll need to use this to provide encoders and decoders for (de)-serialization of data and metadata values. By default only the basic codecs from `spring-core` for `String`, `byte[]`, and `ByteBuffer` are registered. Adding `spring-web` provides access to more that can be registered as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .encoders(encoders -> encoders.add(new Jackson2CborEncoder()))
    .decoders(decoders -> decoders.add(new Jackson2CborDecoder()))
    .build();

RSocketRequester requester = RSocketRequester.builder()
    .rsocketStrategies(strategies)
    .tcp("localhost", 7000);
```

Kotlin

```
val strategies = RSocketStrategies.builder()
        .encoders { it.add(Jackson2CborEncoder()) }
        .decoders { it.add(Jackson2CborDecoder()) }
        .build()

val requester = RSocketRequester.builder()
        .rsocketStrategies(strategies)
        .tcp("localhost", 7000)
```

`RSocketStrategies` is designed for re-use. In some scenarios, e.g. client and server in the same application, it may be preferable to declare it in Spring configuration.

Client Responders

`RSocketRequester.Builder` can be used to configure responders to requests from the server.

You can use annotated handlers for client-side responding based on the same infrastructure that's used on a server, but registered programmatically as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .routeMatcher(new PathPatternRouteMatcher())  ①
    .build();

SocketAcceptor responder =
    RSocketMessageHandler.responder(strategies, new ClientHandler()); ②

RSocketRequester requester = RSocketRequester.builder()
    .rsocketConnector(connector -> connector.acceptor(responder)) ③
    .tcp("localhost", 7000);
```

① Use `PathPatternRouteMatcher`, if `spring-web` is present, for efficient route matching.

② Create a responder from a class with `@MessageMapping` and/or `@ConnectMapping` methods.

③ Register the responder.

Kotlin

```
val strategies = RSocketStrategies.builder()
        .routeMatcher(PathPatternRouteMatcher())  ①
        .build()

val responder =
    RSocketMessageHandler.responder(strategies, new ClientHandler()); ②

val requester = RSocketRequester.builder()
        .rsocketConnector { it.acceptor(responder) } ③
        .tcp("localhost", 7000)
```

① Use `PathPatternRouteMatcher`, if `spring-web` is present, for efficient route matching.

② Create a responder from a class with `@MessageMapping` and/or `@ConnectMapping` methods.

③ Register the responder.

Note the above is only a shortcut designed for programmatic registration of client responders. For alternative scenarios, where client responders are in Spring configuration, you can still declare `RSocketMessageHandler` as a Spring bean and then apply as follows:

Java

```
ApplicationContext context = ... ;
RSocketMessageHandler handler = context.getBean(RSocketMessageHandler.class);

RSocketRequester requester = RSocketRequester.builder()
    .rsocketConnector(connector -> connector.acceptor(handler.responder()))
    .tcp("localhost", 7000);
```

Kotlin

```
import org.springframework.beans.factory.getBean

val context: ApplicationContext = ...
val handler = context.getBean<RSocketMessageHandler>()

val requester = RSocketRequester.builder()
        .rsocketConnector { it.acceptor(handler.responder()) }
        .tcp("localhost", 7000)
```

For the above you may also need to use `setHandlerPredicate` in `RSocketMessageHandler` to switch to a different strategy for detecting client responders, e.g. based on a custom annotation such as `@RSocketClientResponder` vs the default `@Controller`. This is necessary in scenarios with client and server, or multiple clients in the same application.

See also Annotated Responders, for more on the programming model.

Advanced

`RSocketRequesterBuilder` provides a callback to expose the underlying `io.rsocket.core.RSocketConnector` for further configuration options for keepalive intervals, session resumption, interceptors, and more. You can configure options at that level as follows:

Java

```
RSocketRequester requester = RSocketRequester.builder()
    .rsocketConnector(connector -> {
        // ...
    })
    .tcp("localhost", 7000);
```

Kotlin

```
val requester = RSocketRequester.builder()
        .rsocketConnector {
            //...
        }
        .tcp("localhost", 7000)
```

Server Requester

To make requests from a server to connected clients is a matter of obtaining the requester for the connected client from the server.

In Annotated Responders, `@ConnectMapping` and `@MessageMapping` methods support an `RSocketRequester` argument. Use it to access the requester for the connection. Keep in mind that `@ConnectMapping` methods are essentially handlers of the `SETUP` frame which must be handled before requests can begin. Therefore, requests at the very start must be decoupled from handling. For example:

Java

```
@ConnectMapping
Mono<Void> handle(RSocketRequester requester) {
    requester.route("status").data("5")
        .retrieveFlux(StatusReport.class)
        .subscribe(bar -> { ①
            // ...
        });
    return ... ②
}
```

① Start the request asynchronously, independent from handling.

② Perform handling and return completion `Mono<Void>`.

Kotlin

```
@ConnectMapping
suspend fun handle(requester: RSocketRequester) {
    GlobalScope.launch {
        requester.route("status").data("5").retrieveFlow<StatusReport>().collect { ①
            // ...
        }
    }
    /// ... ②
}
```

① Start the request asynchronously, independent from handling.

② Perform handling in the suspending function.

Requests

Once you have a client or server requester, you can make requests as follows:

Java

```
ViewBox viewBox = ... ;

Flux<AirportLocation> locations = requester.route("locate.radars.within") ①
        .data(viewBox) ②
        .retrieveFlux(AirportLocation.class); ③
```

① Specify a route to include in the metadata of the request message.

② Provide data for the request message.

③ Declare the expected response.

Kotlin

```
val viewBox: ViewBox = ...

val locations = requester.route("locate.radars.within") ①
        .data(viewBox) ②
        .retrieveFlow<AirportLocation>() ③
```

① Specify a route to include in the metadata of the request message.

② Provide data for the request message.

③ Declare the expected response.

The interaction type is determined implicitly from the cardinality of the input and output. The above example is a `Request-Stream` because one value is sent and a stream of values is received. For the most part you don't need to think about this as long as the choice of input and output matches an RSocket interaction type and the types of input and output expected by the responder. The only example of an invalid combination is many-to-one.

The `data(Object)` method also accepts any Reactive Streams `Publisher`, including `Flux` and `Mono`, as well as any other producer of value(s) that is registered in the `ReactiveAdapterRegistry`. For a multi-value `Publisher` such as `Flux` which produces the same types of values, consider using one of the overloaded `data` methods to avoid having type checks and `Encoder` lookup on every element:

```
data(Object producer, Class<?> elementClass);
data(Object producer, ParameterizedTypeReference<?> elementTypeRef);
```

The `data(Object)` step is optional. Skip it for requests that don't send data:

Java

```
Mono<AirportLocation> location = requester.route("find.radar.EWR"))
    .retrieveMono(AirportLocation.class);
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveAndAwait

val location = requester.route("find.radar.EWR")
    .retrieveAndAwait<AirportLocation>()
```

Extra metadata values can be added if using composite metadata (the default) and if the values are supported by a registered `Encoder`. For example:

Java

```
String securityToken = ... ;
ViewBox viewBox = ... ;
MimeType mimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0");

Flux<AirportLocation> locations = requester.route("locate.radars.within")
        .metadata(securityToken, mimeType)
        .data(viewBox)
        .retrieveFlux(AirportLocation.class);
```

Kotlin

```
import org.springframework.messaging.rsocket.retrieveFlow

val requester: RSocketRequester = ...

val securityToken: String = ...
val viewBox: ViewBox = ...
val mimeType = MimeType.valueOf("message/x.rsocket.authentication.bearer.v0")

val locations = requester.route("locate.radars.within")
        .metadata(securityToken, mimeType)
        .data(viewBox)
        .retrieveFlow<AirportLocation>()
```

For `Fire-and-Forget` use the `send()` method that returns `Mono<Void>`. Note that the `Mono` indicates only that the message was successfully sent, and not that it was handled.

For `Metadata-Push` use the `sendMetadata()` method with a `Mono<Void>` return value.

6.6.3. Annotated Responders

RSocket responders can be implemented as `@MessageMapping` and `@ConnectMapping` methods. `@MessageMapping` methods handle individual requests while `@ConnectMapping` methods handle connection-level events (setup and metadata push). Annotated responders are supported symmetrically, for responding from the server side and for responding from the client side.

Server Responders

To use annotated responders on the server side, add `RSocketMessageHandler` to your Spring configuration to detect `@Controller` beans with `@MessageMapping` and `@ConnectMapping` methods:

Java

```
@Configuration
static class ServerConfig {

    @Bean
    public RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler handler = new RSocketMessageHandler();
        handler.routeMatcher(new PathPatternRouteMatcher());
        return handler;
    }
}
```

Kotlin

```
@Configuration
class ServerConfig {

    @Bean
    fun rsocketMessageHandler() = RSocketMessageHandler().apply {
        routeMatcher = PathPatternRouteMatcher()
    }
}
```

Then start an RSocket server through the Java RSocket API and plug the `RSocketMessageHandler` for the responder as follows:

Java

```
ApplicationContext context = ... ;
RSocketMessageHandler handler = context.getBean(RSocketMessageHandler.class);

CloseableChannel server =
    RSocketServer.create(handler.responder())
        .bind(TcpServerTransport.create("localhost", 7000))
        .block();
```

Kotlin

```
import org.springframework.beans.factory.getBean

val context: ApplicationContext = ...
val handler = context.getBean<RSocketMessageHandler>()

val server = RSocketServer.create(handler.responder())
        .bind(TcpServerTransport.create("localhost", 7000))
        .awaitSingle()
```

`RSocketMessageHandler` supports composite and routing metadata by default. You can set its MetadataExtractor if you need to switch to a different mime type or register additional metadata mime types.

You'll need to set the `Encoder` and `Decoder` instances required for metadata and data formats to support. You'll likely need the `spring-web` module for codec implementations.

By default `SimpleRouteMatcher` is used for matching routes via `AntPathMatcher`. We recommend plugging in the `PathPatternRouteMatcher` from `spring-web` for efficient route matching. RSocket routes can be hierarchical but are not URL paths. Both route matchers are configured to use "." as separator by default and there is no URL decoding as with HTTP URLs.

`RSocketMessageHandler` can be configured via `RSocketStrategies` which may be useful if you need to share configuration between a client and a server in the same process:

Java

```
@Configuration
static class ServerConfig {

    @Bean
    public RSocketMessageHandler rsocketMessageHandler() {
        RSocketMessageHandler handler = new RSocketMessageHandler();
        handler.setRSocketStrategies(rsocketStrategies());
        return handler;
    }

    @Bean
    public RSocketStrategies rsocketStrategies() {
        return RSocketStrategies.builder()
            .encoders(encoders -> encoders.add(new Jackson2CborEncoder()))
            .decoders(decoders -> decoders.add(new Jackson2CborDecoder()))
            .routeMatcher(new PathPatternRouteMatcher())
            .build();
    }
}
```

Kotlin

```
@Configuration
class ServerConfig {

    @Bean
    fun rsocketMessageHandler() = RSocketMessageHandler().apply {
        rSocketStrategies = rsocketStrategies()
    }

    @Bean
    fun rsocketStrategies() = RSocketStrategies.builder()
            .encoders { it.add(Jackson2CborEncoder()) }
            .decoders { it.add(Jackson2CborDecoder()) }
            .routeMatcher(PathPatternRouteMatcher())
            .build()
}
```

Client Responders

Annotated responders on the client side need to be configured in the `RSocketRequester.Builder`. For details, see Client Responders.

@MessageMapping

Once server or client responder configuration is in place, `@MessageMapping` methods can be used as follows:

Java

```
@Controller
public class RadarsController {

    @MessageMapping("locate.radars.within")
    public Flux<AirportLocation> radars(MapRequest request) {
        // ...
    }
}
```

Kotlin

```
@Controller
class RadarsController {

    @MessageMapping("locate.radars.within")
    fun radars(request: MapRequest): Flow<AirportLocation> {
        // ...
    }
}
```

The above `@MessageMapping` method responds to a Request-Stream interaction having the route "locate.radars.within". It supports a flexible method signature with the option to use the following method arguments:

Method Argument	Description
`@Payload`	The payload of the request. This can be a concrete value of asynchronous types like `Mono` or `Flux`. **Note:** Use of the annotation is optional. A method argument that is not a simple type and is not any of the other supported arguments, is assumed to be the expected payload.
`RSocketRequester`	Requester for making requests to the remote end.
`@DestinationVariable`	Value extracted from the route based on variables in the mapping pattern, e.g. `@MessageMapping("find.radar.{id}")`.
`@Header`	Metadata value registered for extraction as described in MetadataExtractor.
`@Headers Map<String, Object>`	All metadata values registered for extraction as described in MetadataExtractor.

The return value is expected to be one or more Objects to be serialized as response payloads. That can be asynchronous types like `Mono` or `Flux`, a concrete value, or either `void` or a no-value asynchronous type such as `Mono<Void>`.

The RSocket interaction type that an `@MessageMapping` method supports is determined from the

cardinality of the input (i.e. `@Payload` argument) and of the output, where cardinality means the following:

Cardinality	Description
1	Either an explicit value, or a single-value asynchronous type such as `Mono<T>`.
Many	A multi-value asynchronous type such as `Flux<T>`.
0	For input this means the method does not have an `@Payload` argument. For output this is `void` or a no-value asynchronous type such as `Mono<Void>`.

The table below shows all input and output cardinality combinations and the corresponding interaction type(s):

Input Cardinality	Output Cardinality	Interaction Types
0, 1	0	Fire-and-Forget, Request-Response
0, 1	1	Request-Response
0, 1	Many	Request-Stream
Many	0, 1, Many	Request-Channel

@ConnectMapping

`@ConnectMapping` handles the `SETUP` frame at the start of an RSocket connection, and any subsequent metadata push notifications through the `METADATA_PUSH` frame, i.e. `metadataPush(Payload)` in `io.rsocket.RSocket`.

`@ConnectMapping` methods support the same arguments as @MessageMapping but based on metadata and data from the `SETUP` and `METADATA_PUSH` frames. `@ConnectMapping` can have a pattern to narrow handling to specific connections that have a route in the metadata, or if no patterns are declared then all connections match.

`@ConnectMapping` methods cannot return data and must be declared with `void` or `Mono<Void>` as the return value. If handling returns an error for a new connection then the connection is rejected. Handling must not be held up to make requests to the `RSocketRequester` for the connection. See Server Requester for details.

6.6.4. MetadataExtractor

Responders must interpret metadata. Composite metadata allows independently formatted metadata values (e.g. for routing, security, tracing) each with its own mime type. Applications need a way to configure metadata mime types to support, and a way to access extracted values.

`MetadataExtractor` is a contract to take serialized metadata and return decoded name-value pairs that can then be accessed like headers by name, for example via `@Header` in annotated handler methods.

`DefaultMetadataExtractor` can be given `Decoder` instances to decode metadata. Out of the box it has built-in support for "message/x.rsocket.routing.v0" which it decodes to `String` and saves under the "route" key. For any other mime type you'll need to provide a `Decoder` and register the mime type as follows:

Java

```
DefaultMetadataExtractor extractor = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(fooMimeType, Foo.class, "foo");
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val extractor = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Foo>(fooMimeType, "foo")
```

Composite metadata works well to combine independent metadata values. However the requester might not support composite metadata, or may choose not to use it. For this, `DefaultMetadataExtractor` may needs custom logic to map the decoded value to the output map. Here is an example where JSON is used for metadata:

Java

```
DefaultMetadataExtractor extractor = new DefaultMetadataExtractor(metadataDecoders);
extractor.metadataToExtract(
    MimeType.valueOf("application/vnd.myapp.metadata+json"),
    new ParameterizedTypeReference<Map<String,String>>() {},
    (jsonMap, outputMap) -> {
        outputMap.putAll(jsonMap);
    });
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val extractor = DefaultMetadataExtractor(metadataDecoders)
extractor.metadataToExtract<Map<String,
String>>(MimeType.valueOf("application/vnd.myapp.metadata+json")) { jsonMap, outputMap
->
    outputMap.putAll(jsonMap)
}
```

When configuring `MetadataExtractor` through `RSocketStrategies`, you can let `RSocketStrategies.Builder` create the extractor with the configured decoders, and simply use a callback to customize registrations as follows:

Java

```
RSocketStrategies strategies = RSocketStrategies.builder()
    .metadataExtractorRegistry(registry -> {
        registry.metadataToExtract(fooMimeType, Foo.class, "foo");
        // ...
    })
    .build();
```

Kotlin

```
import org.springframework.messaging.rsocket.metadataToExtract

val strategies = RSocketStrategies.builder()
        .metadataExtractorRegistry { registry: MetadataExtractorRegistry ->
            registry.metadataToExtract<Foo>(fooMimeType, "foo")
            // ...
        }
        .build()
```

6.6.5. RSocket Interface

The Spring Framework lets you define an RSocket service as a Java interface with annotated methods for RSocket exchanges. You can then generate a proxy that implements this interface and performs the exchanges. This helps to simplify RSocket remote access by wrapping the use of the underlying RSocketRequester.

One, declare an interface with `@RSocketExchange` methods:

```
interface RadarService {

    @RSocketExchange("radars")
    Flux<AirportLocation> getRadars(@Payload MapRequest request);

    // more RSocket exchange methods...

}
```

Two, create a proxy that will perform the declared RSocket exchanges:

```
RSocketRequester requester = ... ;
RSocketServiceProxyFactory factory =
RSocketServiceProxyFactory.builder(requester).build();

RepositoryService service = factory.createClient(RadarService.class);
```

Method Parameters

Annotated, RSocket exchange methods support flexible method signatures with the following method parameters:

Method argument	Description
`@DestinationVariable`	Add a route variable to pass to `RSocketRequester` along with the route from the `@RSocketExchange` annotation in order to expand template placeholders in the route. This variable can be a String or any Object, which is then formatted via `toString()`.
`@Payload`	Set the input payload(s) for the request. This can be a concrete value, or any producer of values that can be adapted to a Reactive Streams `Publisher` via `ReactiveAdapterRegistry`
`Object`, if followed by `MimeType`	The value for a metadata entry in the input payload. This can be any `Object` as long as the next argument is the metadata entry `MimeType`. The value can be a concrete value or any producer of a single value that can be adapted to a Reactive Streams `Publisher` via `ReactiveAdapterRegistry`.
`MimeType`	The `MimeType` for a metadata entry. The preceding method argument is expected to be the metadata value.

Return Values

Annotated, RSocket exchange methods support return values that are concrete value(s), or any producer of value(s) that can be adapted to a Reactive Streams `Publisher` via `ReactiveAdapterRegistry`.

6.7. Reactive Libraries

`spring-webflux` depends on `reactor-core` and uses it internally to compose asynchronous logic and to provide Reactive Streams support. Generally, WebFlux APIs return `Flux` or `Mono` (since those are used internally) and leniently accept any Reactive Streams `Publisher` implementation as input. The use of `Flux` versus `Mono` is important, because it helps to express cardinality — for example, whether a single or multiple asynchronous values are expected, and that can be essential for making decisions (for example, when encoding or decoding HTTP messages).

For annotated controllers, WebFlux transparently adapts to the reactive library chosen by the application. This is done with the help of the `ReactiveAdapterRegistry`, which provides pluggable support for reactive library and other asynchronous types. The registry has built-in support for RxJava 3, Kotlin coroutines and SmallRye Mutiny, but you can register others, too.

For functional APIs (such as Functional Endpoints, the `WebClient`, and others), the general rules for WebFlux APIs apply — `Flux` and `Mono` as return values and a Reactive Streams `Publisher` as input. When a `Publisher`, whether custom or from another reactive library, is provided, it can be treated only as a stream with unknown semantics (0..N). If, however, the semantics are known, you can wrap it with `Flux` or `Mono.from(Publisher)` instead of passing the raw `Publisher`.

For example, given a `Publisher` that is not a `Mono`, the Jackson JSON message writer expects multiple values. If the media type implies an infinite stream (for example, `application/json+stream`), values are written and flushed individually. Otherwise, values are buffered into a list and rendered as a JSON array.

Chapter 7. Integration

This part of the reference documentation covers Spring Framework's integration with a number of technologies.

7.1. REST Clients

The Spring Framework provides the following choices for making calls to REST endpoints:

- `WebClient` - non-blocking, reactive client w fluent API.
- `RestTemplate` - synchronous client with template method API.
- HTTP Interface - annotated interface with generated, dynamic proxy implementation.

7.1.1. `WebClient`

`WebClient` is a non-blocking, reactive client to perform HTTP requests. It was introduced in 5.0 and offers an alternative to the `RestTemplate`, with support for synchronous, asynchronous, and streaming scenarios.

`WebClient` supports the following:

- Non-blocking I/O.
- Reactive Streams back pressure.
- High concurrency with fewer hardware resources.
- Functional-style, fluent API that takes advantage of Java 8 lambdas.
- Synchronous and asynchronous interactions.
- Streaming up to or streaming down from a server.

See WebClient for more details.

7.1.2. `RestTemplate`

The `RestTemplate` provides a higher level API over HTTP client libraries. It makes it easy to invoke REST endpoints in a single line. It exposes the following groups of overloaded methods:

`RestTemplate` is in maintenance mode, with only requests for minor changes and bugs to be accepted. Please, consider using the WebClient instead.

Table 27. RestTemplate methods

Method group	Description
`getForObject`	Retrieves a representation via GET.
`getForEntity`	Retrieves a `ResponseEntity` (that is, status, headers, and body) by using GET.

Method group	Description
`headForHeaders`	Retrieves all headers for a resource by using HEAD.
`postForLocation`	Creates a new resource by using POST and returns the `Location` header from the response.
`postForObject`	Creates a new resource by using POST and returns the representation from the response.
`postForEntity`	Creates a new resource by using POST and returns the representation from the response.
`put`	Creates or updates a resource by using PUT.
`patchForObject`	Updates a resource by using PATCH and returns the representation from the response. Note that the JDK `HttpURLConnection` does not support `PATCH`, but Apache HttpComponents and others do.
`delete`	Deletes the resources at the specified URI by using DELETE.
`optionsForAllow`	Retrieves allowed HTTP methods for a resource by using ALLOW.
`exchange`	More generalized (and less opinionated) version of the preceding methods that provides extra flexibility when needed. It accepts a `RequestEntity` (including HTTP method, URL, headers, and body as input) and returns a `ResponseEntity`. These methods allow the use of `ParameterizedTypeReference` instead of `Class` to specify a response type with generics.
`execute`	The most generalized way to perform a request, with full control over request preparation and response extraction through callback interfaces.

Initialization

The default constructor uses `java.net.HttpURLConnection` to perform requests. You can switch to a different HTTP library with an implementation of `ClientHttpRequestFactory`. There is built-in support for the following:

- Apache HttpComponents
- Netty
- OkHttp

For example, to switch to Apache HttpComponents, you can use the following:

```
RestTemplate template = new RestTemplate(new
HttpComponentsClientHttpRequestFactory());
```

Each `ClientHttpRequestFactory` exposes configuration options specific to the underlying HTTP client library — for example, for credentials, connection pooling, and other details.

Note that the `java.net` implementation for HTTP requests can raise an exception when accessing the status of a response that represents an error (such as 401). If this is an issue, switch to another HTTP client library.

URIs

Many of the `RestTemplate` methods accept a URI template and URI template variables, either as a `String` variable argument, or as `Map<String,String>`.

The following example uses a `String` variable argument:

```
String result = restTemplate.getForObject(
        "https://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42",
"21");
```

The following example uses a `Map<String, String>`:

```
Map<String, String> vars = Collections.singletonMap("hotel", "42");

String result = restTemplate.getForObject(
        "https://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);
```

Keep in mind URI templates are automatically encoded, as the following example shows:

```
restTemplate.getForObject("https://example.com/hotel list", String.class);

// Results in request to "https://example.com/hotel%20list"
```

You can use the `uriTemplateHandler` property of `RestTemplate` to customize how URIs are encoded. Alternatively, you can prepare a `java.net.URI` and pass it into one of the `RestTemplate` methods that accepts a `URI`.

For more details on working with and encoding URIs, see URI Links.

Headers

You can use the `exchange()` methods to specify request headers, as the following example shows:

```
String uriTemplate = "https://example.com/hotels/{hotel}";
URI uri = UriComponentsBuilder.fromUriString(uriTemplate).build(42);

RequestEntity<Void> requestEntity = RequestEntity.get(uri)
        .header("MyRequestHeader", "MyValue")
        .build();

ResponseEntity<String> response = template.exchange(requestEntity, String.class);

String responseHeader = response.getHeaders().getFirst("MyResponseHeader");
String body = response.getBody();
```

You can obtain response headers through many `RestTemplate` method variants that return `ResponseEntity`.

Body

Objects passed into and returned from `RestTemplate` methods are converted to and from raw content with the help of an `HttpMessageConverter`.

On a POST, an input object is serialized to the request body, as the following example shows:

```
URI location = template.postForLocation("https://example.com/people", person);
```

You need not explicitly set the Content-Type header of the request. In most cases, you can find a compatible message converter based on the source `Object` type, and the chosen message converter sets the content type accordingly. If necessary, you can use the `exchange` methods to explicitly provide the `Content-Type` request header, and that, in turn, influences what message converter is selected.

On a GET, the body of the response is deserialized to an output `Object`, as the following example shows:

```
Person person = restTemplate.getForObject("https://example.com/people/{id}",
Person.class, 42);
```

The `Accept` header of the request does not need to be explicitly set. In most cases, a compatible message converter can be found based on the expected response type, which then helps to populate the `Accept` header. If necessary, you can use the `exchange` methods to provide the `Accept` header explicitly.

By default, `RestTemplate` registers all built-in message converters, depending on classpath checks that help to determine what optional conversion libraries are present. You can also set the message converters to use explicitly.

Message Conversion

WebFlux

The `spring-web` module contains the `HttpMessageConverter` contract for reading and writing the body of HTTP requests and responses through `InputStream` and `OutputStream`. `HttpMessageConverter` instances are used on the client side (for example, in the `RestTemplate`) and on the server side (for example, in Spring MVC REST controllers).

Concrete implementations for the main media (MIME) types are provided in the framework and are, by default, registered with the `RestTemplate` on the client side and with `RequestMappingHandlerAdapter` on the server side (see Configuring Message Converters).

The implementations of `HttpMessageConverter` are described in the following sections. For all converters, a default media type is used, but you can override it by setting the `supportedMediaTypes` bean property. The following table describes each implementation:

Table 28. HttpMessageConverter Implementations

MessageConverter	Description
`StringHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write `String` instances from the HTTP request and response. By default, this converter supports all text media types (`text/*`) and writes with a `Content-Type` of `text/plain`.
`FormHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write form data from the HTTP request and response. By default, this converter reads and writes the `application/x-www-form-urlencoded` media type. Form data is read from and written into a `MultiValueMap<String, String>`. The converter can also write (but not read) multipart data read from a `MultiValueMap<String, Object>`. By default, `multipart/form-data` is supported. As of Spring Framework 5.2, additional multipart subtypes can be supported for writing form data. Consult the javadoc for `FormHttpMessageConverter` for further details.
`ByteArrayHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write byte arrays from the HTTP request and response. By default, this converter supports all media types (`*/*`) and writes with a `Content-Type` of `application/octet-stream`. You can override this by setting the `supportedMediaTypes` property and overriding `getContentType(byte[])`.
`MarshallingHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write XML by using Spring's `Marshaller` and `Unmarshaller` abstractions from the `org.springframework.oxm` package. This converter requires a `Marshaller` and `Unmarshaller` before it can be used. You can inject these through constructor or bean properties. By default, this converter supports `text/xml` and `application/xml`.

MessageConverter	Description
`MappingJackson2HttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write JSON by using Jackson's `ObjectMapper`. You can customize JSON mapping as needed through the use of Jackson's provided annotations. When you need further control (for cases where custom JSON serializers/deserializers need to be provided for specific types), you can inject a custom `ObjectMapper` through the `ObjectMapper` property. By default, this converter supports `application/json`.
`MappingJackson2XmlHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write XML by using Jackson XML extension's `XmlMapper`. You can customize XML mapping as needed through the use of JAXB or Jackson's provided annotations. When you need further control (for cases where custom XML serializers/deserializers need to be provided for specific types), you can inject a custom `XmlMapper` through the `ObjectMapper` property. By default, this converter supports `application/xml`.
`SourceHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write `javax.xml.transform.Source` from the HTTP request and response. Only `DOMSource`, `SAXSource`, and `StreamSource` are supported. By default, this converter supports `text/xml` and `application/xml`.
`BufferedImageHttpMessageConverter`	An `HttpMessageConverter` implementation that can read and write `java.awt.image.BufferedImage` from the HTTP request and response. This converter reads and writes the media type supported by the Java I/O API.

Jackson JSON Views

You can specify a Jackson JSON View to serialize only a subset of the object properties, as the following example shows:

```
MappingJacksonValue value = new MappingJacksonValue(new User("eric", "7!jd#h23"));
value.setSerializationView(User.WithoutPasswordView.class);

RequestEntity<MappingJacksonValue> requestEntity =
    RequestEntity.post(new URI("https://example.com/user")).body(value);

ResponseEntity<String> response = template.exchange(requestEntity, String.class);
```

Multipart

To send multipart data, you need to provide a `MultiValueMap<String, Object>` whose values may be an `Object` for part content, a `Resource` for a file part, or an `HttpEntity` for part content with headers. For example:

```
MultiValueMap<String, Object> parts = new LinkedMultiValueMap<>();

parts.add("fieldPart", "fieldValue");
parts.add("filePart", new FileSystemResource("...logo.png"));
parts.add("jsonPart", new Person("Jason"));

HttpHeaders headers = new HttpHeaders();
headers.setContentType(MediaType.APPLICATION_XML);
parts.add("xmlPart", new HttpEntity<>(myBean, headers));
```

In most cases, you do not have to specify the `Content-Type` for each part. The content type is determined automatically based on the `HttpMessageConverter` chosen to serialize it or, in the case of a `Resource` based on the file extension. If necessary, you can explicitly provide the `MediaType` with an `HttpEntity` wrapper.

Once the `MultiValueMap` is ready, you can pass it to the `RestTemplate`, as show below:

```
MultiValueMap<String, Object> parts = ...;
template.postForObject("https://example.com/upload", parts, Void.class);
```

If the `MultiValueMap` contains at least one non-`String` value, the `Content-Type` is set to `multipart/form-data` by the `FormHttpMessageConverter`. If the `MultiValueMap` has `String` values the `Content-Type` is defaulted to `application/x-www-form-urlencoded`. If necessary the `Content-Type` may also be set explicitly.

7.1.3. HTTP Interface

The Spring Framework lets you define an HTTP service as a Java interface with annotated methods for HTTP exchanges. You can then generate a proxy that implements this interface and performs the exchanges. This helps to simplify HTTP remote access which often involves a facade that wraps the details of using the underlying HTTP client.

One, declare an interface with `@HttpExchange` methods:

```
interface RepositoryService {

    @GetExchange("/repos/{owner}/{repo}")
    Repository getRepository(@PathVariable String owner, @PathVariable String repo);

    // more HTTP exchange methods...

}
```

Two, create a proxy that will perform the declared HTTP exchanges:

```
WebClient client = WebClient.builder().baseUrl("https://api.github.com/").build();
HttpServiceProxyFactory factory =
HttpServiceProxyFactory.builder(WebClientAdapter.forClient(client)).build();

RepositoryService service = factory.createClient(RepositoryService.class);
```

`@HttpExchange` is supported at the type level where it applies to all methods:

```
@HttpExchange(url = "/repos/{owner}/{repo}", accept =
"application/vnd.github.v3+json")
interface RepositoryService {

    @GetExchange
    Repository getRepository(@PathVariable String owner, @PathVariable String repo);

    @PatchExchange(contentType = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
    void updateRepository(@PathVariable String owner, @PathVariable String repo,
            @RequestParam String name, @RequestParam String description, @RequestParam
String homepage);

}
```

Method Parameters

Annotated, HTTP exchange methods support flexible method signatures with the following method parameters:

Method argument	Description
`URI`	Dynamically set the URL for the request, overriding the annotation's `url` attribute.
`HttpMethod`	Dynamically set the HTTP method for the request, overriding the annotation's `method` attribute
`@RequestHeader`	Add a request header or mutliple headers. The argument may be a `Map<String, ?>` or `MultiValueMap<String, ?>` with multiple headers, a `Collection<?>` of values, or an individual value. Type conversion is supported for non-String values.
`@PathVariable`	Add a variable for expand a placeholder in the request URL. The argument may be a `Map<String, ?>` with multiple variables, or an individual value. Type conversion is supported for non-String values.
`@RequestBody`	Provide the body of the request either as an Object to be serialized, or a Reactive Streams `Publisher` such as `Mono`, `Flux`, or any other async type supported through the configured `ReactiveAdapterRegistry`.

Method argument	Description
`@RequestParam`	Add a request parameter or mutliple parameters. The argument may be a `Map<String, ?>` or `MultiValueMap<String, ?>` with multiple parameters, a `Collection<?>` of values, or an individual value. Type conversion is supported for non-String values. When `"content-type"` is set to `"application/x-www-form-urlencoded"`, request parameters are encoded in the request body. Otherwise, they are added as URL query parameters.
`@RequestPart`	Add a request part, which may be a String (form field), `Resource` (file part), Object (entity to be encoded, e.g. as JSON), `HttpEntity` (part content and headers), a Spring `Part`, or Reactive Streams `Publisher` of any of the above.
`@CookieValue`	Add a cookie or mutliple cookies. The argument may be a `Map<String, ?>` or `MultiValueMap<String, ?>` with multiple cookies, a `Collection<?>` of values, or an individual value. Type conversion is supported for non-String values.

Return Values

Annotated, HTTP exchange methods support the following return values:

Method return value	Description
`void`, `Mono<Void>`	Perform the given request, and release the response content, if any.
`HttpHeaders`, `Mono<HttpHeaders>`	Perform the given request, release the response content, if any, and return the response headers.
`<T>`, `Mono<T>`	Perform the given request and decode the response content to the declared return type.
`<T>`, `Flux<T>`	Perform the given request and decode the response content to a stream of the declared element type.
`ResponseEntity<Void>`, `Mono<ResponseEntity<Void>>`	Perform the given request, and release the response content, if any, and return a `ResponseEntity` with the status and headers.
`ResponseEntity<T>`, `Mono<ResponseEntity<T>>`	Perform the given request, decode the response content to the declared return type, and return a `ResponseEntity` with the status, headers, and the decoded body.
`Mono<ResponseEntity<Flux<T>>`	Perform the given request, decode the response content to a stream of the declared element type, and return a `ResponseEntity` with the status, headers, and the decoded response body stream.

You can also use any other async or reactive types registered in the `ReactiveAdapterRegistry`.

Exception Handling

By default, `WebClient` raises `WebClientResponseException` for 4xx and 5xx HTTP status codes. To customize this, you can register a response status handler that applies to all responses performed through the client:

```
WebClient webClient = WebClient.builder()
        .defaultStatusHandler(HttpStatusCode::isError, resp -> ...)
        .build();

WebClientAdapter clientAdapter = WebClientAdapter.forClient(webClient);
HttpServiceProxyFactory factory = HttpServiceProxyFactory
        .builder(clientAdapter).build();
```

For more details and options, such as suppressing error status codes, see the Javadoc of `defaultStatusHandler` in `WebClient.Builder`.

7.2. JMS (Java Message Service)

Spring provides a JMS integration framework that simplifies the use of the JMS API in much the same way as Spring's integration does for the JDBC API.

JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages. The `JmsTemplate` class is used for message production and synchronous message reception. For asynchronous reception similar to Jakarta EE's message-driven bean style, Spring provides a number of message-listener containers that you can use to create Message-Driven POJOs (MDPs). Spring also provides a declarative way to create message listeners.

The `org.springframework.jms.core` package provides the core functionality for using JMS. It contains JMS template classes that simplify the use of the JMS by handling the creation and release of resources, much like the `JdbcTemplate` does for JDBC. The design principle common to Spring template classes is to provide helper methods to perform common operations and, for more sophisticated usage, delegate the essence of the processing task to user-implemented callback interfaces. The JMS template follows the same design. The classes offer various convenience methods for sending messages, consuming messages synchronously, and exposing the JMS session and message producer to the user.

The `org.springframework.jms.support` package provides `JMSException` translation functionality. The translation converts the checked `JMSException` hierarchy to a mirrored hierarchy of unchecked exceptions. If any provider-specific subclasses of the checked `jakarta.jms.JMSException` exist, this exception is wrapped in the unchecked `UncategorizedJmsException`.

The `org.springframework.jms.support.converter` package provides a `MessageConverter` abstraction to convert between Java objects and JMS messages.

The `org.springframework.jms.support.destination` package provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

The `org.springframework.jms.annotation` package provides the necessary infrastructure to support

annotation-driven listener endpoints by using `@JmsListener`.

The `org.springframework.jms.config` package provides the parser implementation for the `jms` namespace as well as the java config support to configure listener containers and create listener endpoints.

Finally, the `org.springframework.jms.connection` package provides an implementation of the `ConnectionFactory` suitable for use in standalone applications. It also contains an implementation of Spring's `PlatformTransactionManager` for JMS (the cunningly named `JmsTransactionManager`). This allows for seamless integration of JMS as a transactional resource into Spring's transaction management mechanisms.

As of Spring Framework 5, Spring's JMS package fully supports JMS 2.0 and requires the JMS 2.0 API to be present at runtime. We recommend the use of a JMS 2.0 compatible provider.

If you happen to use an older message broker in your system, you may try upgrading to a JMS 2.0 compatible driver for your existing broker generation. Alternatively, you may also try to run against a JMS 1.1 based driver, simply putting the JMS 2.0 API jar on the classpath but only using JMS 1.1 compatible API against your driver. Spring's JMS support adheres to JMS 1.1 conventions by default, so with corresponding configuration it does support such a scenario. However, please consider this for transition scenarios only.

7.2.1. Using Spring JMS

This section describes how to use Spring's JMS components.

Using `JmsTemplate`

The `JmsTemplate` class is the central class in the JMS core package. It simplifies the use of JMS, since it handles the creation and release of resources when sending or synchronously receiving messages.

Code that uses the `JmsTemplate` needs only to implement callback interfaces that give them a clearly defined high-level contract. The `MessageCreator` callback interface creates a message when given a `Session` provided by the calling code in `JmsTemplate`. To allow for more complex usage of the JMS API, `SessionCallback` provides the JMS session, and `ProducerCallback` exposes a `Session` and `MessageProducer` pair.

The JMS API exposes two types of send methods, one that takes delivery mode, priority, and time-to-live as Quality of Service (QOS) parameters and one that takes no QOS parameters and uses default values. Since `JmsTemplate` has many send methods, setting the QOS parameters have been exposed as bean properties to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set by using the `setReceiveTimeout` property.

Some JMS providers allow the setting of default QOS values administratively through the configuration of the `ConnectionFactory`. This has the effect that a call to a `MessageProducer` instance's `send` method (`send(Destination destination, Message message)`) uses different QOS default values

than those specified in the JMS specification. In order to provide consistent management of QOS values, the `JmsTemplate` must, therefore, be specifically enabled to use its own QOS values by setting the boolean property `isExplicitQosEnabled` to `true`.

For convenience, `JmsTemplate` also exposes a basic request-reply operation that allows for sending a message and waiting for a reply on a temporary queue that is created as part of the operation.

Instances of the `JmsTemplate` class are thread-safe, once configured. This is important, because it means that you can configure a single instance of a `JmsTemplate` and then safely inject this shared reference into multiple collaborators. To be clear, the `JmsTemplate` is stateful, in that it maintains a reference to a `ConnectionFactory`, but this state is not conversational state.

As of Spring Framework 4.1, `JmsMessagingTemplate` is built on top of `JmsTemplate` and provides an integration with the messaging abstraction — that is, `org.springframework.messaging.Message`. This lets you create the message to send in a generic manner.

Connections

The `JmsTemplate` requires a reference to a `ConnectionFactory`. The `ConnectionFactory` is part of the JMS specification and serves as the entry point for working with JMS. It is used by the client application as a factory to create connections with the JMS provider and encapsulates various configuration parameters, many of which are vendor-specific, such as SSL configuration options.

When using JMS inside an EJB, the vendor provides implementations of the JMS interfaces so that they can participate in declarative transaction management and perform pooling of connections and sessions. In order to use this implementation, Jakarta EE containers typically require that you declare a JMS connection factory as a `resource-ref` inside the EJB or servlet deployment descriptors. To ensure the use of these features with the `JmsTemplate` inside an EJB, the client application should ensure that it references the managed implementation of the `ConnectionFactory`.

Caching Messaging Resources

The standard API involves creating many intermediate objects. To send a message, the following 'API' walk is performed:

```
ConnectionFactory->Connection->Session->MessageProducer->send
```

Between the `ConnectionFactory` and the `Send` operation, three intermediate objects are created and destroyed. To optimize the resource usage and increase performance, Spring provides two implementations of `ConnectionFactory`.

Using `SingleConnectionFactory`

Spring provides an implementation of the `ConnectionFactory` interface, `SingleConnectionFactory`, that returns the same `Connection` on all `createConnection()` calls and ignores calls to `close()`. This is useful for testing and standalone environments so that the same connection can be used for multiple `JmsTemplate` calls that may span any number of transactions. `SingleConnectionFactory` takes a reference to a standard `ConnectionFactory` that would typically come from JNDI.

Using `CachingConnectionFactory`

The `CachingConnectionFactory` extends the functionality of `SingleConnectionFactory` and adds the caching of `Session`, `MessageProducer`, and `MessageConsumer` instances. The initial cache size is set to `1`. You can use the `sessionCacheSize` property to increase the number of cached sessions. Note that the number of actual cached sessions is more than that number, as sessions are cached based on their acknowledgment mode, so there can be up to four cached session instances (one for each acknowledgment mode) when `sessionCacheSize` is set to one. `MessageProducer` and `MessageConsumer` instances are cached within their owning session and also take into account the unique properties of the producers and consumers when caching. MessageProducers are cached based on their destination. MessageConsumers are cached based on a key composed of the destination, selector, noLocal delivery flag, and the durable subscription name (if creating durable consumers).

MessageProducers and MessageConsumers for temporary queues and topics (TemporaryQueue/TemporaryTopic) will never be cached. Unfortunately, WebLogic JMS happens to implement the temporary queue/topic interfaces on its regular destination implementation, mis-indicating that none of its destinations can be cached. Please use a different connection pool/cache on WebLogic, or customize `CachingConnectionFactory` for WebLogic purposes.

Destination Management

Destinations, as `ConnectionFactory` instances, are JMS administered objects that you can store and retrieve in JNDI. When configuring a Spring application context, you can use the JNDI `JndiObjectFactoryBean` factory class or `<jee:jndi-lookup>` to perform dependency injection on your object's references to JMS destinations. However, this strategy is often cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the JMS provider. Examples of such advanced destination management include the creation of dynamic destinations or support for a hierarchical namespace of destinations. The `JmsTemplate` delegates the resolution of a destination name to a JMS destination object that implements the `DestinationResolver` interface. `DynamicDestinationResolver` is the default implementation used by `JmsTemplate` and accommodates resolving dynamic destinations. A `JndiDestinationResolver` is also provided to act as a service locator for destinations contained in JNDI and optionally falls back to the behavior contained in `DynamicDestinationResolver`.

Quite often, the destinations used in a JMS application are only known at runtime and, therefore, cannot be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well-known naming convention. Even though the creation of dynamic destinations is not part of the JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a user-defined name, which differentiates them from temporary destinations, and are often not registered in JNDI. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor-specific. However, a simple implementation choice that is sometimes made by vendors is to disregard the warnings in the JMS specification and to use the method `TopicSession createTopic(String topicName)` or the `QueueSession createQueue(String queueName)` method to create a new destination with default destination properties. Depending on the vendor implementation, `DynamicDestinationResolver` can then also create a physical destination instead of only resolving

one.

The boolean property `pubSubDomain` is used to configure the `JmsTemplate` with knowledge of what JMS domain is being used. By default, the value of this property is false, indicating that the point-to-point domain, `Queues`, is to be used. This property (used by `JmsTemplate`) determines the behavior of dynamic destination resolution through implementations of the `DestinationResolver` interface.

You can also configure the `JmsTemplate` with a default destination through the property `defaultDestination`. The default destination is with send and receive operations that do not refer to a specific destination.

Message Listener Containers

One of the most common uses of JMS messages in the EJB world is to drive message-driven beans (MDBs). Spring offers a solution to create message-driven POJOs (MDPs) in a way that does not tie a user to an EJB container. (See Asynchronous reception: Message-Driven POJOs for detailed coverage of Spring's MDP support.) Since Spring Framework 4.1, endpoint methods can be annotated with `@JmsListener` — see Annotation-driven Listener Endpoints for more details.

A message listener container is used to receive messages from a JMS message queue and drive the `MessageListener` that is injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an MDP and a messaging provider and takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion, and so on. This lets you write the (possibly complex) business logic associated with receiving a message (and possibly respond to it), and delegates boilerplate JMS infrastructure concerns to the framework.

There are two standard JMS message listener containers packaged with Spring, each with its specialized feature set.

- `SimpleMessageListenerContainer`
- `DefaultMessageListenerContainer`

Using `SimpleMessageListenerContainer`

This message listener container is the simpler of the two standard flavors. It creates a fixed number of JMS sessions and consumers at startup, registers the listener by using the standard JMS `MessageConsumer.setMessageListener()` method, and leaves it up the JMS provider to perform listener callbacks. This variant does not allow for dynamic adaption to runtime demands or for participation in externally managed transactions. Compatibility-wise, it stays very close to the spirit of the standalone JMS specification, but is generally not compatible with Jakarta EE's JMS restrictions.

While `SimpleMessageListenerContainer` does not allow for participation in externally managed transactions, it does support native JMS transactions. To enable this feature, you can switch the `sessionTransacted` flag to `true` or, in the XML namespace, set the `acknowledge` attribute to `transacted`. Exceptions thrown from your listener then lead to a rollback, with the message getting redelivered. Alternatively, consider using `CLIENT_ACKNOWLEDGE` mode, which provides redelivery in case of an exception as well but does not use transacted `Session` instances and, therefore, does not include any other `Session` operations (such as sending response messages) in the transaction protocol.

The default `AUTO_ACKNOWLEDGE` mode does not provide proper reliability guarantees. Messages can get lost when listener execution fails (since the provider automatically acknowledges each message after listener invocation, with no exceptions to be propagated to the provider) or when the listener container shuts down (you can configure this by setting the `acceptMessagesWhileStopping` flag). Make sure to use transacted sessions in case of reliability needs (for example, for reliable queue handling and durable topic subscriptions).

Using `DefaultMessageListenerContainer`

This message listener container is used in most cases. In contrast to `SimpleMessageListenerContainer`, this container variant allows for dynamic adaptation to runtime demands and is able to participate in externally managed transactions. Each received message is registered with an XA transaction when configured with a `JtaTransactionManager`. As a result, processing may take advantage of XA transaction semantics. This listener container strikes a good balance between low requirements on the JMS provider, advanced functionality (such as participation in externally managed transactions), and compatibility with Jakarta EE environments.

You can customize the cache level of the container. Note that, when no caching is enabled, a new connection and a new session is created for each message reception. Combining this with a non-durable subscription with high loads may lead to message loss. Make sure to use a proper cache level in such a case.

This container also has recoverable capabilities when the broker goes down. By default, a simple `BackOff` implementation retries every five seconds. You can specify a custom `BackOff` implementation for more fine-grained recovery options. See `ExponentialBackOff` for an example.

Like its sibling (`SimpleMessageListenerContainer`), `DefaultMessageListenerContainer` supports native JMS transactions and allows for customizing the acknowledgment mode. If feasible for your scenario, This is strongly recommended over externally managed transactions — that is, if you can live with occasional duplicate messages in case of the JVM dying. Custom duplicate message detection steps in your business logic can cover such situations — for example, in the form of a business entity existence check or a protocol table check. Any such arrangements are significantly more efficient than the alternative: wrapping your entire processing with an XA transaction (through configuring your `DefaultMessageListenerContainer` with an `JtaTransactionManager`) to cover the reception of the JMS message as well as the execution of the business logic in your message listener (including database operations, etc.).

The default `AUTO_ACKNOWLEDGE` mode does not provide proper reliability guarantees. Messages can get lost when listener execution fails (since the provider automatically acknowledges each message after listener invocation, with no exceptions to be propagated to the provider) or when the listener container shuts down (you can configure this by setting the `acceptMessagesWhileStopping` flag). Make sure to use transacted sessions in case of reliability needs (for example, for reliable queue handling and durable topic subscriptions).

Transaction Management

Spring provides a `JmsTransactionManager` that manages transactions for a single JMS `ConnectionFactory`. This lets JMS applications leverage the managed-transaction features of Spring, as described in Transaction Management section of the Data Access chapter. The `JmsTransactionManager` performs local resource transactions, binding a JMS Connection/Session pair from the specified `ConnectionFactory` to the thread. `JmsTemplate` automatically detects such transactional resources and operates on them accordingly.

In a Jakarta EE environment, the `ConnectionFactory` pools Connection and Session instances, so those resources are efficiently reused across transactions. In a standalone environment, using Spring's `SingleConnectionFactory` result in a shared JMS `Connection`, with each transaction having its own independent `Session`. Alternatively, consider the use of a provider-specific pooling adapter, such as ActiveMQ's `PooledConnectionFactory` class.

You can also use `JmsTemplate` with the `JtaTransactionManager` and an XA-capable JMS `ConnectionFactory` to perform distributed transactions. Note that this requires the use of a JTA transaction manager as well as a properly XA-configured ConnectionFactory. (Check your Jakarta EE server's or JMS provider's documentation.)

Reusing code across a managed and unmanaged transactional environment can be confusing when using the JMS API to create a `Session` from a `Connection`. This is because the JMS API has only one factory method to create a `Session`, and it requires values for the transaction and acknowledgment modes. In a managed environment, setting these values is the responsibility of the environment's transactional infrastructure, so these values are ignored by the vendor's wrapper to the JMS Connection. When you use the `JmsTemplate` in an unmanaged environment, you can specify these values through the use of the properties `sessionTransacted` and `sessionAcknowledgeMode`. When you

use a `PlatformTransactionManager` with `JmsTemplate`, the template is always given a transactional JMS `Session`.

7.2.2. Sending a Message

The `JmsTemplate` contains many convenience methods to send a message. Send methods specify the destination by using a `jakarta.jms.Destination` object, and others specify the destination by using a `String` in a JNDI lookup. The `send` method that takes no destination argument uses the default destination.

The following example uses the `MessageCreator` callback to create a text message from the supplied `Session` object:

```
import jakarta.jms.ConnectionFactory;
import jakarta.jms.JMSException;
import jakarta.jms.Message;
import jakarta.jms.Queue;
import jakarta.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;

public class JmsQueueSender {

    private JmsTemplate jmsTemplate;
    private Queue queue;

    public void setConnectionFactory(ConnectionFactory cf) {
        this.jmsTemplate = new JmsTemplate(cf);
    }

    public void setQueue(Queue queue) {
        this.queue = queue;
    }

    public void simpleSend() {
        this.jmsTemplate.send(this.queue, new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage("hello queue world");
            }
        });
    }
}
```

In the preceding example, the `JmsTemplate` is constructed by passing a reference to a `ConnectionFactory`. As an alternative, a zero-argument constructor and `connectionFactory` is provided and can be used for constructing the instance in JavaBean style (using a `BeanFactory` or plain Java code). Alternatively, consider deriving from Spring's `JmsGatewaySupport` convenience base class, which provides pre-built bean properties for JMS configuration.

The `send(String destinationName, MessageCreator creator)` method lets you send a message by using the string name of the destination. If these names are registered in JNDI, you should set the `destinationResolver` property of the template to an instance of `JndiDestinationResolver`.

If you created the `JmsTemplate` and specified a default destination, the `send(MessageCreator c)` sends a message to that destination.

Using Message Converters

To facilitate the sending of domain model objects, the `JmsTemplate` has various send methods that take a Java object as an argument for a message's data content. The overloaded methods `convertAndSend()` and `receiveAndConvert()` methods in `JmsTemplate` delegate the conversion process to an instance of the `MessageConverter` interface. This interface defines a simple contract to convert between Java objects and JMS messages. The default implementation (`SimpleMessageConverter`) supports conversion between `String` and `TextMessage`, `byte[]` and `BytesMessage`, and `java.util.Map` and `MapMessage`. By using the converter, you and your application code can focus on the business object that is being sent or received through JMS and not be concerned with the details of how it is represented as a JMS message.

The sandbox currently includes a `MapMessageConverter`, which uses reflection to convert between a JavaBean and a `MapMessage`. Other popular implementation choices you might implement yourself are converters that use an existing XML marshalling package (such as JAXB or XStream) to create a `TextMessage` that represents the object.

To accommodate the setting of a message's properties, headers, and body that can not be generically encapsulated inside a converter class, the `MessagePostProcessor` interface gives you access to the message after it has been converted but before it is sent. The following example shows how to modify a message header and a property after a `java.util.Map` is converted to a message:

```
public void sendWithConversion() {
    Map map = new HashMap();
    map.put("Name", "Mark");
    map.put("Age", new Integer(47));
    jmsTemplate.convertAndSend("testQueue", map, new MessagePostProcessor() {
        public Message postProcessMessage(Message message) throws JMSException {
            message.setIntProperty("AccountID", 1234);
            message.setJMSCorrelationID("123-00001");
            return message;
        }
    });
}
```

This results in a message of the following form:

```
MapMessage={
    Header={
        ... standard headers ...
        CorrelationID={123-00001}
    }
    Properties={
        AccountID={Integer:1234}
    }
    Fields={
        Name={String:Mark}
        Age={Integer:47}
    }
}
```

Using `SessionCallback` and `ProducerCallback`

While the send operations cover many common usage scenarios, you might sometimes want to perform multiple operations on a JMS `Session` or `MessageProducer`. The `SessionCallback` and `ProducerCallback` expose the JMS `Session` and `Session` / `MessageProducer` pair, respectively. The `execute()` methods on `JmsTemplate` run these callback methods.

7.2.3. Receiving a Message

This describes how to receive messages with JMS in Spring.

Synchronous Reception

While JMS is typically associated with asynchronous processing, you can consume messages synchronously. The overloaded `receive(..)` methods provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation, since the calling thread can potentially be blocked indefinitely. The `receiveTimeout` property specifies how long the receiver should wait before giving up waiting for a message.

Asynchronous reception: Message-Driven POJOs

Spring also supports annotated-listener endpoints through the use of the `@JmsListener` annotation and provides an open infrastructure to register endpoints programmatically. This is, by far, the most convenient way to setup an asynchronous receiver. See Enable Listener Endpoint Annotations for more details.

In a fashion similar to a Message-Driven Bean (MDB) in the EJB world, the Message-Driven POJO (MDP) acts as a receiver for JMS messages. The one restriction (but see Using `MessageListenerAdapter`) on an MDP is that it must implement the `jakarta.jms.MessageListener` interface. Note that, if your POJO receives messages on multiple threads, it is important to ensure that your implementation is thread-safe.

The following example shows a simple implementation of an MDP:

```
import jakarta.jms.JMSException;
import jakarta.jms.Message;
import jakarta.jms.MessageListener;
import jakarta.jms.TextMessage;

public class ExampleListener implements MessageListener {

    public void onMessage(Message message) {
        if (message instanceof TextMessage textMessage) {
            try {
                System.out.println(textMessage.getText());
            }
            catch (JMSException ex) {
                throw new RuntimeException(ex);
            }
        }
        else {
            throw new IllegalArgumentException("Message must be of type TextMessage");
        }
    }
}
```

Once you have implemented your `MessageListener`, it is time to create a message listener container.

The following example shows how to define and configure one of the message listener containers that ships with Spring (in this case, `DefaultMessageListenerContainer`):

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener"/>

<!-- and this is the message listener container -->
<bean id="jmsContainer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
</bean>
```

See the Spring javadoc of the various message listener containers (all of which implement MessageListenerContainer) for a full description of the features supported by each implementation.

Using the `SessionAwareMessageListener` Interface

The `SessionAwareMessageListener` interface is a Spring-specific interface that provides a similar contract to the JMS `MessageListener` interface but also gives the message-handling method access to the JMS `Session` from which the `Message` was received. The following listing shows the definition of

the SessionAwareMessageListener interface:

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

You can choose to have your MDPs implement this interface (in preference to the standard JMS MessageListener interface) if you want your MDPs to be able to respond to any received messages (by using the Session supplied in the onMessage(Message, Session) method). All of the message listener container implementations that ship with Spring have support for MDPs that implement either the MessageListener or SessionAwareMessageListener interface. Classes that implement the SessionAwareMessageListener come with the caveat that they are then tied to Spring through the interface. The choice of whether or not to use it is left entirely up to you as an application developer or architect.

Note that the onMessage(..) method of the SessionAwareMessageListener interface throws JMSException. In contrast to the standard JMS MessageListener interface, when using the SessionAwareMessageListener interface, it is the responsibility of the client code to handle any thrown exceptions.

Using MessageListenerAdapter

The MessageListenerAdapter class is the final component in Spring's asynchronous messaging support. In a nutshell, it lets you expose almost any class as an MDP (though there are some constraints).

Consider the following interface definition:

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

Notice that, although the interface extends neither the MessageListener nor the SessionAwareMessageListener interface, you can still use it as an MDP by using the MessageListenerAdapter class. Notice also how the various message handling methods are strongly typed according to the contents of the various Message types that they can receive and handle.

Now consider the following implementation of the MessageDelegate interface:

```
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

In particular, note how the preceding implementation of the `MessageDelegate` interface (the `DefaultMessageDelegate` class) has no JMS dependencies at all. It truly is a POJO that we can make into an MDP through the following configuration:

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener"
class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the message listener container... -->
<bean id="jmsContainer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
</bean>
```

The next example shows another MDP that can handle only receiving JMS `TextMessage` messages. Notice how the message handling method is actually called `receive` (the name of the message handling method in a `MessageListenerAdapter` defaults to `handleMessage`), but it is configurable (as you can see later in this section). Notice also how the `receive(..)` method is strongly typed to receive and respond only to JMS `TextMessage` messages. The following listing shows the definition of the `TextMessageDelegate` interface:

```
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

The following listing shows a class that implements the `TextMessageDelegate` interface:

```
public class DefaultTextMessageDelegate implements TextMessageDelegate {
    // implementation elided for clarity...
}
```

The configuration of the attendant `MessageListenerAdapter` would then be as follows:

```
<bean id="messageListener"
class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

Note that, if the `messageListener` receives a JMS `Message` of a type other than `TextMessage`, an `IllegalStateException` is thrown (and subsequently swallowed). Another of the capabilities of the `MessageListenerAdapter` class is the ability to automatically send back a response `Message` if a handler method returns a non-void value. Consider the following interface and class:

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);
}
```

```
public class DefaultResponsiveTextMessageDelegate implements
ResponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

If you use the `DefaultResponsiveTextMessageDelegate` in conjunction with a `MessageListenerAdapter`, any non-null value that is returned from the execution of the `'receive(..)'` method is (in the default configuration) converted into a `TextMessage`. The resulting `TextMessage` is then sent to the `Destination` (if one exists) defined in the JMS `Reply-To` property of the original `Message` or the default `Destination` set on the `MessageListenerAdapter` (if one has been configured). If no `Destination` is found, an `InvalidDestinationException` is thrown (note that this exception is not swallowed and propagates up the call stack).

Processing Messages Within Transactions

Invoking a message listener within a transaction requires only reconfiguration of the listener container.

You can activate local resource transactions through the `sessionTransacted` flag on the listener container definition. Each message listener invocation then operates within an active JMS transaction, with message reception rolled back in case of listener execution failure. Sending a response message (through `SessionAwareMessageListener`) is part of the same local transaction, but any other resource operations (such as database access) operate independently. This usually

requires duplicate message detection in the listener implementation, to cover the case where database processing has committed but message processing failed to commit.

Consider the following bean definition:

```
<bean id="jmsContainer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="sessionTransacted" value="true"/>
</bean>
```

To participate in an externally managed transaction, you need to configure a transaction manager and use a listener container that supports externally managed transactions (typically, `DefaultMessageListenerContainer`).

To configure a message listener container for XA transaction participation, you want to configure a `JtaTransactionManager` (which, by default, delegates to the Jakarta EE server's transaction subsystem). Note that the underlying JMS `ConnectionFactory` needs to be XA-capable and properly registered with your JTA transaction coordinator. (Check your Jakarta EE server's configuration of JNDI resources.) This lets message reception as well as (for example) database access be part of the same transaction (with unified commit semantics, at the expense of XA transaction log overhead).

The following bean definition creates a transaction manager:

```
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Then we need to add it to our earlier container configuration. The container takes care of the rest. The following example shows how to do so:

```
<bean id="jmsContainer"
class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="destination"/>
    <property name="messageListener" ref="messageListener"/>
    <property name="transactionManager" ref="transactionManager"/> ①
</bean>
```

① Our transaction manager.

7.2.4. Support for JCA Message Endpoints

Beginning with version 2.5, Spring also provides support for a JCA-based `MessageListener` container. The `JmsMessageEndpointManager` tries to automatically determine the `ActivationSpec` class name from the provider's `ResourceAdapter` class name. Therefore, it is typically possible to provide Spring's

generic `JmsActivationSpecConfig`, as the following example shows:

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpecConfig">
        <bean
class="org.springframework.jms.listener.endpoint.JmsActivationSpecConfig">
            <property name="destinationName" value="myQueue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener"/>
</bean>
```

Alternatively, you can set up a `JmsMessageEndpointManager` with a given `ActivationSpec` object. The `ActivationSpec` object may also come from a JNDI lookup (using `<jee:jndi-lookup>`). The following example shows how to do so:

```
<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">
    <property name="resourceAdapter" ref="resourceAdapter"/>
    <property name="activationSpec">
        <bean class="org.apache.activemq.ra.ActiveMQActivationSpec">
            <property name="destination" value="myQueue"/>
            <property name="destinationType" value="jakarta.jms.Queue"/>
        </bean>
    </property>
    <property name="messageListener" ref="myMessageListener"/>
</bean>
```

Using Spring's `ResourceAdapterFactoryBean`, you can configure the target `ResourceAdapter` locally, as the following example shows:

```
<bean id="resourceAdapter"
class="org.springframework.jca.support.ResourceAdapterFactoryBean">
    <property name="resourceAdapter">
        <bean class="org.apache.activemq.ra.ActiveMQResourceAdapter">
            <property name="serverUrl" value="tcp://localhost:61616"/>
        </bean>
    </property>
    <property name="workManager">
        <bean class="org.springframework.jca.work.SimpleTaskWorkManager"/>
    </property>
</bean>
```

The specified `WorkManager` can also point to an environment-specific thread pool — typically through a `SimpleTaskWorkManager` instance's `asyncTaskExecutor` property. Consider defining a shared thread pool for all your `ResourceAdapter` instances if you happen to use multiple adapters.

In some environments (such as WebLogic 9 or above), you can instead obtain the entire `ResourceAdapter` object from JNDI (by using `<jee:jndi-lookup>`). The Spring-based message listeners can then interact with the server-hosted `ResourceAdapter`, which also use the server's built-in `WorkManager`.

See the javadoc for `JmsMessageEndpointManager`, `JmsActivationSpecConfig`, and `ResourceAdapterFactoryBean` for more details.

Spring also provides a generic JCA message endpoint manager that is not tied to JMS: `org.springframework.jca.endpoint.GenericMessageEndpointManager`. This component allows for using any message listener type (such as a JMS `MessageListener`) and any provider-specific `ActivationSpec` object. See your JCA provider's documentation to find out about the actual capabilities of your connector, and see the `GenericMessageEndpointManager` javadoc for the Spring-specific configuration details.

JCA-based message endpoint management is very analogous to EJB 2.1 Message-Driven Beans. It uses the same underlying resource provider contract. As with EJB 2.1 MDBs, you can use any message listener interface supported by your JCA provider in the Spring context as well. Spring nevertheless provides explicit "convenience" support for JMS, because JMS is the most common endpoint API used with the JCA endpoint management contract.

7.2.5. Annotation-driven Listener Endpoints

The easiest way to receive a message asynchronously is to use the annotated listener endpoint infrastructure. In a nutshell, it lets you expose a method of a managed bean as a JMS listener endpoint. The following example shows how to use it:

```
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(String data) { ... }
}
```

The idea of the preceding example is that, whenever a message is available on the `jakarta.jms.Destination myDestination`, the `processOrder` method is invoked accordingly (in this case, with the content of the JMS message, similar to what the `MessageListenerAdapter` provides).

The annotated endpoint infrastructure creates a message listener container behind the scenes for each annotated method, by using a `JmsListenerContainerFactory`. Such a container is not registered against the application context but can be easily located for management purposes by using the `JmsListenerEndpointRegistry` bean.

`@JmsListener` is a repeatable annotation on Java 8, so you can associate several JMS destinations with the same method by adding additional `@JmsListener` declarations to it.

Enable Listener Endpoint Annotations

To enable support for `@JmsListener` annotations, you can add `@EnableJms` to one of your `@Configuration` classes, as the following example shows:

```
@Configuration
@EnableJms
public class AppConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory());
        factory.setDestinationResolver(destinationResolver());
        factory.setSessionTransacted(true);
        factory.setConcurrency("3-10");
        return factory;
    }
}
```

By default, the infrastructure looks for a bean named `jmsListenerContainerFactory` as the source for the factory to use to create message listener containers. In this case (and ignoring the JMS infrastructure setup), you can invoke the `processOrder` method with a core poll size of three threads and a maximum pool size of ten threads.

You can customize the listener container factory to use for each annotation or you can configure an explicit default by implementing the `JmsListenerConfigurer` interface. The default is required only if at least one endpoint is registered without a specific container factory. See the javadoc of classes that implement `JmsListenerConfigurer` for details and examples.

If you prefer XML configuration, you can use the `<jms:annotation-driven>` element, as the following example shows:

```
<jms:annotation-driven/>

<bean id="jmsListenerContainerFactory"
        class="org.springframework.jms.config.DefaultJmsListenerContainerFactory">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destinationResolver" ref="destinationResolver"/>
    <property name="sessionTransacted" value="true"/>
    <property name="concurrency" value="3-10"/>
</bean>
```

Programmatic Endpoint Registration

`JmsListenerEndpoint` provides a model of a JMS endpoint and is responsible for configuring the container for that model. The infrastructure lets you programmatically configure endpoints in

addition to the ones that are detected by the `JmsListener` annotation. The following example shows how to do so:

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {
        SimpleJmsListenerEndpoint endpoint = new SimpleJmsListenerEndpoint();
        endpoint.setId("myJmsEndpoint");
        endpoint.setDestination("anotherQueue");
        endpoint.setMessageListener(message -> {
            // processing
        });
        registrar.registerEndpoint(endpoint);
    }
}
```

In the preceding example, we used `SimpleJmsListenerEndpoint`, which provides the actual `MessageListener` to invoke. However, you could also build your own endpoint variant to describe a custom invocation mechanism.

Note that you could skip the use of `@JmsListener` altogether and programmatically register only your endpoints through `JmsListenerConfigurer`.

Annotated Endpoint Method Signature

So far, we have been injecting a simple `String` in our endpoint, but it can actually have a very flexible method signature. In the following example, we rewrite it to inject the `Order` with a custom header:

```
@Component
public class MyService {

    @JmsListener(destination = "myDestination")
    public void processOrder(Order order, @Header("order_type") String orderType) {
        ...
    }
}
```

The main elements you can inject in JMS listener endpoints are as follows:

- The raw `jakarta.jms.Message` or any of its subclasses (provided that it matches the incoming message type).
- The `jakarta.jms.Session` for optional access to the native JMS API (for example, for sending a custom reply).

- The `org.springframework.messaging.Message` that represents the incoming JMS message. Note that this message holds both the custom and the standard headers (as defined by `JmsHeaders`).
- `@Header`-annotated method arguments to extract a specific header value, including standard JMS headers.
- A `@Headers`-annotated argument that must also be assignable to `java.util.Map` for getting access to all headers.
- A non-annotated element that is not one of the supported types (`Message` or `Session`) is considered to be the payload. You can make that explicit by annotating the parameter with `@Payload`. You can also turn on validation by adding an extra `@Valid`.

The ability to inject Spring's `Message` abstraction is particularly useful to benefit from all the information stored in the transport-specific message without relying on transport-specific API. The following example shows how to do so:

```
@JmsListener(destination = "myDestination")
public void processOrder(Message<Order> order) { ... }
```

Handling of method arguments is provided by `DefaultMessageHandlerMethodFactory`, which you can further customize to support additional method arguments. You can customize the conversion and validation support there as well.

For instance, if we want to make sure our `Order` is valid before processing it, we can annotate the payload with `@Valid` and configure the necessary validator, as the following example shows:

```
@Configuration
@EnableJms
public class AppConfig implements JmsListenerConfigurer {

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {
        registrar.setMessageHandlerMethodFactory(myJmsHandlerMethodFactory());
    }

    @Bean
    public DefaultMessageHandlerMethodFactory myHandlerMethodFactory() {
        DefaultMessageHandlerMethodFactory factory = new
DefaultMessageHandlerMethodFactory();
        factory.setValidator(myValidator());
        return factory;
    }
}
```

Response Management

The existing support in `MessageListenerAdapter` already lets your method have a non-`void` return type. When that is the case, the result of the invocation is encapsulated in a `jakarta.jms.Message`,

sent either in the destination specified in the `JMSReplyTo` header of the original message or in the default destination configured on the listener. You can now set that default destination by using the `@SendTo` annotation of the messaging abstraction.

Assuming that our `processOrder` method should now return an `OrderStatus`, we can write it to automatically send a response, as the following example shows:

```
@JmsListener(destination = "myDestination")
@SendTo("status")
public OrderStatus processOrder(Order order) {
    // order processing
    return status;
}
```

If you have several `@JmsListener`-annotated methods, you can also place the `@SendTo` annotation at the class level to share a default reply destination.

If you need to set additional headers in a transport-independent manner, you can return a `Message` instead, with a method similar to the following:

```
@JmsListener(destination = "myDestination")
@SendTo("status")
public Message<OrderStatus> processOrder(Order order) {
    // order processing
    return MessageBuilder
             .withPayload(status)
             .setHeader("code", 1234)
             .build();
}
```

If you need to compute the response destination at runtime, you can encapsulate your response in a `JmsResponse` instance that also provides the destination to use at runtime. We can rewrite the previous example as follows:

```
@JmsListener(destination = "myDestination")
public JmsResponse<Message<OrderStatus>> processOrder(Order order) {
    // order processing
    Message<OrderStatus> response = MessageBuilder
             .withPayload(status)
             .setHeader("code", 1234)
             .build();
    return JmsResponse.forQueue(response, "status");
}
```

Finally, if you need to specify some QoS values for the response such as the priority or the time to live, you can configure the `JmsListenerContainerFactory` accordingly, as the following example

shows:

```
@Configuration
@EnableJms
public class AppConfig {

    @Bean
    public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory());
        QosSettings replyQosSettings = new QosSettings();
        replyQosSettings.setPriority(2);
        replyQosSettings.setTimeToLive(10000);
        factory.setReplyQosSettings(replyQosSettings);
        return factory;
    }
}
```

7.2.6. JMS Namespace Support

Spring provides an XML namespace for simplifying JMS configuration. To use the JMS namespace elements, you need to reference the JMS schema, as the following example shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jms="http://www.springframework.org/schema/jms" ①
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/jms
https://www.springframework.org/schema/jms/spring-jms.xsd">

    <!-- bean definitions here -->

</beans>
```

① Referencing the JMS schema.

The namespace consists of three top-level elements: `<annotation-driven/>`, `<listener-container/>` and `<jca-listener-container/>`. `<annotation-driven/>` enables the use of annotation-driven listener endpoints. `<listener-container/>` and `<jca-listener-container/>` define shared listener container configuration and can contain `<listener/>` child elements. The following example shows a basic configuration for two listeners:

```
<jms:listener-container>

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger"
method="log"/>

</jms:listener-container>
```

The preceding example is equivalent to creating two distinct listener container bean definitions and two distinct `MessageListenerAdapter` bean definitions, as shown in Using `MessageListenerAdapter`. In addition to the attributes shown in the preceding example, the `listener` element can contain several optional ones. The following table describes all of the available attributes:

Table 29. Attributes of the JMS <listener> element

Attribute	Description
`id`	A bean name for the hosting listener container. If not specified, a bean name is automatically generated.
`destination` (required)	The destination name for this listener, resolved through the `DestinationResolver` strategy.
`ref` (required)	The bean name of the handler object.
`method`	The name of the handler method to invoke. If the `ref` attribute points to a `MessageListener` or Spring `SessionAwareMessageListener`, you can omit this attribute.
`response-destination`	The name of the default response destination to which to send response messages. This is applied in case of a request message that does not carry a `JMSReplyTo` field. The type of this destination is determined by the listener-container's `response-destination-type` attribute. Note that this applies only to a listener method with a return value, for which each result object is converted into a response message.
`subscription`	The name of the durable subscription, if any.
`selector`	An optional message selector for this listener.
`concurrency`	The number of concurrent sessions or consumers to start for this listener. This value can either be a simple number indicating the maximum number (for example, `5`) or a range indicating the lower as well as the upper limit (for example, `3-5`). Note that a specified minimum is only a hint and might be ignored at runtime. The default is the value provided by the container.

The `<listener-container/>` element also accepts several optional attributes. This allows for customization of the various strategies (for example, `taskExecutor` and `destinationResolver`) as well as basic JMS settings and resource references. By using these attributes, you can define highly-customized listener containers while still benefiting from the convenience of the namespace.

You can automatically expose such settings as a `JmsListenerContainerFactory` by specifying the `id` of

the bean to expose through the `factory-id` attribute, as the following example shows:

```
<jms:listener-container connection-factory="myConnectionFactory"
        task-executor="myTaskExecutor"
        destination-resolver="myDestinationResolver"
        transaction-manager="myTransactionManager"
        concurrency="10">

    <jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

    <jms:listener destination="queue.confirmations" ref="confirmationLogger"
method="log"/>

</jms:listener-container>
```

The following table describes all available attributes. See the class-level javadoc of the `AbstractMessageListenerContainer` and its concrete subclasses for more details on the individual properties. The javadoc also provides a discussion of transaction choices and message redelivery scenarios.

Table 30. Attributes of the JMS <listener-container> element

Attribute	Description
`container-type`	The type of this listener container. The available options are `default`, `simple`, `default102`, or `simple102` (the default option is `default`).
`container-class`	A custom listener container implementation class as a fully qualified class name. The default is Spring's standard `DefaultMessageListenerContainer` or `SimpleMessageListenerContainer`, according to the `container-type` attribute.
`factory-id`	Exposes the settings defined by this element as a `JmsListenerContainerFactory` with the specified `id` so that they can be reused with other endpoints.
`connection-factory`	A reference to the JMS `ConnectionFactory` bean (the default bean name is `connectionFactory`).
`task-executor`	A reference to the Spring `TaskExecutor` for the JMS listener invokers.
`destination-resolver`	A reference to the `DestinationResolver` strategy for resolving JMS `Destination` instances.
`message-converter`	A reference to the `MessageConverter` strategy for converting JMS Messages to listener method arguments. The default is a `SimpleMessageConverter`.
`error-handler`	A reference to an `ErrorHandler` strategy for handling any uncaught exceptions that may occur during the execution of the `MessageListener`.
`destination-type`	The JMS destination type for this listener: `queue`, `topic`, `durableTopic`, `sharedTopic`, or `sharedDurableTopic`. This potentially enables the `pubSubDomain`, `subscriptionDurable` and `subscriptionShared` properties of the container. The default is `queue` (which disables those three properties).

Attribute	Description
`response-destination-type`	The JMS destination type for responses: `queue` or `topic`. The default is the value of the `destination-type` attribute.
`client-id`	The JMS client ID for this listener container. You must specify it when you use durable subscriptions.
`cache`	The cache level for JMS resources: `none`, `connection`, `session`, `consumer`, or `auto`. By default (`auto`), the cache level is effectively `consumer`, unless an external transaction manager has been specified — in which case, the effective default will be `none` (assuming Jakarta EE-style transaction management, where the given ConnectionFactory is an XA-aware pool).
`acknowledge`	The native JMS acknowledge mode: `auto`, `client`, `dups-ok`, or `transacted`. A value of `transacted` activates a locally transacted `Session`. As an alternative, you can specify the `transaction-manager` attribute, described later in table. The default is `auto`.
`transaction-manager`	A reference to an external `PlatformTransactionManager` (typically an XA-based transaction coordinator, such as Spring's `JtaTransactionManager`). If not specified, native acknowledging is used (see the `acknowledge` attribute).
`concurrency`	The number of concurrent sessions or consumers to start for each listener. It can either be a simple number indicating the maximum number (for example, `5`) or a range indicating the lower as well as the upper limit (for example, `3-5`). Note that a specified minimum is just a hint and might be ignored at runtime. The default is `1`. You should keep concurrency limited to `1` in case of a topic listener or if queue ordering is important. Consider raising it for general queues.
`prefetch`	The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers.
`receive-timeout`	The timeout (in milliseconds) to use for receive calls. The default is `1000` (one second). `-1` indicates no timeout.
`back-off`	Specifies the `BackOff` instance to use to compute the interval between recovery attempts. If the `BackOffExecution` implementation returns `BackOffExecution#STOP`, the listener container does not further try to recover. The `recovery-interval` value is ignored when this property is set. The default is a `FixedBackOff` with an interval of 5000 milliseconds (that is, five seconds).
`recovery-interval`	Specifies the interval between recovery attempts, in milliseconds. It offers a convenient way to create a `FixedBackOff` with the specified interval. For more recovery options, consider specifying a `BackOff` instance instead. The default is 5000 milliseconds (that is, five seconds).
`phase`	The lifecycle phase within which this container should start and stop. The lower the value, the earlier this container starts and the later it stops. The default is `Integer.MAX_VALUE`, meaning that the container starts as late as possible and stops as soon as possible.

Configuring a JCA-based listener container with the `jms` schema support is very similar, as the following example shows:

```
<jms:jca-listener-container resource-adapter="myResourceAdapter"
        destination-resolver="myDestinationResolver"
        transaction-manager="myTransactionManager"
        concurrency="10">

    <jms:listener destination="queue.orders" ref="myMessageListener"/>

</jms:jca-listener-container>
```

The following table describes the available configuration options for the JCA variant:

Table 31. Attributes of the JMS <jca-listener-container/> element

Attribute	Description
`factory-id`	Exposes the settings defined by this element as a `JmsListenerContainerFactory` with the specified `id` so that they can be reused with other endpoints.
`resource-adapter`	A reference to the JCA `ResourceAdapter` bean (the default bean name is `resourceAdapter`).
`activation-spec-factory`	A reference to the `JmsActivationSpecFactory`. The default is to autodetect the JMS provider and its `ActivationSpec` class (see `DefaultJmsActivationSpecFactory`).
`destination-resolver`	A reference to the `DestinationResolver` strategy for resolving JMS `Destinations`.
`message-converter`	A reference to the `MessageConverter` strategy for converting JMS Messages to listener method arguments. The default is `SimpleMessageConverter`.
`destination-type`	The JMS destination type for this listener: `queue`, `topic`, `durableTopic`, `sharedTopic`. or `sharedDurableTopic`. This potentially enables the `pubSubDomain`, `subscriptionDurable`, and `subscriptionShared` properties of the container. The default is `queue` (which disables those three properties).
`response-destination-type`	The JMS destination type for responses: `queue` or `topic`. The default is the value of the `destination-type` attribute.
`client-id`	The JMS client ID for this listener container. It needs to be specified when using durable subscriptions.
`acknowledge`	The native JMS acknowledge mode: `auto`, `client`, `dups-ok`, or `transacted`. A value of `transacted` activates a locally transacted `Session`. As an alternative, you can specify the `transaction-manager` attribute described later. The default is `auto`.
`transaction-manager`	A reference to a Spring `JtaTransactionManager` or a `jakarta.transaction.TransactionManager` for kicking off an XA transaction for each incoming message. If not specified, native acknowledging is used (see the `acknowledge` attribute).

Attribute	Description
`concurrency`	The number of concurrent sessions or consumers to start for each listener. It can either be a simple number indicating the maximum number (for example `5`) or a range indicating the lower as well as the upper limit (for example, `3-5`). Note that a specified minimum is only a hint and is typically ignored at runtime when you use a JCA listener container. The default is 1.
`prefetch`	The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers.

7.3. JMX

The JMX (Java Management Extensions) support in Spring provides features that let you easily and transparently integrate your Spring application into a JMX infrastructure.

JMX?

This chapter is not an introduction to JMX. It does not try to explain why you might want to use JMX. If you are new to JMX, see Further Resources at the end of this chapter.

Specifically, Spring's JMX support provides four core features:

- The automatic registration of any Spring bean as a JMX MBean.
- A flexible mechanism for controlling the management interface of your beans.
- The declarative exposure of MBeans over remote, JSR-160 connectors.
- The simple proxying of both local and remote MBean resources.

These features are designed to work without coupling your application components to either Spring or JMX interfaces and classes. Indeed, for the most part, your application classes need not be aware of either Spring or JMX in order to take advantage of the Spring JMX features.

7.3.1. Exporting Your Beans to JMX

The core class in Spring's JMX framework is the `MBeanExporter`. This class is responsible for taking your Spring beans and registering them with a JMX `MBeanServer`. For example, consider the following class:

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;
    private int age;
    private boolean isSuperman;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }
}
```

To expose the properties and methods of this bean as attributes and operations of an MBean, you can configure an instance of the `MBeanExporter` class in your configuration file and pass in the bean, as the following example shows:

```
<beans>
    <!-- this bean must not be lazily initialized if the exporting is to happen -->
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-
init="false">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
    </bean>
    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

The pertinent bean definition from the preceding configuration snippet is the `exporter` bean. The `beans` property tells the `MBeanExporter` exactly which of your beans must be exported to the JMX `MBeanServer`. In the default configuration, the key of each entry in the `beans Map` is used as the `ObjectName` for the bean referenced by the corresponding entry value. You can change this behavior, as described in Controlling `ObjectName` Instances for Your Beans.

With this configuration, the `testBean` bean is exposed as an MBean under the `ObjectName` `bean:name=testBean1`. By default, all `public` properties of the bean are exposed as attributes and all `public` methods (except those inherited from the `Object` class) are exposed as operations.

`MBeanExporter` is a `Lifecycle` bean (see Startup and Shutdown Callbacks). By default, MBeans are exported as late as possible during the application lifecycle. You can configure the `phase` at which the export happens or disable automatic registration by setting the `autoStartup` flag.

Creating an MBeanServer

The configuration shown in the preceding section assumes that the application is running in an environment that has one (and only one) `MBeanServer` already running. In this case, Spring tries to locate the running `MBeanServer` and register your beans with that server (if any). This behavior is useful when your application runs inside a container (such as Tomcat or IBM WebSphere) that has its own `MBeanServer`.

However, this approach is of no use in a standalone environment or when running inside a container that does not provide an `MBeanServer`. To address this, you can create an `MBeanServer` instance declaratively by adding an instance of the `org.springframework.jmx.support.MBeanServerFactoryBean` class to your configuration. You can also ensure that a specific `MBeanServer` is used by setting the value of the `MBeanExporter` instance's `server` property to the `MBeanServer` value returned by an `MBeanServerFactoryBean`, as the following example shows:

```
<beans>

    <bean id="mbeanServer"
class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

    <!--
    this bean needs to be eagerly pre-instantiated in order for the exporting to
occur;
    this means that it must not be marked as lazily initialized
    -->
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="server" ref="mbeanServer"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

In the preceding example, an instance of `MBeanServer` is created by the `MBeanServerFactoryBean` and is supplied to the `MBeanExporter` through the `server` property. When you supply your own `MBeanServer` instance, the `MBeanExporter` does not try to locate a running `MBeanServer` and uses the supplied `MBeanServer` instance. For this to work correctly, you must have a JMX implementation on your classpath.

Reusing an Existing `MBeanServer`

If no server is specified, the `MBeanExporter` tries to automatically detect a running `MBeanServer`. This works in most environments, where only one `MBeanServer` instance is used. However, when multiple instances exist, the exporter might pick the wrong server. In such cases, you should use the `MBeanServer` `agentId` to indicate which instance to be used, as the following example shows:

```
<beans>
    <bean id="mbeanServer"
class="org.springframework.jmx.support.MBeanServerFactoryBean">
        <!-- indicate to first look for a server -->
        <property name="locateExistingServerIfPossible" value="true"/>
        <!-- search for the MBeanServer instance with the given agentId -->
        <property name="agentId" value="MBeanServer_instance_agentId>"/>
    </bean>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="server" ref="mbeanServer"/>
        ...
    </bean>
</beans>
```

For platforms or cases where the existing `MBeanServer` has a dynamic (or unknown) `agentId` that is retrieved through lookup methods, you should use factory-method, as the following example shows:

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="server">
            <!-- Custom MBeanServerLocator -->
            <bean class="platform.package.MBeanServerLocator" factory-
method="locateMBeanServer"/>
        </property>
    </bean>

    <!-- other beans here -->

</beans>
```

Lazily Initialized MBeans

If you configure a bean with an `MBeanExporter` that is also configured for lazy initialization, the `MBeanExporter` does not break this contract and avoids instantiating the bean. Instead, it registers a proxy with the `MBeanServer` and defers obtaining the bean from the container until the first invocation on the proxy occurs.

Automatic Registration of MBeans

Any beans that are exported through the `MBeanExporter` and are already valid MBeans are registered as-is with the `MBeanServer` without further intervention from Spring. You can cause MBeans to be automatically detected by the `MBeanExporter` by setting the `autodetect` property to `true`, as the following example shows:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true"
class="org.springframework.jmx.export.TestDynamicMBean"/>
```

In the preceding example, the bean called `spring:mbean=true` is already a valid JMX MBean and is automatically registered by Spring. By default, a bean that is autodetected for JMX registration has its bean name used as the `ObjectName`. You can override this behavior, as detailed in Controlling `ObjectName` Instances for Your Beans.

Controlling the Registration Behavior

Consider the scenario where a Spring `MBeanExporter` attempts to register an `MBean` with an `MBeanServer` by using the `ObjectName bean:name=testBean1`. If an `MBean` instance has already been registered under that same `ObjectName`, the default behavior is to fail (and throw an `InstanceAlreadyExistsException`).

You can control exactly what happens when an `MBean` is registered with an `MBeanServer`. Spring's JMX support allows for three different registration behaviors to control the registration behavior when the registration process finds that an `MBean` has already been registered under the same `ObjectName`. The following table summarizes these registration behaviors:

Table 32. Registration Behaviors

Registration behavior	Explanation
`FAIL_ON_EXISTING`	This is the default registration behavior. If an `MBean` instance has already been registered under the same `ObjectName`, the `MBean` that is being registered is not registered, and an `InstanceAlreadyExistsException` is thrown. The existing `MBean` is unaffected.
`IGNORE_EXISTING`	If an `MBean` instance has already been registered under the same `ObjectName`, the `MBean` that is being registered is not registered. The existing `MBean` is unaffected, and no `Exception` is thrown. This is useful in settings where multiple applications want to share a common `MBean` in a shared `MBeanServer`.
`REPLACE_EXISTING`	If an `MBean` instance has already been registered under the same `ObjectName`, the existing `MBean` that was previously registered is unregistered, and the new `MBean` is registered in its place (the new `MBean` effectively replaces the previous instance).

The values in the preceding table are defined as enums on the `RegistrationPolicy` class. If you want to change the default registration behavior, you need to set the value of the `registrationPolicy` property on your `MBeanExporter` definition to one of those values.

The following example shows how to change from the default registration behavior to the `REPLACE_EXISTING` behavior:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="registrationPolicy" value="REPLACE_EXISTING"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

7.3.2. Controlling the Management Interface of Your Beans

In the example in the preceding section, you had little control over the management interface of your bean. All of the `public` properties and methods of each exported bean were exposed as JMX attributes and operations, respectively. To exercise finer-grained control over exactly which properties and methods of your exported beans are actually exposed as JMX attributes and operations, Spring JMX provides a comprehensive and extensible mechanism for controlling the management interfaces of your beans.

Using the `MBeanInfoAssembler` Interface

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `org.springframework.jmx.export.assembler.MBeanInfoAssembler` interface, which is responsible for defining the management interface of each bean that is exposed. The default implementation, `org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler`, defines a management interface that exposes all public properties and methods (as you saw in the examples in the preceding sections). Spring provides two additional implementations of the `MBeanInfoAssembler` interface that let you control the generated management interface by using either source-level metadata or any arbitrary interface.

Using Source-level Metadata: Java Annotations

By using the `MetadataMBeanInfoAssembler`, you can define the management interfaces for your beans by using source-level metadata. The reading of metadata is encapsulated by the `org.springframework.jmx.export.metadata.JmxAttributeSource` interface. Spring JMX provides a default implementation that uses Java annotations, namely `org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource`. You must configure the `MetadataMBeanInfoAssembler` with an implementation instance of the `JmxAttributeSource` interface for it to function correctly (there is no default).

To mark a bean for export to JMX, you should annotate the bean class with the `ManagedResource` annotation. You must mark each method you wish to expose as an operation with the `ManagedOperation` annotation and mark each property you wish to expose with the `ManagedAttribute` annotation. When marking properties, you can omit either the annotation of the getter or the setter to create a write-only or read-only attribute, respectively.

A `ManagedResource`-annotated bean must be public, as must the methods exposing an operation or an attribute.

The following example shows the annotated version of the `JmxTestBean` class that we used in Creating an MBeanServer:

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(
        objectName="bean:name=testBean4",
        description="My Managed Bean",
        log=true,
        logFile="jmx.log",
        currencyTimeLimit=15,
        persistPolicy="OnUpdate",
        persistPeriod=200,
        persistLocation="foo",
        persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

    private String name;
    private int age;

    @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @ManagedAttribute(description="The Name Attribute",
            currencyTimeLimit=20,
            defaultValue="bar",
            persistPolicy="OnUpdate")
    public void setName(String name) {
        this.name = name;
    }
```

```
    @ManagedAttribute(defaultValue="foo", persistPeriod=300)
    public String getName() {
        return name;
    }

    @ManagedOperation(description="Add two numbers")
    @ManagedOperationParameters({
        @ManagedOperationParameter(name = "x", description = "The first number"),
        @ManagedOperationParameter(name = "y", description = "The second number")})
    public int add(int x, int y) {
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

}
```

In the preceding example, you can see that the `JmxTestBean` class is marked with the `ManagedResource` annotation and that this `ManagedResource` annotation is configured with a set of properties. These properties can be used to configure various aspects of the MBean that is generated by the `MBeanExporter` and are explained in greater detail later in Source-level Metadata Types.

Both the `age` and `name` properties are annotated with the `ManagedAttribute` annotation, but, in the case of the `age` property, only the getter is marked. This causes both of these properties to be included in the management interface as attributes, but the `age` attribute is read-only.

Finally, the `add(int, int)` method is marked with the `ManagedOperation` attribute, whereas the `dontExposeMe()` method is not. This causes the management interface to contain only one operation (`add(int, int)`) when you use the `MetadataMBeanInfoAssembler`.

The following configuration shows how you can configure the `MBeanExporter` to use the `MetadataMBeanInfoAssembler`:

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"

class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metadata -->
    <bean id="assembler"

class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <!-- will pick up the ObjectName from the annotation -->
    <bean id="namingStrategy"
            class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>
</beans>
```

In the preceding example, an `MetadataMBeanInfoAssembler` bean has been configured with an instance of the `AnnotationJmxAttributeSource` class and passed to the `MBeanExporter` through the assembler property. This is all that is required to take advantage of metadata-driven management interfaces for your Spring-exposed MBeans.

Source-level Metadata Types

The following table describes the source-level metadata types that are available for use in Spring JMX:

Table 33. Source-level metadata types

Purpose	Annotation	Annotation Type
Mark all instances of a `Class` as JMX managed resources.	`@ManagedResource`	Class
Mark a method as a JMX operation.	`@ManagedOperation`	Method

Purpose	Annotation	Annotation Type
Mark a getter or setter as one half of a JMX attribute.	`@ManagedAttribute`	Method (only getters and setters)
Define descriptions for operation parameters.	`@ManagedOperationParameter` and `@ManagedOperationParameters`	Method

The following table describes the configuration parameters that are available for use on these source-level metadata types:

Table 34. Source-level metadata parameters

Parameter	Description	Applies to
`ObjectName`	Used by `MetadataNamingStrategy` to determine the `ObjectName` of a managed resource.	`ManagedResource`
`description`	Sets the friendly description of the resource, attribute or operation.	`ManagedResource`, `ManagedAttribute`, `ManagedOperation`, or `ManagedOperationParameter`
`currencyTimeLimit`	Sets the value of the `currencyTimeLimit` descriptor field.	`ManagedResource` or `ManagedAttribute`
`defaultValue`	Sets the value of the `defaultValue` descriptor field.	`ManagedAttribute`
`log`	Sets the value of the `log` descriptor field.	`ManagedResource`
`logFile`	Sets the value of the `logFile` descriptor field.	`ManagedResource`
`persistPolicy`	Sets the value of the `persistPolicy` descriptor field.	`ManagedResource`
`persistPeriod`	Sets the value of the `persistPeriod` descriptor field.	`ManagedResource`
`persistLocation`	Sets the value of the `persistLocation` descriptor field.	`ManagedResource`
`persistName`	Sets the value of the `persistName` descriptor field.	`ManagedResource`
`name`	Sets the display name of an operation parameter.	`ManagedOperationParameter`
`index`	Sets the index of an operation parameter.	`ManagedOperationParameter`

Using the `AutodetectCapableMBeanInfoAssembler` Interface

To simplify configuration even further, Spring includes the `AutodetectCapableMBeanInfoAssembler` interface, which extends the `MBeanInfoAssembler` interface to add support for autodetection of MBean resources. If you configure the `MBeanExporter` with an instance of `AutodetectCapableMBeanInfoAssembler`, it is allowed to "vote" on the inclusion of beans for exposure to JMX.

The only implementation of the `AutodetectCapableMBeanInfo` interface is the `MetadataMBeanInfoAssembler`, which votes to include any bean that is marked with the

`ManagedResource` attribute. The default approach in this case is to use the bean name as the `ObjectName`, which results in a configuration similar to the following:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <!-- notice how no 'beans' are explicitly configured here -->
        <property name="autodetect" value="true"/>
        <property name="assembler" ref="assembler"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="assembler"
class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource">
            <bean
class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
        </property>
    </bean>

</beans>
```

Notice that, in the preceding configuration, no beans are passed to the `MBeanExporter`. However, the `JmxTestBean` is still registered, since it is marked with the `ManagedResource` attribute and the `MetadataMBeanInfoAssembler` detects this and votes to include it. The only problem with this approach is that the name of the `JmxTestBean` now has business meaning. You can address this issue by changing the default behavior for `ObjectName` creation as defined in Controlling `ObjectName` Instances for Your Beans.

Defining Management Interfaces by Using Java Interfaces

In addition to the `MetadataMBeanInfoAssembler`, Spring also includes the `InterfaceBasedMBeanInfoAssembler`, which lets you constrain the methods and properties that are exposed based on the set of methods defined in a collection of interfaces.

Although the standard mechanism for exposing MBeans is to use interfaces and a simple naming scheme, `InterfaceBasedMBeanInfoAssembler` extends this functionality by removing the need for naming conventions, letting you use more than one interface and removing the need for your beans to implement the MBean interfaces.

Consider the following interface, which is used to define a management interface for the `JmxTestBean` class that we showed earlier:

```
public interface IJmxTestBean {

    public int add(int x, int y);

    public long myOperation();

    public int getAge();

    public void setAge(int age);

    public void setName(String name);

    public String getName();

}
```

This interface defines the methods and properties that are exposed as operations and attributes on the JMX MBean. The following code shows how to configure Spring JMX to use this interface as the definition for the management interface:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean5" value-ref="testBean"/>
            </map>
        </property>
        <property name="assembler">
            <bean
class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
                <property name="managedInterfaces">
                    <value>org.springframework.jmx.IJmxTestBean</value>
                </property>
            </bean>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

In the preceding example, the `InterfaceBasedMBeanInfoAssembler` is configured to use the `IJmxTestBean` interface when constructing the management interface for any bean. It is important to understand that beans processed by the `InterfaceBasedMBeanInfoAssembler` are not required to

implement the interface used to generate the JMX management interface.

In the preceding case, the `IJmxTestBean` interface is used to construct all management interfaces for all beans. In many cases, this is not the desired behavior, and you may want to use different interfaces for different beans. In this case, you can pass `InterfaceBasedMBeanInfoAssembler` a `Properties` instance through the `interfaceMappings` property, where the key of each entry is the bean name and the value of each entry is a comma-separated list of interface names to use for that bean.

If no management interface is specified through either the `managedInterfaces` or `interfaceMappings` properties, the `InterfaceBasedMBeanInfoAssembler` reflects on the bean and uses all of the interfaces implemented by that bean to create the management interface.

Using `MethodNameBasedMBeanInfoAssembler`

`MethodNameBasedMBeanInfoAssembler` lets you specify a list of method names that are exposed to JMX as attributes and operations. The following code shows a sample configuration:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="bean:name=testBean5" value-ref="testBean"/>
        </map>
    </property>
    <property name="assembler">
        <bean
class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
            <property name="managedMethods">
                <value>add,myOperation,getName,setName,getAge</value>
            </property>
        </bean>
    </property>
</bean>
```

In the preceding example, you can see that the `add` and `myOperation` methods are exposed as JMX operations, and `getName()`, `setName(String)`, and `getAge()` are exposed as the appropriate half of a JMX attribute. In the preceding code, the method mappings apply to beans that are exposed to JMX. To control method exposure on a bean-by-bean basis, you can use the `methodMappings` property of `MethodNameMBeanInfoAssembler` to map bean names to lists of method names.

7.3.3. Controlling `ObjectName` Instances for Your Beans

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `ObjectNamingStrategy` to obtain an `ObjectName` instance for each of the beans it registers. By default, the default implementation, `KeyNamingStrategy` uses the key of the `beans Map` as the `ObjectName`. In addition, the `KeyNamingStrategy` can map the key of the `beans Map` to an entry in a `Properties` file (or files) to resolve the `ObjectName`. In addition to the `KeyNamingStrategy`, Spring provides two additional `ObjectNamingStrategy` implementations: the `IdentityNamingStrategy` (which builds an `ObjectName` based on the JVM identity of the bean) and the `MetadataNamingStrategy` (which uses source-level metadata to obtain the `ObjectName`).

Reading `ObjectName` Instances from Properties

You can configure your own `KeyNamingStrategy` instance and configure it to read `ObjectName` instances from a `Properties` instance rather than use a bean key. The `KeyNamingStrategy` tries to locate an entry in the `Properties` with a key that corresponds to the bean key. If no entry is found or if the `Properties` instance is `null`, the bean key itself is used.

The following code shows a sample configuration for the `KeyNamingStrategy`:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="namingStrategy"
class="org.springframework.jmx.export.naming.KeyNamingStrategy">
        <property name="mappings">
            <props>
                <prop key="testBean">bean:name=testBean1</prop>
            </props>
        </property>
        <property name="mappingLocations">
            <value>names1.properties,names2.properties</value>
        </property>
    </bean>

</beans>
```

The preceding example configures an instance of `KeyNamingStrategy` with a `Properties` instance that is merged from the `Properties` instance defined by the mapping property and the properties files located in the paths defined by the mappings property. In this configuration, the `testBean` bean is given an `ObjectName` of `bean:name=testBean1`, since this is the entry in the `Properties` instance that has a key corresponding to the bean key.

If no entry in the `Properties` instance can be found, the bean key name is used as the `ObjectName`.

Using `MetadataNamingStrategy`

`MetadataNamingStrategy` uses the `objectName` property of the `ManagedResource` attribute on each bean

to create the `ObjectName`. The following code shows the configuration for the `MetadataNamingStrategy`:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="testBean" value-ref="testBean"/>
            </map>
        </property>
        <property name="namingStrategy" ref="namingStrategy"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="namingStrategy"
class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="attributeSource"/>
    </bean>

    <bean id="attributeSource"

class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

</beans>
```

If no `objectName` has been provided for the `ManagedResource` attribute, an `ObjectName` is created with the following format: *[fully-qualified-package-name]:type=[short-classname],name=[bean-name]*. For example, the generated `ObjectName` for the following bean would be `com.example:type=MyClass,name=myBean`:

```
<bean id="myBean" class="com.example.MyClass"/>
```

Configuring Annotation-based MBean Export

If you prefer to use the annotation-based approach to define your management interfaces, a convenience subclass of `MBeanExporter` is available: `AnnotationMBeanExporter`. When defining an instance of this subclass, you no longer need the `namingStrategy`, `assembler`, and `attributeSource` configuration, since it always uses standard Java annotation-based metadata (autodetection is always enabled as well). In fact, rather than defining an `MBeanExporter` bean, an even simpler syntax is supported by the `@EnableMBeanExport` `@Configuration` annotation, as the following example shows:

```
@Configuration
@EnableMBeanExport
public class AppConfig {

}
```

If you prefer XML-based configuration, the `<context:mbean-export/>` element serves the same purpose and is shown in the following listing:

```
<context:mbean-export/>
```

If necessary, you can provide a reference to a particular MBean `server`, and the `defaultDomain` attribute (a property of `AnnotationMBeanExporter`) accepts an alternate value for the generated MBean `ObjectName` domains. This is used in place of the fully qualified package name as described in the previous section on MetadataNamingStrategy, as the following example shows:

```
@EnableMBeanExport(server="myMBeanServer", defaultDomain="myDomain")
@Configuration
ContextConfiguration {

}
```

The following example shows the XML equivalent of the preceding annotation-based example:

```
<context:mbean-export server="myMBeanServer" default-domain="myDomain"/>
```

Do not use interface-based AOP proxies in combination with autodetection of JMX annotations in your bean classes. Interface-based proxies “hide” the target class, which also hides the JMX-managed resource annotations. Hence, you should use target-class proxies in that case (through setting the 'proxy-target-class' flag on `<aop:config/>`, `<tx:annotation-driven/>` and so on). Otherwise, your JMX beans might be silently ignored at startup.

7.3.4. Using JSR-160 Connectors

For remote access, Spring JMX module offers two `FactoryBean` implementations inside the `org.springframework.jmx.support` package for creating both server- and client-side connectors.

Server-side Connectors

To have Spring JMX create, start, and expose a JSR-160 `JMXConnectorServer`, you can use the following configuration:

```
<bean id="serverConnector"
class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

By default, `ConnectorServerFactoryBean` creates a `JMXConnectorServer` bound to `service:jmx:jmxmp://localhost:9875`. The `serverConnector` bean thus exposes the local `MBeanServer` to clients through the JMXMP protocol on localhost, port 9875. Note that the JMXMP protocol is marked as optional by the JSR 160 specification. Currently, the main open-source JMX implementation, MX4J, and the one provided with the JDK do not support JMXMP.

To specify another URL and register the `JMXConnectorServer` itself with the `MBeanServer`, you can use the `serviceUrl` and `ObjectName` properties, respectively, as the following example shows:

```
<bean id="serverConnector"
        class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=rmi"/>
    <property name="serviceUrl"

value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>
</bean>
```

If the `ObjectName` property is set, Spring automatically registers your connector with the `MBeanServer` under that `ObjectName`. The following example shows the full set of parameters that you can pass to the `ConnectorServerFactoryBean` when creating a `JMXConnector`:

```
<bean id="serverConnector"
        class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=iiop"/>
    <property name="serviceUrl"
        value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
    <property name="threaded" value="true"/>
    <property name="daemon" value="true"/>
    <property name="environment">
        <map>
            <entry key="someKey" value="someValue"/>
        </map>
    </property>
</bean>
```

Note that, when you use a RMI-based connector, you need the lookup service (`tnameserv` or `rmiregistry`) to be started in order for the name registration to complete.

Client-side Connectors

To create an `MBeanServerConnection` to a remote JSR-160-enabled `MBeanServer`, you can use the `MBeanServerConnectionFactoryBean`, as the following example shows:

```
<bean id="clientConnector"
class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl"
value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxrmi"/>
</bean>
```

JMX over Hessian or SOAP

JSR-160 permits extensions to the way in which communication is done between the client and the server. The examples shown in the preceding sections use the mandatory RMI-based implementation required by the JSR-160 specification (IIOP and JRMP) and the (optional) JMXMP. By using other providers or JMX implementations (such as MX4J) you can take advantage of protocols such as SOAP or Hessian over simple HTTP or SSL and others, as the following example shows:

```
<bean id="serverConnector"
class="org.springframework.jmx.support.ConnectorServerFactoryBean">
    <property name="objectName" value="connector:name=burlap"/>
    <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

In the preceding example, we used MX4J 3.0.0. See the official MX4J documentation for more information.

7.3.5. Accessing MBeans through Proxies

Spring JMX lets you create proxies that re-route calls to MBeans that are registered in a local or remote `MBeanServer`. These proxies provide you with a standard Java interface, through which you can interact with your MBeans. The following code shows how to configure a proxy for an MBean running in a local `MBeanServer`:

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

In the preceding example, you can see that a proxy is created for the MBean registered under the `ObjectName` of `bean:name=testBean`. The set of interfaces that the proxy implements is controlled by the `proxyInterfaces` property, and the rules for mapping methods and properties on these interfaces to operations and attributes on the MBean are the same rules used by the `InterfaceBasedMBeanInfoAssembler`.

The `MBeanProxyFactoryBean` can create a proxy to any MBean that is accessible through an `MBeanServerConnection`. By default, the local `MBeanServer` is located and used, but you can override this and provide an `MBeanServerConnection` that points to a remote `MBeanServer` to cater for proxies that point to remote MBeans:

```
<bean id="clientConnector"
        class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>
</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
    <property name="server" ref="clientConnector"/>
</bean>
```

In the preceding example, we create an `MBeanServerConnection` that points to a remote machine that uses the `MBeanServerConnectionFactoryBean`. This `MBeanServerConnection` is then passed to the `MBeanProxyFactoryBean` through the `server` property. The proxy that is created forwards all invocations to the `MBeanServer` through this `MBeanServerConnection`.

7.3.6. Notifications

Spring's JMX offering includes comprehensive support for JMX notifications.

Registering Listeners for Notifications

Spring's JMX support makes it easy to register any number of `NotificationListeners` with any number of MBeans (this includes MBeans exported by Spring's `MBeanExporter` and MBeans registered through some other mechanism). For example, consider the scenario where one would like to be informed (through a `Notification`) each and every time an attribute of a target MBean changes. The following example writes notifications to the console:

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
        implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return
AttributeChangeNotification.class.isAssignableFrom(notification.getClass());
    }

}
```

The following example adds `ConsoleLoggingNotificationListener` (defined in the preceding example) to `notificationListenerMappings`:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="bean:name=testBean1">
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

With the preceding configuration in place, every time a JMX `Notification` is broadcast from the target MBean (`bean:name=testBean1`), the `ConsoleLoggingNotificationListener` bean that was registered as a listener through the `notificationListenerMappings` property is notified. The `ConsoleLoggingNotificationListener` bean can then take whatever action it deems appropriate in response to the `Notification`.

You can also use straight bean names as the link between exported beans and listeners, as the following example shows:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListenerMappings">
            <map>
                <entry key="testBean">
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

If you want to register a single `NotificationListener` instance for all of the beans that the enclosing `MBeanExporter` exports, you can use the special wildcard (`*`) as the key for an entry in the `notificationListenerMappings` property map, as the following example shows:

```
<property name="notificationListenerMappings">
    <map>
        <entry key="*">
            <bean class="com.example.ConsoleLoggingNotificationListener"/>
        </entry>
    </map>
</property>
```

If you need to do the inverse (that is, register a number of distinct listeners against an MBean), you must instead use the `notificationListeners` list property (in preference to the `notificationListenerMappings` property). This time, instead of configuring a `NotificationListener` for

a single MBean, we configure `NotificationListenerBean` instances. A `NotificationListenerBean` encapsulates a `NotificationListener` and the `ObjectName` (or `ObjectNames`) that it is to be registered against in an `MBeanServer`. The `NotificationListenerBean` also encapsulates a number of other properties, such as a `NotificationFilter` and an arbitrary handback object that can be used in advanced JMX notification scenarios.

The configuration when using `NotificationListenerBean` instances is not wildly different to what was presented previously, as the following example shows:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.NotificationListenerBean">
                    <constructor-arg>
                        <bean class="com.example.ConsoleLoggingNotificationListener"/>
                    </constructor-arg>
                    <property name="mappedObjectNames">
                        <list>
                            <value>bean:name=testBean1</value>
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

The preceding example is equivalent to the first notification example. Assume, then, that we want to be given a handback object every time a `Notification` is raised and that we also want to filter out extraneous `Notifications` by supplying a `NotificationFilter`. The following example accomplishes these goals:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean1"/>
                <entry key="bean:name=testBean2" value-ref="testBean2"/>
            </map>
        </property>
        <property name="notificationListeners">
            <list>
                <bean class="org.springframework.jmx.export.NotificationListenerBean">
                    <constructor-arg ref="customerNotificationListener"/>
                    <property name="mappedObjectNames">
                        <list>
                            <!-- handles notifications from two distinct MBeans -->
                            <value>bean:name=testBean1</value>
                            <value>bean:name=testBean2</value>
                        </list>
                    </property>
                    <property name="handback">
                        <bean class="java.lang.String">
                            <constructor-arg value="This could be anything..."/>
                        </bean>
                    </property>
                    <property name="notificationFilter"
ref="customerNotificationListener"/>
                </bean>
            </list>
        </property>
    </bean>

    <!-- implements both the NotificationListener and NotificationFilter interfaces
-->
    <bean id="customerNotificationListener"
class="com.example.ConsoleLoggingNotificationListener"/>

    <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

    <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="ANOTHER TEST"/>
        <property name="age" value="200"/>
    </bean>

</beans>
```

(For a full discussion of what a handback object is and, indeed, what a `NotificationFilter` is, see the

section of the JMX specification (1.2) entitled 'The JMX Notification Model'.)

Publishing Notifications

Spring provides support not only for registering to receive `Notifications` but also for publishing `Notifications`.

This section is really only relevant to Spring-managed beans that have been exposed as MBeans through an `MBeanExporter`. Any existing user-defined MBeans should use the standard JMX APIs for notification publication.

The key interface in Spring's JMX notification publication support is the `NotificationPublisher` interface (defined in the `org.springframework.jmx.export.notification` package). Any bean that is going to be exported as an MBean through an `MBeanExporter` instance can implement the related `NotificationPublisherAware` interface to gain access to a `NotificationPublisher` instance. The `NotificationPublisherAware` interface supplies an instance of a `NotificationPublisher` to the implementing bean through a simple setter method, which the bean can then use to publish `Notifications`.

As stated in the javadoc of the `NotificationPublisher` interface, managed beans that publish events through the `NotificationPublisher` mechanism are not responsible for the state management of notification listeners. Spring's JMX support takes care of handling all the JMX infrastructure issues. All you need to do, as an application developer, is implement the `NotificationPublisherAware` interface and start publishing events by using the supplied `NotificationPublisher` instance. Note that the `NotificationPublisher` is set after the managed bean has been registered with an `MBeanServer`.

Using a `NotificationPublisher` instance is quite straightforward. You create a JMX `Notification` instance (or an instance of an appropriate `Notification` subclass), populate the notification with the data pertinent to the event that is to be published, and invoke the `sendNotification(Notification)` on the `NotificationPublisher` instance, passing in the `Notification`.

In the following example, exported instances of the `JmxTestBean` publish a `NotificationEvent` every time the `add(int, int)` operation is invoked:

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add", this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher notificationPublisher)
{
        this.publisher = notificationPublisher;
    }

}
```

The `NotificationPublisher` interface and the machinery to get it all working is one of the nicer features of Spring's JMX support. It does, however, come with the price tag of coupling your classes to both Spring and JMX. As always, the advice here is to be pragmatic. If you need the functionality offered by the `NotificationPublisher` and you can accept the coupling to both Spring and JMX, then do so.

7.3.7. Further Resources

This section contains links to further resources about JMX:

- The JMX homepage at Oracle.
- The JMX specification (JSR-000003).
- The JMX Remote API specification (JSR-000160).
- The MX4J homepage. (MX4J is an open-source implementation of various JMX specs.)

7.4. Email

This section describes how to send email with the Spring Framework.

Library dependencies

The following JAR needs to be on the classpath of your application in order to use the Spring Framework's email library:

- The JavaMail / Jakarta Mail 1.6 library

This library is freely available on the web — for example, in Maven Central as `com.sun.mail:jakarta.mail`. Please make sure to use the latest 1.6.x version rather than Jakarta Mail 2.0 (which comes with a different package namespace).

The Spring Framework provides a helpful utility library for sending email that shields you from the specifics of the underlying mailing system and is responsible for low-level resource handling on behalf of the client.

The `org.springframework.mail` package is the root level package for the Spring Framework's email support. The central interface for sending emails is the `MailSender` interface. A simple value object that encapsulates the properties of a simple mail such as `from` and `to` (plus many others) is the `SimpleMailMessage` class. This package also contains a hierarchy of checked exceptions that provide a higher level of abstraction over the lower level mail system exceptions, with the root exception being `MailException`. See the javadoc for more information on the rich mail exception hierarchy.

The `org.springframework.mail.javamail.JavaMailSender` interface adds specialized JavaMail features, such as MIME message support to the `MailSender` interface (from which it inherits). `JavaMailSender` also provides a callback interface called `org.springframework.mail.javamail.MimeMessagePreparator` for preparing a `MimeMessage`.

7.4.1. Usage

Assume that we have a business interface called `OrderManager`, as the following example shows:

```
public interface OrderManager {

    void placeOrder(Order order);

}
```

Further assume that we have a requirement stating that an email message with an order number needs to be generated and sent to a customer who placed the relevant order.

Basic `MailSender` and `SimpleMailMessage` Usage

The following example shows how to use `MailSender` and `SimpleMailMessage` to send an email when

someone places an order:

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class SimpleOrderManager implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage templateMessage;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setTemplateMessage(SimpleMailMessage templateMessage) {
        this.templateMessage = templateMessage;
    }

    public void placeOrder(Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        // Create a thread safe "copy" of the template message and customize it
        SimpleMailMessage msg = new SimpleMailMessage(this.templateMessage);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            "Dear " + order.getCustomer().getFirstName()
                + order.getCustomer().getLastName()
                + ", thank you for placing order. Your order number is "
                + order.getOrderNumber());
        try {
            this.mailSender.send(msg);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }

}
```

The following example shows the bean definitions for the preceding code:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host" value="mail.mycompany.example"/>
</bean>

<!-- this is a template message that we can pre-load with default state -->
<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
    <property name="from" value="customerservice@mycompany.example"/>
    <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.SimpleOrderManager">
    <property name="mailSender" ref="mailSender"/>
    <property name="templateMessage" ref="templateMessage"/>
</bean>
```

Using `JavaMailSender` and `MimeMessagePreparator`

This section describes another implementation of `OrderManager` that uses the `MimeMessagePreparator` callback interface. In the following example, the `mailSender` property is of type `JavaMailSender` so that we are able to use the JavaMail `MimeMessage` class:

```
import jakarta.mail.Message;
import jakarta.mail.MessagingException;
import jakarta.mail.internet.InternetAddress;
import jakarta.mail.internet.MimeMessage;

import jakarta.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class SimpleOrderManager implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {
        // Do the business calculations...
        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePreparator() {
            public void prepare(MimeMessage mimeMessage) throws Exception {
                mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(order.getCustomer().getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress("mail@mycompany.example"));
                mimeMessage.setText("Dear " + order.getCustomer().getFirstName() + " "
+
                        order.getCustomer().getLastName() + ", thanks for your order.
" +
                        "Your order number is " + order.getOrderNumber() + ".");
            }
        };

        try {
            this.mailSender.send(preparator);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }

}
```

The mail code is a crosscutting concern and could well be a candidate for refactoring into a custom Spring AOP aspect, which could then be run at appropriate joinpoints on the `OrderManager` target.

The Spring Framework's mail support ships with the standard JavaMail implementation. See the relevant javadoc for more information.

7.4.2. Using the JavaMail `MimeMessageHelper`

A class that comes in pretty handy when dealing with JavaMail messages is `org.springframework.mail.javamail.MimeMessageHelper`, which shields you from having to use the verbose JavaMail API. Using the `MimeMessageHelper`, it is pretty easy to create a `MimeMessage`, as the following example shows:

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");

sender.send(message);
```

Sending Attachments and Inline Resources

Multipart email messages allow for both attachments and inline resources. Examples of inline resources include an image or a stylesheet that you want to use in your message but that you do not want displayed as an attachment.

Attachments

The following example shows you how to use the `MimeMessageHelper` to send an email with a single JPEG image attachment:

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

helper.setText("Check out this image!");

// let's attach the infamous windows Sample file (this time copied to c:/)
FileSystemResource file = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addAttachment("CoolImage.jpg", file);

sender.send(message);
```

Inline Resources

The following example shows you how to use the `MimeMessageHelper` to send an email with an inline image:

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML
helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addInline("identifier1234", res);

sender.send(message);
```

Inline resources are added to the `MimeMessage` by using the specified `Content-ID` (`identifier1234` in the above example). The order in which you add the text and the resource are very important. Be sure to first add the text and then the resources. If you are doing it the other way around, it does not work.

Creating Email Content by Using a Templating Library

The code in the examples shown in the previous sections explicitly created the content of the email message, by using methods calls such as `message.setText(..)`. This is fine for simple cases, and it is okay in the context of the aforementioned examples, where the intent was to show you the very basics of the API.

In your typical enterprise application, though, developers often do not create the content of email messages by using the previously shown approach for a number of reasons:

- Creating HTML-based email content in Java code is tedious and error prone.
- There is no clear separation between display logic and business logic.
- Changing the display structure of the email content requires writing Java code, recompiling, redeploying, and so on.

Typically, the approach taken to address these issues is to use a template library (such as FreeMarker) to define the display structure of email content. This leaves your code tasked only with creating the data that is to be rendered in the email template and sending the email. It is definitely a best practice when the content of your email messages becomes even moderately complex, and, with the Spring Framework's support classes for FreeMarker, it becomes quite easy to do.

7.5. Task Execution and Scheduling

The Spring Framework provides abstractions for the asynchronous execution and scheduling of tasks with the `TaskExecutor` and `TaskScheduler` interfaces, respectively. Spring also features implementations of those interfaces that support thread pools or delegation to CommonJ within an application server environment. Ultimately, the use of these implementations behind the common interfaces abstracts away the differences between Java SE 5, Java SE 6, and Jakarta EE environments.

Spring also features integration classes to support scheduling with the `Timer` (part of the JDK since 1.3) and the Quartz Scheduler (https://www.quartz-scheduler.org/). You can set up both of those schedulers by using a `FactoryBean` with optional references to `Timer` or `Trigger` instances, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the `Timer` is available that lets you invoke a method of an existing target object (analogous to the normal `MethodInvokingFactoryBean` operation).

7.5.1. The Spring `TaskExecutor` Abstraction

Executors are the JDK name for the concept of thread pools. The “executor” naming is due to the fact that there is no guarantee that the underlying implementation is actually a pool. An executor may be single-threaded or even synchronous. Spring’s abstraction hides implementation details between the Java SE and Jakarta EE environments.

Spring’s `TaskExecutor` interface is identical to the `java.util.concurrent.Executor` interface. In fact, originally, its primary reason for existence was to abstract away the need for Java 5 when using thread pools. The interface has a single method (`execute(Runnable task)`) that accepts a task for execution based on the semantics and configuration of the thread pool.

The `TaskExecutor` was originally created to give other Spring components an abstraction for thread pooling where needed. Components such as the `ApplicationEventMulticaster`, JMS’s `AbstractMessageListenerContainer`, and Quartz integration all use the `TaskExecutor` abstraction to pool threads. However, if your beans need thread pooling behavior, you can also use this abstraction for your own needs.

`TaskExecutor` Types

Spring includes a number of pre-built implementations of `TaskExecutor`. In all likelihood, you should never need to implement your own. The variants that Spring provides are as follows:

- `SyncTaskExecutor`: This implementation does not run invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where multi-threading is not necessary, such as in simple test cases.
- `SimpleAsyncTaskExecutor`: This implementation does not reuse any threads. Rather, it starts up a new thread for each invocation. However, it does support a concurrency limit that blocks any invocations that are over the limit until a slot has been freed up. If you are looking for true pooling, see `ThreadPoolTaskExecutor`, later in this list.
- `ConcurrentTaskExecutor`: This implementation is an adapter for a `java.util.concurrent.Executor` instance. There is an alternative (`ThreadPoolTaskExecutor`) that exposes the `Executor`

configuration parameters as bean properties. There is rarely a need to use `ConcurrentTaskExecutor` directly. However, if the `ThreadPoolTaskExecutor` is not flexible enough for your needs, `ConcurrentTaskExecutor` is an alternative.

- `ThreadPoolTaskExecutor`: This implementation is most commonly used. It exposes bean properties for configuring a `java.util.concurrent.ThreadPoolExecutor` and wraps it in a `TaskExecutor`. If you need to adapt to a different kind of `java.util.concurrent.Executor`, we recommend that you use a `ConcurrentTaskExecutor` instead.
- `DefaultManagedTaskExecutor`: This implementation uses a JNDI-obtained `ManagedExecutorService` in a JSR-236 compatible runtime environment (such as a Jakarta EE application server), replacing a CommonJ WorkManager for that purpose.

Using a `TaskExecutor`

Spring's `TaskExecutor` implementations are used as simple JavaBeans. In the following example, we define a bean that uses the `ThreadPoolTaskExecutor` to asynchronously print out a set of messages:

```
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

    private class MessagePrinterTask implements Runnable {

        private String message;

        public MessagePrinterTask(String message) {
            this.message = message;
        }

        public void run() {
            System.out.println(message);
        }
    }

    private TaskExecutor taskExecutor;

    public TaskExecutorExample(TaskExecutor taskExecutor) {
        this.taskExecutor = taskExecutor;
    }

    public void printMessages() {
        for(int i = 0; i < 25; i++) {
            taskExecutor.execute(new MessagePrinterTask("Message" + i));
        }
    }
}
```

As you can see, rather than retrieving a thread from the pool and executing it yourself, you add your `Runnable` to the queue. Then the `TaskExecutor` uses its internal rules to decide when the task

gets run.

To configure the rules that the `TaskExecutor` uses, we expose simple bean properties:

```
<bean id="taskExecutor"
class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
    <property name="corePoolSize" value="5"/>
    <property name="maxPoolSize" value="10"/>
    <property name="queueCapacity" value="25"/>
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
    <constructor-arg ref="taskExecutor"/>
</bean>
```

7.5.2. The Spring `TaskScheduler` Abstraction

In addition to the `TaskExecutor` abstraction, Spring 3.0 introduced a `TaskScheduler` with a variety of methods for scheduling tasks to run at some point in the future. The following listing shows the `TaskScheduler` interface definition:

```
public interface TaskScheduler {

    ScheduledFuture schedule(Runnable task, Trigger trigger);

    ScheduledFuture schedule(Runnable task, Instant startTime);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Instant startTime, Duration
period);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Duration period);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Instant startTime, Duration
delay);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Duration delay);
```

The simplest method is the one named `schedule` that takes only a `Runnable` and an `Instant`. That causes the task to run once after the specified time. All of the other methods are capable of scheduling tasks to run repeatedly. The fixed-rate and fixed-delay methods are for simple, periodic execution, but the method that accepts a `Trigger` is much more flexible.

`Trigger` Interface

The `Trigger` interface is essentially inspired by JSR-236 which, as of Spring 3.0, was not yet officially implemented. The basic idea of the `Trigger` is that execution times may be determined based on past execution outcomes or even arbitrary conditions. If these determinations do take into account the outcome of the preceding execution, that information is available within a `TriggerContext`. The

`Trigger` interface itself is quite simple, as the following listing shows:

```
public interface Trigger {

    Date nextExecutionTime(TriggerContext triggerContext);
}
```

The `TriggerContext` is the most important part. It encapsulates all of the relevant data and is open for extension in the future, if necessary. The `TriggerContext` is an interface (a `SimpleTriggerContext` implementation is used by default). The following listing shows the available methods for `Trigger` implementations.

```
public interface TriggerContext {

    Date lastScheduledExecutionTime();

    Date lastActualExecutionTime();

    Date lastCompletionTime();
}
```

`Trigger` Implementations

Spring provides two implementations of the `Trigger` interface. The most interesting one is the `CronTrigger`. It enables the scheduling of tasks based on cron expressions. For example, the following task is scheduled to run 15 minutes past each hour but only during the 9-to-5 "business hours" on weekdays:

```
scheduler.schedule(task, new CronTrigger("0 15 9-17 * * MON-FRI"));
```

The other implementation is a `PeriodicTrigger` that accepts a fixed period, an optional initial delay value, and a boolean to indicate whether the period should be interpreted as a fixed-rate or a fixed-delay. Since the `TaskScheduler` interface already defines methods for scheduling tasks at a fixed rate or with a fixed delay, those methods should be used directly whenever possible. The value of the `PeriodicTrigger` implementation is that you can use it within components that rely on the `Trigger` abstraction. For example, it may be convenient to allow periodic triggers, cron-based triggers, and even custom trigger implementations to be used interchangeably. Such a component could take advantage of dependency injection so that you can configure such `Triggers` externally and, therefore, easily modify or extend them.

`TaskScheduler` implementations

As with Spring's `TaskExecutor` abstraction, the primary benefit of the `TaskScheduler` arrangement is that an application's scheduling needs are decoupled from the deployment environment. This abstraction level is particularly relevant when deploying to an application server environment where threads should not be created directly by the application itself. For such scenarios, Spring

provides a `TimerManagerTaskScheduler` that delegates to a CommonJ `TimerManager` on WebLogic or WebSphere as well as a more recent `DefaultManagedTaskScheduler` that delegates to a JSR-236 `ManagedScheduledExecutorService` in a Jakarta EE environment. Both are typically configured with a JNDI lookup.

Whenever external thread management is not a requirement, a simpler alternative is a local `ScheduledExecutorService` setup within the application, which can be adapted through Spring's `ConcurrentTaskScheduler`. As a convenience, Spring also provides a `ThreadPoolTaskScheduler`, which internally delegates to a `ScheduledExecutorService` to provide common bean-style configuration along the lines of `ThreadPoolTaskExecutor`. These variants work perfectly fine for locally embedded thread pool setups in lenient application server environments, as well — in particular on Tomcat and Jetty.

7.5.3. Annotation Support for Scheduling and Asynchronous Execution

Spring provides annotation support for both task scheduling and asynchronous method execution.

Enable Scheduling Annotations

To enable support for `@Scheduled` and `@Async` annotations, you can add `@EnableScheduling` and `@EnableAsync` to one of your `@Configuration` classes, as the following example shows:

```
@Configuration
@EnableAsync
@EnableScheduling
public class AppConfig {
}
```

You can pick and choose the relevant annotations for your application. For example, if you need only support for `@Scheduled`, you can omit `@EnableAsync`. For more fine-grained control, you can additionally implement the `SchedulingConfigurer` interface, the `AsyncConfigurer` interface, or both. See the `SchedulingConfigurer` and `AsyncConfigurer` javadoc for full details.

If you prefer XML configuration, you can use the `<task:annotation-driven>` element, as the following example shows:

```
<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>
<task:executor id="myExecutor" pool-size="5"/>
<task:scheduler id="myScheduler" pool-size="10"/>
```

Note that, with the preceding XML, an executor reference is provided for handling those tasks that correspond to methods with the `@Async` annotation, and the scheduler reference is provided for managing those methods annotated with `@Scheduled`.

The default advice mode for processing `@Async` annotations is `proxy` which allows for interception of calls through the proxy only. Local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to `aspectj` mode in combination with compile-time or load-time weaving.

The `@Scheduled` annotation

You can add the `@Scheduled` annotation to a method, along with trigger metadata. For example, the following method is invoked every five seconds (5000 milliseconds) with a fixed delay, meaning that the period is measured from the completion time of each preceding invocation.

```
@Scheduled(fixedDelay = 5000)
public void doSomething() {
    // something that should run periodically
}
```

By default, milliseconds will be used as the time unit for fixed delay, fixed rate, and initial delay values. If you would like to use a different time unit such as seconds or minutes, you can configure this via the `timeUnit` attribute in `@Scheduled`.

For example, the previous example can also be written as follows.

```
@Scheduled(fixedDelay = 5, timeUnit = TimeUnit.SECONDS)
public void doSomething() {
    // something that should run periodically
}
```

If you need a fixed-rate execution, you can use the `fixedRate` attribute within the annotation. The following method is invoked every five seconds (measured between the successive start times of each invocation).

```
@Scheduled(fixedRate = 5, timeUnit = TimeUnit.SECONDS)
public void doSomething() {
    // something that should run periodically
}
```

For fixed-delay and fixed-rate tasks, you can specify an initial delay by indicating the amount of time to wait before the first execution of the method, as the following `fixedRate` example shows.

```
@Scheduled(initialDelay = 1000, fixedRate = 5000)
public void doSomething() {
    // something that should run periodically
}
```

If simple periodic scheduling is not expressive enough, you can provide a cron expression. The following example runs only on weekdays:

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should run on weekdays only
}
```

You can also use the `zone` attribute to specify the time zone in which the cron expression is resolved.

Notice that the methods to be scheduled must have void returns and must not accept any arguments. If the method needs to interact with other objects from the application context, those would typically have been provided through dependency injection.

As of Spring Framework 4.3, `@Scheduled` methods are supported on beans of any scope.

Make sure that you are not initializing multiple instances of the same `@Scheduled` annotation class at runtime, unless you do want to schedule callbacks to each such instance. Related to this, make sure that you do not use `@Configurable` on bean classes that are annotated with `@Scheduled` and registered as regular Spring beans with the container. Otherwise, you would get double initialization (once through the container and once through the `@Configurable` aspect), with the consequence of each `@Scheduled` method being invoked twice.

The `@Async` annotation

You can provide the `@Async` annotation on a method so that invocation of that method occurs asynchronously. In other words, the caller returns immediately upon invocation, while the actual execution of the method occurs in a task that has been submitted to a Spring `TaskExecutor`. In the simplest case, you can apply the annotation to a method that returns `void`, as the following example shows:

```
@Async
void doSomething() {
    // this will be run asynchronously
}
```

Unlike the methods annotated with the `@Scheduled` annotation, these methods can expect arguments, because they are invoked in the "normal" way by callers at runtime rather than from a scheduled task being managed by the container. For example, the following code is a legitimate application of the `@Async` annotation:

```
@Async
void doSomething(String s) {
    // this will be run asynchronously
}
```

Even methods that return a value can be invoked asynchronously. However, such methods are required to have a `Future`-typed return value. This still provides the benefit of asynchronous execution so that the caller can perform other tasks prior to calling `get()` on that `Future`. The following example shows how to use `@Async` on a method that returns a value:

```
@Async
Future<String> returnSomething(int i) {
    // this will be run asynchronously
}
```

`@Async` methods may not only declare a regular `java.util.concurrent.Future` return type but also Spring's `org.springframework.util.concurrent.ListenableFuture` or, as of Spring 4.2, JDK 8's `java.util.concurrent.CompletableFuture`, for richer interaction with the asynchronous task and for immediate composition with further processing steps.

You can not use `@Async` in conjunction with lifecycle callbacks such as `@PostConstruct`. To asynchronously initialize Spring beans, you currently have to use a separate initializing Spring bean that then invokes the `@Async` annotated method on the target, as the following example shows:

```
public class SampleBeanImpl implements SampleBean {

    @Async
    void doSomething() {
        // ...
    }

}

public class SampleBeanInitializer {

    private final SampleBean bean;

    public SampleBeanInitializer(SampleBean bean) {
        this.bean = bean;
    }

    @PostConstruct
    public void initialize() {
        bean.doSomething();
    }

}
```

There is no direct XML equivalent for `@Async`, since such methods should be designed for asynchronous execution in the first place, not externally re-declared to be asynchronous. However, you can manually set up Spring's `AsyncExecutionInterceptor` with Spring AOP, in combination with a custom pointcut.

Executor Qualification with `@Async`

By default, when specifying `@Async` on a method, the executor that is used is the one configured when enabling async support, i.e. the "annotation-driven" element if you are using XML or your `AsyncConfigurer` implementation, if any. However, you can use the `value` attribute of the `@Async` annotation when you need to indicate that an executor other than the default should be used when executing a given method. The following example shows how to do so:

```
@Async("otherExecutor")
void doSomething(String s) {
    // this will be run asynchronously by "otherExecutor"
}
```

In this case, `"otherExecutor"` can be the name of any `Executor` bean in the Spring container, or it may be the name of a qualifier associated with any `Executor` (for example, as specified with the `<qualifier>` element or Spring's `@Qualifier` annotation).

Exception Management with `@Async`

When an `@Async` method has a `Future`-typed return value, it is easy to manage an exception that was thrown during the method execution, as this exception is thrown when calling `get` on the `Future` result. With a `void` return type, however, the exception is uncaught and cannot be transmitted. You can provide an `AsyncUncaughtExceptionHandler` to handle such exceptions. The following example shows how to do so:

```
public class MyAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler
{

    @Override
    public void handleUncaughtException(Throwable ex, Method method, Object... params)
{
        // handle exception
    }
}
```

By default, the exception is merely logged. You can define a custom `AsyncUncaughtExceptionHandler` by using `AsyncConfigurer` or the `<task:annotation-driven/>` XML element.

7.5.4. The `task` Namespace

As of version 3.0, Spring includes an XML namespace for configuring `TaskExecutor` and `TaskScheduler` instances. It also provides a convenient way to configure tasks to be scheduled with a trigger.

The 'scheduler' Element

The following element creates a `ThreadPoolTaskScheduler` instance with the specified thread pool size:

```
<task:scheduler id="scheduler" pool-size="10"/>
```

The value provided for the `id` attribute is used as the prefix for thread names within the pool. The `scheduler` element is relatively straightforward. If you do not provide a `pool-size` attribute, the default thread pool has only a single thread. There are no other configuration options for the scheduler.

The `executor` Element

The following creates a `ThreadPoolTaskExecutor` instance:

```
<task:executor id="executor" pool-size="10"/>
```

As with the scheduler shown in the previous section, the value provided for the `id` attribute is used as the prefix for thread names within the pool. As far as the pool size is concerned, the `executor`

element supports more configuration options than the `scheduler` element. For one thing, the thread pool for a `ThreadPoolTaskExecutor` is itself more configurable. Rather than only a single size, an executor's thread pool can have different values for the core and the max size. If you provide a single value, the executor has a fixed-size thread pool (the core and max sizes are the same). However, the `executor` element's `pool-size` attribute also accepts a range in the form of `min-max`. The following example sets a minimum value of `5` and a maximum value of `25`:

```
<task:executor
        id="executorWithPoolSizeRange"
        pool-size="5-25"
        queue-capacity="100"/>
```

In the preceding configuration, a `queue-capacity` value has also been provided. The configuration of the thread pool should also be considered in light of the executor's queue capacity. For the full description of the relationship between pool size and queue capacity, see the documentation for `ThreadPoolExecutor`. The main idea is that, when a task is submitted, the executor first tries to use a free thread if the number of active threads is currently less than the core size. If the core size has been reached, the task is added to the queue, as long as its capacity has not yet been reached. Only then, if the queue's capacity has been reached, does the executor create a new thread beyond the core size. If the max size has also been reached, then the executor rejects the task.

By default, the queue is unbounded, but this is rarely the desired configuration, because it can lead to `OutOfMemoryErrors` if enough tasks are added to that queue while all pool threads are busy. Furthermore, if the queue is unbounded, the max size has no effect at all. Since the executor always tries the queue before creating a new thread beyond the core size, a queue must have a finite capacity for the thread pool to grow beyond the core size (this is why a fixed-size pool is the only sensible case when using an unbounded queue).

Consider the case, as mentioned above, when a task is rejected. By default, when a task is rejected, a thread pool executor throws a `TaskRejectedException`. However, the rejection policy is actually configurable. The exception is thrown when using the default rejection policy, which is the `AbortPolicy` implementation. For applications where some tasks can be skipped under heavy load, you can instead configure either `DiscardPolicy` or `DiscardOldestPolicy`. Another option that works well for applications that need to throttle the submitted tasks under heavy load is the `CallerRunsPolicy`. Instead of throwing an exception or discarding tasks, that policy forces the thread that is calling the submit method to run the task itself. The idea is that such a caller is busy while running that task and not able to submit other tasks immediately. Therefore, it provides a simple way to throttle the incoming load while maintaining the limits of the thread pool and queue. Typically, this allows the executor to "catch up" on the tasks it is handling and thereby frees up some capacity on the queue, in the pool, or both. You can choose any of these options from an enumeration of values available for the `rejection-policy` attribute on the `executor` element.

The following example shows an `executor` element with a number of attributes to specify various behaviors:

```
<task:executor
        id="executorWithCallerRunsPolicy"
        pool-size="5-25"
        queue-capacity="100"
        rejection-policy="CALLER_RUNS"/>
```

Finally, the `keep-alive` setting determines the time limit (in seconds) for which threads may remain idle before being stopped. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads get stopped. A time value of zero causes excess threads to stop immediately after executing a task without remaining follow-up work in the task queue. The following example sets the `keep-alive` value to two minutes:

```
<task:executor
        id="executorWithKeepAlive"
        pool-size="5-25"
        keep-alive="120"/>
```

The 'scheduled-tasks' Element

The most powerful feature of Spring's task namespace is the support for configuring tasks to be scheduled within a Spring Application Context. This follows an approach similar to other "method-invokers" in Spring, such as that provided by the JMS namespace for configuring message-driven POJOs. Basically, a `ref` attribute can point to any Spring-managed object, and the `method` attribute provides the name of a method to be invoked on that object. The following listing shows a simple example:

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

The scheduler is referenced by the outer element, and each individual task includes the configuration of its trigger metadata. In the preceding example, that metadata defines a periodic trigger with a fixed delay indicating the number of milliseconds to wait after each task execution has completed. Another option is `fixed-rate`, indicating how often the method should be run regardless of how long any previous execution takes. Additionally, for both `fixed-delay` and `fixed-rate` tasks, you can specify an 'initial-delay' parameter, indicating the number of milliseconds to wait before the first execution of the method. For more control, you can instead provide a `cron` attribute to provide a cron expression. The following example shows these other options:

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="beanA" method="methodA" fixed-delay="5000" initial-
delay="1000"/>
    <task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>
    <task:scheduled ref="beanC" method="methodC" cron="*/5 * * * * MON-FRI"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>
```

7.5.5. Cron Expressions

All Spring cron expressions have to conform to the same format, whether you are using them in `@Scheduled` annotations, `task:scheduled-tasks` elements, or someplace else. A well-formed cron expression, such as `* * * * * *`, consists of six space-separated time and date fields, each with its own range of valid values:

```
 ┌──────────────────────────── second (0-59)
 │ ┌────────────────────────── minute (0 - 59)
 │ │ ┌──────────────────────── hour (0 - 23)
 │ │ │ ┌────────────────────── day of the month (1 - 31)
 │ │ │ │ ┌──────────────────── month (1 - 12) (or JAN-DEC)
 │ │ │ │ │ ┌────────────────── day of the week (0 - 7)
 │ │ │ │ │ │          (0 or 7 is Sunday, or MON-SUN)
 │ │ │ │ │ │
* * * * * *
```

There are some rules that apply:

- A field may be an asterisk (`*`), which always stands for “first-last”. For the day-of-the-month or day-of-the-week fields, a question mark (`?`) may be used instead of an asterisk.
- Commas (`,`) are used to separate items of a list.
- Two numbers separated with a hyphen (`-`) express a range of numbers. The specified range is inclusive.
- Following a range (or `*`) with `/` specifies the interval of the number’s value through the range.
- English names can also be used for the month and day-of-week fields. Use the first three letters of the particular day or month (case does not matter).
- The day-of-month and day-of-week fields can contain a `L` character, which has a different meaning
  - In the day-of-month field, `L` stands for *the last day of the month*. If followed by a negative offset (that is, `L-n`), it means `n`*th-to-last day of the month*.
  - In the day-of-week field, `L` stands for *the last day of the week*. If prefixed by a number or three-letter name (`dL` or `DDDL`), it means *the last day of week (`d` or `DDD`) in the month*.
- The day-of-month field can be `nW`, which stands for *the nearest weekday to day of the month* `n`. If `n` falls on Saturday, this yields the Friday before it. If `n` falls on Sunday, this yields the Monday

after, which also happens if `n` is `1` and falls on a Saturday (that is: `1W` stands for *the first weekday of the month*).

- If the day-of-month field is `LW`, it means *the last weekday of the month*.
- The day-of-week field can be `d#n` (or `DDD#n`), which stands for *the `n`th day of week `d` (or `DDD`) in the month*.

Here are some examples:

Cron Expression	Meaning
`0 0 * * * *`	top of every hour of every day
`*/10 * * * * *`	every ten seconds
`0 0 8-10 * * *`	8, 9 and 10 o'clock of every day
`0 0 6,19 * * *`	6:00 AM and 7:00 PM every day
`0 0/30 8-10 * * *`	8:00, 8:30, 9:00, 9:30, 10:00 and 10:30 every day
`0 0 9-17 * * MON-FRI`	on the hour nine-to-five weekdays
`0 0 0 25 DEC ?`	every Christmas Day at midnight
`0 0 0 L * *`	last day of the month at midnight
`0 0 0 L-3 * *`	third-to-last day of the month at midnight
`0 0 0 * * 5L`	last Friday of the month at midnight
`0 0 0 * * THUL`	last Thursday of the month at midnight
`0 0 0 1W * *`	first weekday of the month at midnight
`0 0 0 LW * *`	last weekday of the month at midnight
`0 0 0 ? * 5#2`	the second Friday in the month at midnight
`0 0 0 ? * MON#1`	the first Monday in the month at midnight

Macros

Expressions such as `0 0 * * * *` are hard for humans to parse and are, therefore, hard to fix in case of bugs. To improve readability, Spring supports the following macros, which represent commonly used sequences. You can use these macros instead of the six-digit value, thus: `@Scheduled(cron = "@hourly")`.

Macro	Meaning
`@yearly` (or `@annually`)	once a year (`0 0 0 1 1 *`)
`@monthly`	once a month (`0 0 0 1 * *`)
`@weekly`	once a week (`0 0 0 * * 0`)
`@daily` (or `@midnight`)	once a day (`0 0 0 * * *`), or
`@hourly`	once an hour, (`0 0 * * * *`)

7.5.6. Using the Quartz Scheduler

Quartz uses `Trigger`, `Job`, and `JobDetail` objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, see https://www.quartz-scheduler.org/. For convenience purposes, Spring offers a couple of classes that simplify using Quartz within Spring-based applications.

Using the `JobDetailFactoryBean`

Quartz `JobDetail` objects contain all the information needed to run a job. Spring provides a `JobDetailFactoryBean`, which provides bean-style properties for XML configuration purposes. Consider the following example:

```
<bean name="exampleJob"
class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
    <property name="jobClass" value="example.ExampleJob"/>
    <property name="jobDataAsMap">
        <map>
            <entry key="timeout" value="5"/>
        </map>
    </property>
</bean>
```

The job detail configuration has all the information it needs to run the job (`ExampleJob`). The timeout is specified in the job data map. The job data map is available through the `JobExecutionContext` (passed to you at execution time), but the `JobDetail` also gets its properties from the job data mapped to properties of the job instance. So, in the following example, the `ExampleJob` contains a bean property named `timeout`, and the `JobDetail` has it applied automatically:

```
package example;

public class ExampleJob extends QuartzJobBean {

    private int timeout;

    /**
     * Setter called after the ExampleJob is instantiated
     * with the value from the JobDetailFactoryBean (5)
     */
    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    protected void executeInternal(JobExecutionContext ctx) throws
JobExecutionException {
        // do the actual work
    }
}
```

All additional properties from the job data map are available to you as well.

By using the `name` and `group` properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the bean name of the `JobDetailFactoryBean` (`exampleJob` in the preceding example above).

Using the `MethodInvokingJobDetailFactoryBean`

Often you merely need to invoke a method on a specific object. By using the `MethodInvokingJobDetailFactoryBean`, you can do exactly this, as the following example shows:

```
<bean id="jobDetail"
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
</bean>
```

The preceding example results in the `doIt` method being called on the `exampleBusinessObject` method, as the following example shows:

```
public class ExampleBusinessObject {

    // properties and collaborators

    public void doIt() {
        // do the actual work
    }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

By using the `MethodInvokingJobDetailFactoryBean`, you need not create one-line jobs that merely invoke a method. You need only create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same `JobDetail`, it is possible that, before the first job has finished, the second one starts. If `JobDetail` classes implement the `Stateful` interface, this does not happen. The second job does not start before the first one has finished. To make jobs resulting from the `MethodInvokingJobDetailFactoryBean` be non-concurrent, set the `concurrent` flag to `false`, as the following example shows:

```
<bean id="jobDetail"
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject"/>
    <property name="targetMethod" value="doIt"/>
    <property name="concurrent" value="false"/>
</bean>
```

By default, jobs will run in a concurrent fashion.

Wiring up Jobs by Using Triggers and `SchedulerFactoryBean`

We have created job details and jobs. We have also reviewed the convenience bean that lets you invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done by using triggers and a `SchedulerFactoryBean`. Several triggers are available within Quartz, and Spring offers two Quartz `FactoryBean` implementations with convenient defaults: `CronTriggerFactoryBean` and `SimpleTriggerFactoryBean`.

Triggers need to be scheduled. Spring offers a `SchedulerFactoryBean` that exposes triggers to be set as properties. `SchedulerFactoryBean` schedules the actual jobs with those triggers.

The following listing uses both a `SimpleTriggerFactoryBean` and a `CronTriggerFactoryBean`:

```
<bean id="simpleTrigger"
class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail"/>
    <!-- 10 seconds -->
    <property name="startDelay" value="10000"/>
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000"/>
</bean>

<bean id="cronTrigger"
class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
    <property name="jobDetail" ref="exampleJob"/>
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?"/>
</bean>
```

The preceding example sets up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one running every morning at 6 AM. To finalize everything, we need to set up the `SchedulerFactoryBean`, as the following example shows:

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger"/>
            <ref bean="simpleTrigger"/>
        </list>
    </property>
</bean>
```

More properties are available for the `SchedulerFactoryBean`, such as the calendars used by the job details, properties to customize Quartz with, and a Spring-provided JDBC DataSource. See the `SchedulerFactoryBean` javadoc for more information.

`SchedulerFactoryBean` also recognizes a `quartz.properties` file in the classpath, based on Quartz property keys, as with regular Quartz configuration. Please note that many `SchedulerFactoryBean` settings interact with common Quartz settings in the properties file; it is therefore not recommended to specify values at both levels. For example, do not set an "org.quartz.jobStore.class" property if you mean to rely on a Spring-provided DataSource, or specify an `org.springframework.scheduling.quartz.LocalDataSourceJobStore` variant which is a full-fledged replacement for the standard `org.quartz.impl.jdbcjobstore.JobStoreTX`.

7.6. Cache Abstraction

Since version 3.1, the Spring Framework provides support for transparently adding caching to an existing Spring application. Similar to the transaction support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

In Spring Framework 4.1, the cache abstraction was significantly extended with support for JSR-107 annotations and more customization options.

7.6.1. Understanding the Cache Abstraction

Cache vs Buffer

The terms, “buffer” and “cache,” tend to be used interchangeably. Note, however, that they represent different things. Traditionally, a buffer is used as an intermediate temporary store for data between a fast and a slow entity. As one party would have to wait for the other (which affects performance), the buffer alleviates this by allowing entire blocks of data to move at once rather than in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are visible to at least one party that is aware of it.

A cache, on the other hand, is, by definition, hidden, and neither party is aware that caching occurs. It also improves performance but does so by letting the same data be read multiple times in a fast fashion.

You can find a further explanation of the differences between a buffer and a cache here.

At its core, the cache abstraction applies caching to Java methods, thus reducing the number of executions based on the information available in the cache. That is, each time a targeted method is invoked, the abstraction applies a caching behavior that checks whether the method has been already invoked for the given arguments. If it has been invoked, the cached result is returned without having to invoke the actual method. If the method has not been invoked, then it is invoked, and the result is cached and returned to the user so that, the next time the method is invoked, the cached result is returned. This way, expensive methods (whether CPU- or IO-bound) can be invoked only once for a given set of parameters and the result reused without having to actually invoke the method again. The caching logic is applied transparently without any interference to the invoker.

This approach works only for methods that are guaranteed to return the same output (result) for a given input (or arguments) no matter how many times they are invoked.

The caching abstraction provides other cache-related operations, such as the ability to update the content of the cache or to remove one or all entries. These are useful if the cache deals with data that can change during the course of the application.

As with other services in the Spring Framework, the caching service is an abstraction (not a cache implementation) and requires the use of actual storage to store the cache data — that is, the abstraction frees you from having to write the caching logic but does not provide the actual data store. This abstraction is materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

Spring provides a few implementations of that abstraction: JDK `java.util.concurrent.ConcurrentMap` based caches, Gemfire cache, Caffeine, and JSR-107 compliant caches (such as Ehcache 3.x). See Plugging-in Different Back-end Caches for more information on plugging in other cache stores and providers.

The caching abstraction has no special handling for multi-threaded and multi-process environments, as such features are handled by the cache implementation.

If you have a multi-process environment (that is, an application deployed on several nodes), you need to configure your cache provider accordingly. Depending on your use cases, a copy of the same data on several nodes can be enough. However, if you change the data during the course of the application, you may need to enable other propagation mechanisms.

Caching a particular item is a direct equivalent of the typical get-if-not-found-then-proceed-and-put-eventually code blocks found with programmatic cache interaction. No locks are applied, and several threads may try to load the same item concurrently. The same applies to eviction. If several threads are trying to update or evict data concurrently, you may use stale data. Certain cache providers offer advanced features in that area. See the documentation of your cache provider for more details.

To use the cache abstraction, you need to take care of two aspects:

- Caching declaration: Identify the methods that need to be cached and their policies.
- Cache configuration: The backing cache where the data is stored and from which it is read.

7.6.2. Declarative Annotation-based Caching

For caching declaration, Spring's caching abstraction provides a set of Java annotations:

- `@Cacheable`: Triggers cache population.
- `@CacheEvict`: Triggers cache eviction.
- `@CachePut`: Updates the cache without interfering with the method execution.
- `@Caching`: Regroups multiple cache operations to be applied on a method.
- `@CacheConfig`: Shares some common cache-related settings at class-level.

The `@Cacheable` Annotation

As the name implies, you can use `@Cacheable` to demarcate methods that are cacheable — that is, methods for which the result is stored in the cache so that, on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually invoke the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method, as the following example shows:

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

In the preceding snippet, the `findBook` method is associated with the cache named `books`. Each time the method is called, the cache is checked to see whether the invocation has already been run and does not have to be repeated. While in most cases, only one cache is declared, the annotation lets multiple names be specified so that more than one cache is being used. In this case, each of the caches is checked before invoking the method — if at least one cache is hit, the associated value is returned.

All the other caches that do not contain the value are also updated, even though the cached method was not actually invoked.

The following example uses `@Cacheable` on the `findBook` method with multiple caches:

```
@Cacheable({"books", "isbns"})
public Book findBook(ISBN isbn) {...}
```

Default Key Generation

Since caches are essentially key-value stores, each invocation of a cached method needs to be translated into a suitable key for cache access. The caching abstraction uses a simple `KeyGenerator` based on the following algorithm:

- If no params are given, return `SimpleKey.EMPTY`.
- If only one param is given, return that instance.
- If more than one param is given, return a `SimpleKey` that contains all parameters.

This approach works well for most use-cases, as long as parameters have natural keys and implement valid `hashCode()` and `equals()` methods. If that is not the case, you need to change the strategy.

To provide a different default key generator, you need to implement the `org.springframework.cache.interceptor.KeyGenerator` interface.

The default key generation strategy changed with the release of Spring 4.0. Earlier versions of Spring used a key generation strategy that, for multiple key parameters, considered only the `hashCode()` of parameters and not `equals()`. This could cause unexpected key collisions (see SPR-10237 for background). The new `SimpleKeyGenerator` uses a compound key for such scenarios.

If you want to keep using the previous key strategy, you can configure the deprecated `org.springframework.cache.interceptor.DefaultKeyGenerator` class or create a custom hash-based `KeyGenerator` implementation.

Custom Key Generation Declaration

Since caching is generic, the target methods are quite likely to have various signatures that cannot be readily mapped on top of the cache structure. This tends to become obvious when the target method has multiple arguments out of which only some are suitable for caching (while the rest are used only by the method logic). Consider the following example:

```
@Cacheable("books")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

At first glance, while the two `boolean` arguments influence the way the book is found, they are no

use for the cache. Furthermore, what if only one of the two is important while the other is not?

For such cases, the `@Cacheable` annotation lets you specify how the key is generated through its `key` attribute. You can use SpEL to pick the arguments of interest (or their nested properties), perform operations, or even invoke arbitrary methods without having to write any code or implement any interface. This is the recommended approach over the default generator, since methods tend to be quite different in signatures as the code base grows. While the default strategy might work for some methods, it rarely works for all methods.

The following examples use various SpEL declarations (if you are not familiar with SpEL, do yourself a favor and read Spring Expression Language):

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="#isbn.rawNumber")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="T(someType).hash(#isbn)")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The preceding snippets show how easy it is to select a certain argument, one of its properties, or even an arbitrary (static) method.

If the algorithm responsible for generating the key is too specific or if it needs to be shared, you can define a custom `keyGenerator` on the operation. To do so, specify the name of the `KeyGenerator` bean implementation to use, as the following example shows:

```
@Cacheable(cacheNames="books", keyGenerator="myKeyGenerator")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The `key` and `keyGenerator` parameters are mutually exclusive and an operation that specifies both results in an exception.

Default Cache Resolution

The caching abstraction uses a simple `CacheResolver` that retrieves the caches defined at the operation level by using the configured `CacheManager`.

To provide a different default cache resolver, you need to implement the `org.springframework.cache.interceptor.CacheResolver` interface.

Custom Cache Resolution

The default cache resolution fits well for applications that work with a single `CacheManager` and have no complex cache resolution requirements.

For applications that work with several cache managers, you can set the `cacheManager` to use for

each operation, as the following example shows:

```
@Cacheable(cacheNames="books", cacheManager="anotherCacheManager") ①
public Book findBook(ISBN isbn) {...}
```

① Specifying `anotherCacheManager`.

You can also replace the `CacheResolver` entirely in a fashion similar to that of replacing key generation. The resolution is requested for every cache operation, letting the implementation actually resolve the caches to use based on runtime arguments. The following example shows how to specify a `CacheResolver`:

```
@Cacheable(cacheResolver="runtimeCacheResolver") ①
public Book findBook(ISBN isbn) {...}
```

① Specifying the `CacheResolver`.

Since Spring 4.1, the `value` attribute of the cache annotations are no longer mandatory, since this particular information can be provided by the `CacheResolver` regardless of the content of the annotation.

Similarly to `key` and `keyGenerator`, the `cacheManager` and `cacheResolver` parameters are mutually exclusive, and an operation specifying both results in an exception, as a custom `CacheManager` is ignored by the `CacheResolver` implementation. This is probably not what you expect.

Synchronized Caching

In a multi-threaded environment, certain operations might be concurrently invoked for the same argument (typically on startup). By default, the cache abstraction does not lock anything, and the same value may be computed several times, defeating the purpose of caching.

For those particular cases, you can use the `sync` attribute to instruct the underlying cache provider to lock the cache entry while the value is being computed. As a result, only one thread is busy computing the value, while the others are blocked until the entry is updated in the cache. The following example shows how to use the `sync` attribute:

```
@Cacheable(cacheNames="foos", sync=true) ①
public Foo executeExpensiveOperation(String id) {...}
```

① Using the `sync` attribute.

This is an optional feature, and your favorite cache library may not support it. All `CacheManager` implementations provided by the core framework support it. See the documentation of your cache provider for more details.

Conditional Caching

Sometimes, a method might not be suitable for caching all the time (for example, it might depend on the given arguments). The cache annotations support such use cases through the `condition` parameter, which takes a `SpEL` expression that is evaluated to either `true` or `false`. If `true`, the method is cached. If not, it behaves as if the method is not cached (that is, the method is invoked every time no matter what values are in the cache or what arguments are used). For example, the following method is cached only if the argument `name` has a length shorter than 32:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32") ①
public Book findBook(String name)
```

① Setting a condition on `@Cacheable`.

In addition to the `condition` parameter, you can use the `unless` parameter to veto the adding of a value to the cache. Unlike `condition`, `unless` expressions are evaluated after the method has been invoked. To expand on the previous example, perhaps we only want to cache paperback books, as the following example does:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32",
unless="#result.hardback") ①
public Book findBook(String name)
```

① Using the `unless` attribute to block hardbacks.

The cache abstraction supports `java.util.Optional` return types. If an `Optional` value is *present*, it will be stored in the associated cache. If an `Optional` value is not present, `null` will be stored in the associated cache. `#result` always refers to the business entity and never a supported wrapper, so the previous example can be rewritten as follows:

```
@Cacheable(cacheNames="book", condition="#name.length() < 32",
unless="#result?.hardback")
public Optional<Book> findBook(String name)
```

Note that `#result` still refers to `Book` and not `Optional<Book>`. Since it might be `null`, we use SpEL's safe navigation operator.

Available Caching SpEL Evaluation Context

Each `SpEL` expression evaluates against a dedicated `context`. In addition to the built-in parameters, the framework provides dedicated caching-related metadata, such as the argument names. The following table describes the items made available to the context so that you can use them for key and conditional computations:

Table 35. Cache SpEL available metadata

Name	Location	Description	Example
`methodName`	Root object	The name of the method being invoked	`#root.methodName`
`method`	Root object	The method being invoked	`#root.method.name`
`target`	Root object	The target object being invoked	`#root.target`
`targetClass`	Root object	The class of the target being invoked	`#root.targetClass`
`args`	Root object	The arguments (as array) used for invoking the target	`#root.args[0]`
`caches`	Root object	Collection of caches against which the current method is run	`#root.caches[0].name`
Argument name	Evaluation context	Name of any of the method arguments. If the names are not available (perhaps due to having no debug information), the argument names are also available under the `#a<#arg>` where `#arg` stands for the argument index (starting from `0`).	`#iban` or `#a0` (you can also use `#p0` or `#p<#arg>` notation as an alias).
`result`	Evaluation context	The result of the method call (the value to be cached). Only available in `unless` expressions, `cache put` expressions (to compute the `key`), or `cache evict` expressions (when `beforeInvocation` is `false`). For supported wrappers (such as `Optional`), `#result` refers to the actual object, not the wrapper.	`#result`

The `@CachePut` Annotation

When the cache needs to be updated without interfering with the method execution, you can use the `@CachePut` annotation. That is, the method is always invoked and its result is placed into the cache (according to the `@CachePut` options). It supports the same options as `@Cacheable` and should be used for cache population rather than method flow optimization. The following example uses the `@CachePut` annotation:

```
@CachePut(cacheNames="book", key="#isbn")
public Book updateBook(ISBN isbn, BookDescriptor descriptor)
```

Using `@CachePut` and `@Cacheable` annotations on the same method is generally strongly discouraged because they have different behaviors. While the latter causes the method invocation to be skipped by using the cache, the former forces the invocation in order to run a cache update. This leads to unexpected behavior and, with the exception of specific corner-cases (such as annotations having conditions that exclude them from each other), such declarations should be avoided. Note also that such conditions should not rely on the result object (that is, the `#result` variable), as these are validated up-front to confirm the exclusion.

The `@CacheEvict` annotation

The cache abstraction allows not just population of a cache store but also eviction. This process is useful for removing stale or unused data from the cache. As opposed to `@Cacheable`, `@CacheEvict` demarcates methods that perform cache eviction (that is, methods that act as triggers for removing data from the cache). Similarly to its sibling, `@CacheEvict` requires specifying one or more caches that are affected by the action, allows a custom cache and key resolution or a condition to be specified, and features an extra parameter (`allEntries`) that indicates whether a cache-wide eviction needs to be performed rather than just an entry eviction (based on the key). The following example evicts all entries from the `books` cache:

```
@CacheEvict(cacheNames="books", allEntries=true) ①
public void loadBooks(InputStream batch)
```

① Using the `allEntries` attribute to evict all entries from the cache.

This option comes in handy when an entire cache region needs to be cleared out. Rather than evicting each entry (which would take a long time, since it is inefficient), all the entries are removed in one operation, as the preceding example shows. Note that the framework ignores any key specified in this scenario as it does not apply (the entire cache is evicted, not only one entry).

You can also indicate whether the eviction should occur after (the default) or before the method is invoked by using the `beforeInvocation` attribute. The former provides the same semantics as the rest of the annotations: Once the method completes successfully, an action (in this case, eviction) on the cache is run. If the method does not run (as it might be cached) or an exception is thrown, the eviction does not occur. The latter (`beforeInvocation=true`) causes the eviction to always occur before the method is invoked. This is useful in cases where the eviction does not need to be tied to

the method outcome.

Note that `void` methods can be used with `@CacheEvict` - as the methods act as a trigger, the return values are ignored (as they do not interact with the cache). This is not the case with `@Cacheable` which adds data to the cache or updates data in the cache and, thus, requires a result.

The `@Caching` Annotation

Sometimes, multiple annotations of the same type (such as `@CacheEvict` or `@CachePut`) need to be specified — for example, because the condition or the key expression is different between different caches. `@Caching` lets multiple nested `@Cacheable`, `@CachePut`, and `@CacheEvict` annotations be used on the same method. The following example uses two `@CacheEvict` annotations:

```
@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary",
key="#p0") })
public Book importBooks(String deposit, Date date)
```

The `@CacheConfig` annotation

So far, we have seen that caching operations offer many customization options and that you can set these options for each operation. However, some of the customization options can be tedious to configure if they apply to all operations of the class. For instance, specifying the name of the cache to use for every cache operation of the class can be replaced by a single class-level definition. This is where `@CacheConfig` comes into play. The following examples uses `@CacheConfig` to set the name of the cache:

```
@CacheConfig("books") ①
public class BookRepositoryImpl implements BookRepository {

    @Cacheable
    public Book findBook(ISBN isbn) {...}
}
```

① Using `@CacheConfig` to set the name of the cache.

`@CacheConfig` is a class-level annotation that allows sharing the cache names, the custom `KeyGenerator`, the custom `CacheManager`, and the custom `CacheResolver`. Placing this annotation on the class does not turn on any caching operation.

An operation-level customization always overrides a customization set on `@CacheConfig`. Therefore, this gives three levels of customizations for each cache operation:

- Globally configured, available for `CacheManager`, `KeyGenerator`.
- At the class level, using `@CacheConfig`.
- At the operation level.

Enabling Caching Annotations

It is important to note that even though declaring the cache annotations does not automatically trigger their actions - like many things in Spring, the feature has to be declaratively enabled (which means if you ever suspect caching is to blame, you can disable it by removing only one configuration line rather than all the annotations in your code).

To enable caching annotations add the annotation `@EnableCaching` to one of your `@Configuration` classes:

```
@Configuration
@EnableCaching
public class AppConfig {
}
```

Alternatively, for XML configuration you can use the `cache:annotation-driven` element:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
https://www.springframework.org/schema/cache/spring-cache.xsd">

        <cache:annotation-driven/>
</beans>
```

Both the `cache:annotation-driven` element and the `@EnableCaching` annotation let you specify various options that influence the way the caching behavior is added to the application through AOP. The configuration is intentionally similar with that of `@Transactional`.

The default advice mode for processing caching annotations is `proxy`, which allows for interception of calls through the proxy only. Local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to `aspectj` mode in combination with compile-time or load-time weaving.

For more detail about advanced customizations (using Java configuration) that are required to implement `CachingConfigurer`, see the javadoc.

Table 36. Cache annotation settings

XML Attribute	Annotation Attribute	Default	Description
`cache-manager`	N/A (see the `CachingConfigurer` javadoc)	`cacheManager`	The name of the cache manager to use. A default `CacheResolver` is initialized behind the scenes with this cache manager (or `cacheManager` if not set). For more fine-grained management of the cache resolution, consider setting the 'cache-resolver' attribute.
`cache-resolver`	N/A (see the `CachingConfigurer` javadoc)	A `SimpleCacheResolver` using the configured `cacheManager`.	The bean name of the CacheResolver that is to be used to resolve the backing caches. This attribute is not required and needs to be specified only as an alternative to the 'cache-manager' attribute.
`key-generator`	N/A (see the `CachingConfigurer` javadoc)	`SimpleKeyGenerator`	Name of the custom key generator to use.
`error-handler`	N/A (see the `CachingConfigurer` javadoc)	`SimpleCacheErrorHandler`	The name of the custom cache error handler to use. By default, any exception thrown during a cache related operation is thrown back at the client.
`mode`	`mode`	`proxy`	The default mode (`proxy`) processes annotated beans to be proxied by using Spring's AOP framework (following proxy semantics, as discussed earlier, applying to method calls coming in through the proxy only). The alternative mode (`aspectj`) instead weaves the affected classes with Spring's AspectJ caching aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires `spring-aspects.jar` in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See Spring configuration for details on how to set up load-time weaving.)
`proxy-target-class`	`proxyTargetClass`	`false`	Applies to proxy mode only. Controls what type of caching proxies are created for classes annotated with the `@Cacheable` or `@CacheEvict` annotations. If the `proxy-target-class` attribute is set to `true`, class-based proxies are created. If `proxy-target-class` is `false` or if the attribute is omitted, standard JDK interface-based proxies are created. (See Proxying Mechanisms for a detailed examination of the different proxy types.)

XML Attribute	Annotation Attribute	Default	Description
`order`	`order`	Ordered.LOWEST_PRECEDENCE	Defines the order of the cache advice that is applied to beans annotated with `@Cacheable` or `@CacheEvict`. (For more information about the rules related to ordering AOP advice, see Advice Ordering.) No specified ordering means that the AOP subsystem determines the order of the advice.

`<cache:annotation-driven/>` looks for `@Cacheable/@CachePut/@CacheEvict/@Caching` only on beans in the same application context in which it is defined. This means that, if you put `<cache:annotation-driven/>` in a `WebApplicationContext` for a `DispatcherServlet`, it checks for beans only in your controllers, not your services. See the MVC section for more information.

Method visibility and cache annotations

When you use proxies, you should apply the cache annotations only to methods with public visibility. If you do annotate protected, private, or package-visible methods with these annotations, no error is raised, but the annotated method does not exhibit the configured caching settings. Consider using AspectJ (see the rest of this section) if you need to annotate non-public methods, as it changes the bytecode itself.

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the `@Cache*` annotations, as opposed to annotating interfaces. You certainly can place an `@Cache*` annotation on an interface (or an interface method), but this works only if you use the proxy mode (`mode="proxy"`). If you use the weaving-based aspect (`mode="aspectj"`), the caching settings are not recognized on interface-level declarations by the weaving infrastructure.

In proxy mode (the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation (in effect, a method within the target object that calls another method of the target object) does not lead to actual caching at runtime even if the invoked method is marked with `@Cacheable`. Consider using the `aspectj` mode in this case. Also, the proxy must be fully initialized to provide the expected behavior, so you should not rely on this feature in your initialization code (that is, `@PostConstruct`).

Using Custom Annotations

Custom annotation and AspectJ

This feature works only with the proxy-based approach but can be enabled with a bit of extra effort by using AspectJ.

The `spring-aspects` module defines an aspect for the standard annotations only. If you have defined your own annotations, you also need to define an aspect for those. Check `AnnotationCacheAspect` for an example.

The caching abstraction lets you use your own annotations to identify what method triggers cache population or eviction. This is quite handy as a template mechanism, as it eliminates the need to duplicate cache annotation declarations, which is especially useful if the key or condition are specified or if the foreign imports (`org.springframework`) are not allowed in your code base. Similarly to the rest of the stereotype annotations, you can use `@Cacheable`, `@CachePut`, `@CacheEvict`, and `@CacheConfig` as meta-annotations (that is, annotations that can annotate other annotations). In the following example, we replace a common `@Cacheable` declaration with our own custom annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Cacheable(cacheNames="books", key="#isbn")
public @interface SlowService {
}
```

In the preceding example, we have defined our own `SlowService` annotation, which itself is annotated with `@Cacheable`. Now we can replace the following code:

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

The following example shows the custom annotation with which we can replace the preceding code:

```
@SlowService
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

Even though `@SlowService` is not a Spring annotation, the container automatically picks up its declaration at runtime and understands its meaning. Note that, as mentioned earlier, annotation-driven behavior needs to be enabled.

7.6.3. JCache (JSR-107) Annotations

Since version 4.1, Spring's caching abstraction fully supports the JCache standard (JSR-107) annotations: `@CacheResult`, `@CachePut`, `@CacheRemove`, and `@CacheRemoveAll` as well as the `@CacheDefaults`, `@CacheKey`, and `@CacheValue` companions. You can use these annotations even

without migrating your cache store to JSR-107. The internal implementation uses Spring's caching abstraction and provides default `CacheResolver` and `KeyGenerator` implementations that are compliant with the specification. In other words, if you are already using Spring's caching abstraction, you can switch to these standard annotations without changing your cache storage (or configuration, for that matter).

Feature Summary

For those who are familiar with Spring's caching annotations, the following table describes the main differences between the Spring annotations and their JSR-107 counterparts:

Table 37. Spring vs. JSR-107 caching annotations

Spring	JSR-107	Remark
`@Cacheable`	`@CacheResult`	Fairly similar. `@CacheResult` can cache specific exceptions and force the execution of the method regardless of the content of the cache.
`@CachePut`	`@CachePut`	While Spring updates the cache with the result of the method invocation, JCache requires that it be passed it as an argument that is annotated with `@CacheValue`. Due to this difference, JCache allows updating the cache before or after the actual method invocation.
`@CacheEvict`	`@CacheRemove`	Fairly similar. `@CacheRemove` supports conditional eviction when the method invocation results in an exception.
`@CacheEvict(allEntries=true)`	`@CacheRemoveAll`	See `@CacheRemove`.
`@CacheConfig`	`@CacheDefaults`	Lets you configure the same concepts, in a similar fashion.

JCache has the notion of `javax.cache.annotation.CacheResolver`, which is identical to the Spring's `CacheResolver` interface, except that JCache supports only a single cache. By default, a simple implementation retrieves the cache to use based on the name declared on the annotation. It should be noted that, if no cache name is specified on the annotation, a default is automatically generated. See the javadoc of `@CacheResult#cacheName()` for more information.

`CacheResolver` instances are retrieved by a `CacheResolverFactory`. It is possible to customize the factory for each cache operation, as the following example shows:

```
@CacheResult(cacheNames="books", cacheResolverFactory=MyCacheResolverFactory.class) ①
public Book findBook(ISBN isbn)
```

① Customizing the factory for this operation.

For all referenced classes, Spring tries to locate a bean with the given type. If more than one match exists, a new instance is created and can use the regular bean lifecycle callbacks, such as dependency injection.

Keys are generated by a `javax.cache.annotation.CacheKeyGenerator` that serves the same purpose as

Spring's `KeyGenerator`. By default, all method arguments are taken into account, unless at least one parameter is annotated with `@CacheKey`. This is similar to Spring's custom key generation declaration. For instance, the following are identical operations, one using Spring's abstraction and the other using JCache:

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@CacheResult(cacheName="books")
public Book findBook(@CacheKey ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

You can also specify the `CacheKeyResolver` on the operation, similar to how you can specify the `CacheResolverFactory`.

JCache can manage exceptions thrown by annotated methods. This can prevent an update of the cache, but it can also cache the exception as an indicator of the failure instead of calling the method again. Assume that `InvalidIsbnNotFoundException` is thrown if the structure of the ISBN is invalid. This is a permanent failure (no book could ever be retrieved with such a parameter). The following caches the exception so that further calls with the same, invalid, ISBN throw the cached exception directly instead of invoking the method again:

```
@CacheResult(cacheName="books", exceptionCacheName="failures"
             cachedExceptions = InvalidIsbnNotFoundException.class)
public Book findBook(ISBN isbn)
```

Enabling JSR-107 Support

You do not need to do anything specific to enable the JSR-107 support alongside Spring's declarative annotation support. Both `@EnableCaching` and the `cache:annotation-driven` XML element automatically enable the JCache support if both the JSR-107 API and the `spring-context-support` module are present in the classpath.

Depending on your use case, the choice is basically yours. You can even mix and match services by using the JSR-107 API on some and using Spring's own annotations on others. However, if these services impact the same caches, you should use a consistent and identical key generation implementation.

7.6.4. Declarative XML-based Caching

If annotations are not an option (perhaps due to having no access to the sources or no external code), you can use XML for declarative caching. So, instead of annotating the methods for caching, you can specify the target method and the caching directives externally (similar to the declarative transaction management advice). The example from the previous section can be translated into the following example:

```
<!-- the service we want to make cacheable -->
<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->
<cache:advice id="cacheAdvice" cache-manager="cacheManager">
    <cache:caching cache="books">
        <cache:cacheable method="findBook" key="#isbn"/>
        <cache:cache-evict method="loadBooks" all-entries="true"/>
    </cache:caching>
</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces -->
<aop:config>
    <aop:advisor advice-ref="cacheAdvice" pointcut="execution(*
x.y.BookService.*(..))"/>
</aop:config>

<!-- cache manager definition omitted -->
```

In the preceding configuration, the `bookService` is made cacheable. The caching semantics to apply are encapsulated in the `cache:advice` definition, which causes the `findBooks` method to be used for putting data into the cache and the `loadBooks` method for evicting data. Both definitions work against the `books` cache.

The `aop:config` definition applies the cache advice to the appropriate points in the program by using the AspectJ pointcut expression (more information is available in Aspect Oriented Programming with Spring). In the preceding example, all methods from the `BookService` are considered and the cache advice is applied to them.

The declarative XML caching supports all of the annotation-based model, so moving between the two should be fairly easy. Furthermore, both can be used inside the same application. The XML-based approach does not touch the target code. However, it is inherently more verbose. When dealing with classes that have overloaded methods that are targeted for caching, identifying the proper methods does take an extra effort, since the `method` argument is not a good discriminator. In these cases, you can use the AspectJ pointcut to cherry pick the target methods and apply the appropriate caching functionality. However, through XML, it is easier to apply package or group or interface-wide caching (again, due to the AspectJ pointcut) and to create template-like definitions (as we did in the preceding example by defining the target cache through the `cache:definitions cache` attribute).

7.6.5. Configuring the Cache Storage

The cache abstraction provides several storage integration options. To use them, you need to declare an appropriate `CacheManager` (an entity that controls and manages `Cache` instances and that can be used to retrieve these for storage).

JDK `ConcurrentMap`-based Cache

The JDK-based `Cache` implementation resides under `org.springframework.cache.concurrent` package. It lets you use `ConcurrentHashMap` as a backing `Cache` store. The following example shows how to configure two caches:

```
<!-- simple cache manager -->
<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
        <set>
            <bean
class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean"
p:name="default"/>
            <bean
class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean"
p:name="books"/>
        </set>
    </property>
</bean>
```

The preceding snippet uses the `SimpleCacheManager` to create a `CacheManager` for the two nested `ConcurrentMapCache` instances named `default` and `books`. Note that the names are configured directly for each cache.

As the cache is created by the application, it is bound to its lifecycle, making it suitable for basic use cases, tests, or simple applications. The cache scales well and is very fast, but it does not provide any management, persistence capabilities, or eviction contracts.

Ehcache-based Cache

Ehcache 3.x is fully JSR-107 compliant and no dedicated support is required for it. See JSR-107 Cache for details.

Caffeine Cache

Caffeine is a Java 8 rewrite of Guava's cache, and its implementation is located in the `org.springframework.cache.caffeine` package and provides access to several features of Caffeine.

The following example configures a `CacheManager` that creates the cache on demand:

```
<bean id="cacheManager"
        class="org.springframework.cache.caffeine.CaffeineCacheManager"/>
```

You can also provide the caches to use explicitly. In that case, only those are made available by the manager. The following example shows how to do so:

```
<bean id="cacheManager"
class="org.springframework.cache.caffeine.CaffeineCacheManager">
    <property name="cacheNames">
        <set>
            <value>default</value>
            <value>books</value>
        </set>
    </property>
</bean>
```

The Caffeine `CacheManager` also supports custom `Caffeine` and `CacheLoader`. See the Caffeine documentation for more information about those.

GemFire-based Cache

GemFire is a memory-oriented, disk-backed, elastically scalable, continuously available, active (with built-in pattern-based subscription notifications), globally replicated database and provides fully-featured edge caching. For further information on how to use GemFire as a `CacheManager` (and more), see the Spring Data GemFire reference documentation.

JSR-107 Cache

Spring's caching abstraction can also use JSR-107-compliant caches. The JCache implementation is located in the `org.springframework.cache.jcache` package.

Again, to use it, you need to declare the appropriate `CacheManager`. The following example shows how to do so:

```
<bean id="cacheManager"
        class="org.springframework.cache.jcache.JCacheCacheManager"
        p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup  -->
<bean id="jCacheManager" .../>
```

Dealing with Caches without a Backing Store

Sometimes, when switching environments or doing testing, you might have cache declarations without having an actual backing cache configured. As this is an invalid configuration, an exception is thrown at runtime, since the caching infrastructure is unable to find a suitable store. In situations like this, rather than removing the cache declarations (which can prove tedious), you can wire in a simple dummy cache that performs no caching — that is, it forces the cached methods to be invoked every time. The following example shows how to do so:

```
<bean id="cacheManager"
class="org.springframework.cache.support.CompositeCacheManager">
    <property name="cacheManagers">
        <list>
            <ref bean="jdkCache"/>
            <ref bean="gemfireCache"/>
        </list>
    </property>
    <property name="fallbackToNoOpCache" value="true"/>
</bean>
```

The `CompositeCacheManager` in the preceding chains multiple `CacheManager` instances and, through the `fallbackToNoOpCache` flag, adds a no-op cache for all the definitions not handled by the configured cache managers. That is, every cache definition not found in either `jdkCache` or `gemfireCache` (configured earlier in the example) is handled by the no-op cache, which does not store any information, causing the target method to be invoked every time.

7.6.6. Plugging-in Different Back-end Caches

Clearly, there are plenty of caching products out there that you can use as a backing store. For those that do not support JSR-107 you need to provide a `CacheManager` and a `Cache` implementation. This may sound harder than it is, since, in practice, the classes tend to be simple adapters that map the caching abstraction framework on top of the storage API, as the *Caffeine* classes do. Most `CacheManager` classes can use the classes in the `org.springframework.cache.support` package (such as `AbstractCacheManager` which takes care of the boiler-plate code, leaving only the actual mapping to be completed).

7.6.7. How can I Set the TTL/TTI/Eviction policy/XXX feature?

Directly through your cache provider. The cache abstraction is an abstraction, not a cache implementation. The solution you use might support various data policies and different topologies that other solutions do not support (for example, the JDK `ConcurrentHashMap` — exposing that in the cache abstraction would be useless because there would no backing support). Such functionality should be controlled directly through the backing cache (when configuring it) or through its native API.

7.7. Appendix

7.7.1. XML Schemas

This part of the appendix lists XML schemas related to integration technologies.

The `jee` Schema

The `jee` elements deal with issues related to Jakarta EE (Enterprise Edition) configuration, such as looking up a JNDI object and defining EJB references.

To use the elements in the `jee` schema, you need to have the following preamble at the top of your

Spring XML configuration file. The text in the following snippet references the correct schema so that the elements in the `jee` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jee
https://www.springframework.org/schema/jee/spring-jee.xsd">

    <!-- bean definitions here -->

</beans>
```

<jee:jndi-lookup/> (simple)

The following example shows how to use JNDI to look up a data source without the `jee` schema:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
</bean>
<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

The following example shows how to use JNDI to look up a data source with the `jee` schema:

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">
    <!-- Spring will do the cast automatically (as usual) -->
    <property name="dataSource" ref="dataSource"/>
</bean>
```

`<jee:jndi-lookup/>` (with Single JNDI Environment Setting)

The following example shows how to use JNDI to look up an environment variable without `jee`:

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

The following example shows how to use JNDI to look up an environment variable with `jee`:

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>ping=pong</jee:environment>
</jee:jndi-lookup>
```

`<jee:jndi-lookup/>` (with Multiple JNDI Environment Settings)

The following example shows how to use JNDI to look up multiple environment variables without `jee`:

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="sing">song</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

The following example shows how to use JNDI to look up multiple environment variables with `jee`:

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties
format) -->
    <jee:environment>
        sing=song
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

`<jee:jndi-lookup/>` (Complex)

The following example shows how to use JNDI to look up a data source and a number of different properties without `jee`:

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultThing"/>
    <property name="proxyInterface" value="com.myapp.Thing"/>
</bean>
```

The following example shows how to use JNDI to look up a data source and a number of different properties with `jee`:

```
<jee:jndi-lookup id="simple"
        jndi-name="jdbc/MyDataSource"
        cache="true"
        resource-ref="true"
        lookup-on-startup="false"
        expected-type="com.myapp.DefaultThing"
        proxy-interface="com.myapp.Thing"/>
```

`<jee:local-slsb/>` (Simple)

The `<jee:local-slsb/>` element configures a reference to a local EJB Stateless Session Bean.

The following example shows how to configures a reference to a local EJB Stateless Session Bean without `jee`:

```
<bean id="simple"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

The following example shows how to configures a reference to a local EJB Stateless Session Bean with `jee`:

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"/>
```

`<jee:local-slsb/>` (Complex)

The `<jee:local-slsb/>` element configures a reference to a local EJB Stateless Session Bean.

The following example shows how to configures a reference to a local EJB Stateless Session Bean and a number of properties without `jee`:

```
<bean id="complexLocalEjb"
        class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/RentalServiceBean"/>
    <property name="businessInterface" value="com.example.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
</bean>
```

The following example shows how to configures a reference to a local EJB Stateless Session Bean and a number of properties with `jee`:

```
<jee:local-slsb id="complexLocalEjb"
        jndi-name="ejb/RentalServiceBean"
        business-interface="com.foo.service.RentalService"
        cache-home="true"
        lookup-home-on-startup="true"
        resource-ref="true">
```

<jee:remote-slsb/>

The `<jee:remote-slsb/>` element configures a reference to a `remote` EJB Stateless Session Bean.

The following example shows how to configures a reference to a remote EJB Stateless Session Bean without `jee`:

```
<bean id="complexRemoteEjb"

class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
    <property name="jndiName" value="ejb/MyRemoteBean"/>
    <property name="businessInterface" value="com.foo.service.RentalService"/>
    <property name="cacheHome" value="true"/>
    <property name="lookupHomeOnStartup" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="homeInterface" value="com.foo.service.RentalService"/>
    <property name="refreshHomeOnConnectFailure" value="true"/>
</bean>
```

The following example shows how to configures a reference to a remote EJB Stateless Session Bean with `jee`:

```
<jee:remote-slsb id="complexRemoteEjb"
        jndi-name="ejb/MyRemoteBean"
        business-interface="com.foo.service.RentalService"
        cache-home="true"
        lookup-home-on-startup="true"
        resource-ref="true"
        home-interface="com.foo.service.RentalService"
        refresh-home-on-connect-failure="true">
```

The `jms` Schema

The `jms` elements deal with configuring JMS-related beans, such as Spring's Message Listener Containers. These elements are detailed in the section of the JMS chapter entitled JMS Namespace Support. See that chapter for full details on this support and the `jms` elements themselves.

In the interest of completeness, to use the elements in the `jms` schema, you need to have the following preamble at the top of your Spring XML configuration file. The text in the following snippet references the correct schema so that the elements in the `jms` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/jms
https://www.springframework.org/schema/jms/spring-jms.xsd">

    <!-- bean definitions here -->

</beans>
```

Using `<context:mbean-export/>`

This element is detailed in Configuring Annotation-based MBean Export.

The `cache` Schema

You can use the `cache` elements to enable support for Spring's `@CacheEvict`, `@CachePut`, and `@Caching` annotations. It it also supports declarative XML-based caching. See Enabling Caching Annotations and Declarative XML-based Caching for details.

To use the elements in the `cache` schema, you need to have the following preamble at the top of your Spring XML configuration file. The text in the following snippet references the correct schema so that the elements in the `cache` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
https://www.springframework.org/schema/cache/spring-cache.xsd">

    <!-- bean definitions here -->

</beans>
```

Chapter 8. Language Support

8.1. Kotlin

Kotlin is a statically typed language that targets the JVM (and other platforms) which allows writing concise and elegant code while providing very good interoperability with existing libraries written in Java.

The Spring Framework provides first-class support for Kotlin and lets developers write Kotlin applications almost as if the Spring Framework was a native Kotlin framework. Most of the code samples of the reference documentation are provided in Kotlin in addition to Java.

The easiest way to build a Spring application with Kotlin is to leverage Spring Boot and its dedicated Kotlin support. This comprehensive tutorial will teach you how to build Spring Boot applications with Kotlin using start.spring.io.

Feel free to join the #spring channel of Kotlin Slack or ask a question with `spring` and `kotlin` as tags on Stackoverflow if you need support.

8.1.1. Requirements

Spring Framework supports Kotlin 1.3+ and requires `kotlin-stdlib` (or one of its variants, such as `kotlin-stdlib-jdk8`) and `kotlin-reflect` to be present on the classpath. They are provided by default if you bootstrap a Kotlin project on start.spring.io.

Kotlin inline classes are not yet supported.

The Jackson Kotlin module is required for serializing or deserializing JSON data for Kotlin classes with Jackson, so make sure to add the `com.fasterxml.jackson.module:jackson-module-kotlin` dependency to your project if you have such need. It is automatically registered when found in the classpath.

8.1.2. Extensions

Kotlin extensions provide the ability to extend existing classes with additional functionality. The Spring Framework Kotlin APIs use these extensions to add new Kotlin-specific conveniences to existing Spring APIs.

The Spring Framework KDoc API lists and documents all available Kotlin extensions and DSLs.

Keep in mind that Kotlin extensions need to be imported to be used. This means, for example, that the `GenericApplicationContext.registerBean` Kotlin extension is available only if `org.springframework.context.support.registerBean` is imported. That said, similar to static imports, an IDE should automatically suggest the import in most cases.

For example, Kotlin reified type parameters provide a workaround for JVM generics type erasure,

and the Spring Framework provides some extensions to take advantage of this feature. This allows for a better Kotlin API `RestTemplate`, for the new `WebClient` from Spring WebFlux, and for various other APIs.

Other libraries, such as Reactor and Spring Data, also provide Kotlin extensions for their APIs, thus giving a better Kotlin development experience overall.

To retrieve a list of `User` objects in Java, you would normally write the following:

```
Flux<User> users  = client.get().retrieve().bodyToFlux(User.class)
```

With Kotlin and the Spring Framework extensions, you can instead write the following:

```
val users = client.get().retrieve().bodyToFlux<User>()
// or (both are equivalent)
val users : Flux<User> = client.get().retrieve().bodyToFlux()
```

As in Java, `users` in Kotlin is strongly typed, but Kotlin's clever type inference allows for shorter syntax.

8.1.3. Null-safety

One of Kotlin's key features is null-safety, which cleanly deals with `null` values at compile time rather than bumping into the famous `NullPointerException` at runtime. This makes applications safer through nullability declarations and expressing "value or no value" semantics without paying the cost of wrappers, such as `Optional`. (Kotlin allows using functional constructs with nullable values. See this comprehensive guide to Kotlin null-safety.)

Although Java does not let you express null-safety in its type-system, the Spring Framework provides null-safety of the whole Spring Framework API via tooling-friendly annotations declared in the `org.springframework.lang` package. By default, types from Java APIs used in Kotlin are recognized as platform types, for which null-checks are relaxed. Kotlin support for JSR-305 annotations and Spring nullability annotations provide null-safety for the whole Spring Framework API to Kotlin developers, with the advantage of dealing with `null`-related issues at compile time.

Libraries such as Reactor or Spring Data provide null-safe APIs to leverage this feature.

You can configure JSR-305 checks by adding the `-Xjsr305` compiler flag with the following options: `-Xjsr305={strict|warn|ignore}`.

For kotlin versions 1.1+, the default behavior is the same as `-Xjsr305=warn`. The `strict` value is required to have Spring Framework API null-safety taken into account in Kotlin types inferred from Spring API but should be used with the knowledge that Spring API nullability declaration could evolve even between minor releases and that more checks may be added in the future.

Generic type arguments, varargs, and array elements nullability are not supported yet, but should be in an upcoming release. See this discussion for up-to-date information.

8.1.4. Classes and Interfaces

The Spring Framework supports various Kotlin constructs, such as instantiating Kotlin classes through primary constructors, immutable classes data binding, and function optional parameters with default values.

Kotlin parameter names are recognized through a dedicated `KotlinReflectionParameterNameDiscoverer`, which allows finding interface method parameter names without requiring the Java 8 `-parameters` compiler flag to be enabled during compilation.

You can declare configuration classes as top level or nested but not inner, since the later requires a reference to the outer class.

8.1.5. Annotations

The Spring Framework also takes advantage of Kotlin null-safety to determine if an HTTP parameter is required without having to explicitly define the `required` attribute. That means `@RequestParam name: String?` is treated as not required and, conversely, `@RequestParam name: String` is treated as being required. This feature is also supported on the Spring Messaging `@Header` annotation.

In a similar fashion, Spring bean injection with `@Autowired`, `@Bean`, or `@Inject` uses this information to determine if a bean is required or not.

For example, `@Autowired lateinit var thing: Thing` implies that a bean of type `Thing` must be registered in the application context, while `@Autowired lateinit var thing: Thing?` does not raise an error if such a bean does not exist.

Following the same principle, `@Bean fun play(toy: Toy, car: Car?) = Baz(toy, Car)` implies that a bean of type `Toy` must be registered in the application context, while a bean of type `Car` may or may not exist. The same behavior applies to autowired constructor parameters.

If you use bean validation on classes with properties or a primary constructor parameters, you may need to use annotation use-site targets, such as `@field:NotNull` or `@get:Size(min=5, max=15)`, as described in this Stack Overflow response.

8.1.6. Bean Definition DSL

Spring Framework supports registering beans in a functional way by using lambdas as an alternative to XML or Java configuration (`@Configuration` and `@Bean`). In a nutshell, it lets you register beans with a lambda that acts as a `FactoryBean`. This mechanism is very efficient, as it does not require any reflection or CGLIB proxies.

In Java, you can, for example, write the following:

```
class Foo {}

class Bar {
    private final Foo foo;
    public Bar(Foo foo) {
        this.foo = foo;
    }
}

GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(Foo.class);
context.registerBean(Bar.class, () -> new Bar(context.getBean(Foo.class)));
```

In Kotlin, with reified type parameters and `GenericApplicationContext` Kotlin extensions, you can instead write the following:

```
class Foo

class Bar(private val foo: Foo)

val context = GenericApplicationContext().apply {
    registerBean<Foo>()
    registerBean { Bar(it.getBean()) }
}
```

When the class `Bar` has a single constructor, you can even just specify the bean class, the constructor parameters will be autowired by type:

```
val context = GenericApplicationContext().apply {
    registerBean<Foo>()
    registerBean<Bar>()
}
```

In order to allow a more declarative approach and cleaner syntax, Spring Framework provides a Kotlin bean definition DSL It declares an `ApplicationContextInitializer` through a clean declarative API, which lets you deal with profiles and `Environment` for customizing how beans are registered.

In the following example notice that:

- Type inference usually allows to avoid specifying the type for bean references like `ref("bazBean")`
- It is possible to use Kotlin top level functions to declare beans using callable references like `bean(::myRouter)` in this example
- When specifying `bean<Bar>()` or `bean(::myRouter)`, parameters are autowired by type

- The `FooBar` bean will be registered only if the `foobar` profile is active

```
class Foo
class Bar(private val foo: Foo)
class Baz(var message: String = "")
class FooBar(private val baz: Baz)

val myBeans = beans {
    bean<Foo>()
    bean<Bar>()
    bean("bazBean") {
        Baz().apply {
            message = "Hello world"
        }
    }
    profile("foobar") {
        bean { FooBar(ref("bazBean")) }
    }
    bean(::myRouter)
}

fun myRouter(foo: Foo, bar: Bar, baz: Baz) = router {
    // ...
}
```

This DSL is programmatic, meaning it allows custom registration logic of beans through an `if` expression, a `for` loop, or any other Kotlin constructs.

You can then use this `beans()` function to register beans on the application context, as the following example shows:

```
val context = GenericApplicationContext().apply {
    myBeans.initialize(this)
    refresh()
}
```

Spring Boot is based on JavaConfig and does not yet provide specific support for functional bean definition, but you can experimentally use functional bean definitions through Spring Boot's `ApplicationContextInitializer` support. See this Stack Overflow answer for more details and up-to-date information. See also the experimental Kofu DSL developed in Spring Fu incubator.

8.1.7. Web

Router DSL

Spring Framework comes with a Kotlin router DSL available in 3 flavors:

- WebMvc.fn DSL with router { }
- WebFlux.fn Reactive DSL with router { }
- WebFlux.fn Coroutines DSL with coRouter { }

These DSL let you write clean and idiomatic Kotlin code to build a `RouterFunction` instance as the following example shows:

```
@Configuration
class RouterRouterConfiguration {

    @Bean
    fun mainRouter(userHandler: UserHandler) = router {
        accept(TEXT_HTML).nest {
            GET("/") { ok().render("index") }
            GET("/sse") { ok().render("sse") }
            GET("/users", userHandler::findAllView)
        }
        "/api".nest {
            accept(APPLICATION_JSON).nest {
                GET("/users", userHandler::findAll)
            }
            accept(TEXT_EVENT_STREAM).nest {
                GET("/users", userHandler::stream)
            }
        }
        resources("/**", ClassPathResource("static/"))
    }
}
```

This DSL is programmatic, meaning that it allows custom registration logic of beans through an `if` expression, a `for` loop, or any other Kotlin constructs. That can be useful when you need to register routes depending on dynamic data (for example, from a database).

See MiXiT project for a concrete example.

MockMvc DSL

A Kotlin DSL is provided via `MockMvc` Kotlin extensions in order to provide a more idiomatic Kotlin API and to allow better discoverability (no usage of static methods).

```
val mockMvc: MockMvc = ...
mockMvc.get("/person/{name}", "Lee") {
    secure = true
    accept = APPLICATION_JSON
    headers {
        contentLanguage = Locale.FRANCE
    }
    principal = Principal { "foo" }
}.andExpect {
    status { isOk }
    content { contentType(APPLICATION_JSON) }
    jsonPath("$.name") { value("Lee") }
    content { json("""{"someBoolean": false}""", false) }
}.andDo {
    print()
}
```

Kotlin Script Templates

Spring Framework provides a `ScriptTemplateView` which supports JSR-223 to render templates by using script engines.

By leveraging `scripting-jsr223` dependencies, it is possible to use such feature to render Kotlin-based templates with kotlinx.html DSL or Kotlin multiline interpolated `String`.

`build.gradle.kts`

```
dependencies {
        runtime("org.jetbrains.kotlin:kotlin-scripting-jsr223:${kotlinVersion}")
}
```

Configuration is usually done with `ScriptTemplateConfigurer` and `ScriptTemplateViewResolver` beans.

`KotlinScriptConfiguration.kt`

```
@Configuration
class KotlinScriptConfiguration {

    @Bean
    fun kotlinScriptConfigurer() = ScriptTemplateConfigurer().apply {
        engineName = "kotlin"
        setScripts("scripts/render.kts")
        renderFunction = "render"
        isSharedEngine = false
    }

    @Bean
    fun kotlinScriptViewResolver() = ScriptTemplateViewResolver().apply {
        setPrefix("templates/")
        setSuffix(".kts")
    }
}
```

See the kotlin-script-templating example project for more details.

Kotlin multiplatform serialization

As of Spring Framework 5.3, Kotlin multiplatform serialization is supported in Spring MVC, Spring WebFlux and Spring Messaging (RSocket). The builtin support currently targets CBOR, JSON, and ProtoBuf formats.

To enable it, follow those instructions to add the related dependency and plugin. With Spring MVC and WebFlux, both Kotlin serialization and Jackson will be configured by default if they are in the classpath since Kotlin serialization is designed to serialize only Kotlin classes annotated with `@Serializable`. With Spring Messaging (RSocket), make sure that neither Jackson, GSON or JSONB are in the classpath if you want automatic configuration, if Jackson is needed configure `KotlinSerializationJsonMessageConverter` manually.

8.1.8. Coroutines

Kotlin Coroutines are Kotlin lightweight threads allowing to write non-blocking code in an imperative way. On language side, suspending functions provides an abstraction for asynchronous operations while on library side kotlinx.coroutines provides functions like `async { }` and types like `Flow`.

Spring Framework provides support for Coroutines on the following scope:

- Deferred and Flow return values support in Spring MVC and WebFlux annotated `@Controller`
- Suspending function support in Spring MVC and WebFlux annotated `@Controller`
- Extensions for WebFlux client and server functional API.
- WebFlux.fn coRouter { } DSL
- Suspending function and `Flow` support in RSocket `@MessageMapping` annotated methods

- Extensions for `RSocketRequester`

Dependencies

Coroutines support is enabled when `kotlinx-coroutines-core` and `kotlinx-coroutines-reactor` dependencies are in the classpath:

`build.gradle.kts`

```
dependencies {

    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:${coroutinesVersion}")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
reactor:${coroutinesVersion}")
}
```

Version `1.4.0` and above are supported.

How Reactive translates to Coroutines?

For return values, the translation from Reactive to Coroutines APIs is the following:

- `fun handler(): Mono<Void>` becomes `suspend fun handler()`
- `fun handler(): Mono<T>` becomes `suspend fun handler(): T` or `suspend fun handler(): T?` depending on if the `Mono` can be empty or not (with the advantage of being more statically typed)
- `fun handler(): Flux<T>` becomes `fun handler(): Flow<T>`

For input parameters:

- If laziness is not needed, `fun handler(mono: Mono<T>)` becomes `fun handler(value: T)` since a suspending functions can be invoked to get the value parameter.
- If laziness is needed, `fun handler(mono: Mono<T>)` becomes `fun handler(supplier: suspend () → T)` or `fun handler(supplier: suspend () → T?)`

`Flow` is `Flux` equivalent in Coroutines world, suitable for hot or cold stream, finite or infinite streams, with the following main differences:

- `Flow` is push-based while `Flux` is push-pull hybrid
- Backpressure is implemented via suspending functions
- `Flow` has only a single suspending `collect` method and operators are implemented as extensions
- Operators are easy to implement thanks to Coroutines
- Extensions allow to add custom operators to `Flow`
- Collect operations are suspending functions
- `map` operator supports asynchronous operation (no need for `flatMap`) since it takes a suspending function parameter

Read this blog post about Going Reactive with Spring, Coroutines and Kotlin Flow for more details, including how to run code concurrently with Coroutines.

Controllers

Here is an example of a Coroutines `@RestController`.

```
@RestController
class CoroutinesRestController(client: WebClient, banner: Banner) {

    @GetMapping("/suspend")
    suspend fun suspendingEndpoint(): Banner {
        delay(10)
        return banner
    }

    @GetMapping("/flow")
    fun flowEndpoint() = flow {
        delay(10)
        emit(banner)
        delay(10)
        emit(banner)
    }

    @GetMapping("/deferred")
    fun deferredEndpoint() = GlobalScope.async {
        delay(10)
        banner
    }

    @GetMapping("/sequential")
    suspend fun sequential(): List<Banner> {
        val banner1 = client
                .get()
                .uri("/suspend")
                .accept(MediaType.APPLICATION_JSON)
                .awaitExchange()
                .awaitBody<Banner>()
        val banner2 = client
                .get()
                .uri("/suspend")
                .accept(MediaType.APPLICATION_JSON)
                .awaitExchange()
                .awaitBody<Banner>()
        return listOf(banner1, banner2)
    }

    @GetMapping("/parallel")
    suspend fun parallel(): List<Banner> = coroutineScope {
        val deferredBanner1: Deferred<Banner> = async {
```

```
            client
                    .get()
                    .uri("/suspend")
                    .accept(MediaType.APPLICATION_JSON)
                    .awaitExchange()
                    .awaitBody<Banner>()
        }
        val deferredBanner2: Deferred<Banner> = async {
            client
                    .get()
                    .uri("/suspend")
                    .accept(MediaType.APPLICATION_JSON)
                    .awaitExchange()
                    .awaitBody<Banner>()
        }
        listOf(deferredBanner1.await(), deferredBanner2.await())
    }

    @GetMapping("/error")
    suspend fun error() {
        throw IllegalStateException()
    }

    @GetMapping("/cancel")
    suspend fun cancel() {
        throw CancellationException()
    }

}
```

View rendering with a `@Controller` is also supported.

```
@Controller
class CoroutinesViewController(banner: Banner) {

    @GetMapping("/")
    suspend fun render(model: Model): String {
        delay(10)
        model["banner"] = banner
        return "index"
    }
}
```

WebFlux.fn

Here is an example of Coroutines router defined via the coRouter { } DSL and related handlers.

```
@Configuration
class RouterConfiguration {

    @Bean
    fun mainRouter(userHandler: UserHandler) = coRouter {
        GET("/", userHandler::listView)
        GET("/api/user", userHandler::listApi)
    }
}
```

```
class UserHandler(builder: WebClient.Builder) {

    private val client = builder.baseUrl("...").build()

    suspend fun listView(request: ServerRequest): ServerResponse =
            ServerResponse.ok().renderAndAwait("users", mapOf("users" to
            client.get().uri("...").awaitExchange().awaitBody<User>()))

    suspend fun listApi(request: ServerRequest): ServerResponse =

ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyAndAwait(
                client.get().uri("...").awaitExchange().awaitBody<User>())
}
```

Transactions

Transactions on Coroutines are supported via the programmatic variant of the Reactive transaction management provided as of Spring Framework 5.2.

For suspending functions, a `TransactionalOperator.executeAndAwait` extension is provided.

```
import org.springframework.transaction.reactive.executeAndAwait

class PersonRepository(private val operator: TransactionalOperator) {

    suspend fun initDatabase() = operator.executeAndAwait {
        insertPerson1()
        insertPerson2()
    }

    private suspend fun insertPerson1() {
        // INSERT SQL statement
    }

    private suspend fun insertPerson2() {
        // INSERT SQL statement
    }
}
```

For Kotlin `Flow`, a `Flow<T>.transactional` extension is provided.

```
import org.springframework.transaction.reactive.transactional

class PersonRepository(private val operator: TransactionalOperator) {

    fun updatePeople() = findPeople().map(::updatePerson).transactional(operator)

    private fun findPeople(): Flow<Person> {
        // SELECT SQL statement
    }

    private suspend fun updatePerson(person: Person): Person {
        // UPDATE SQL statement
    }
}
```

8.1.9. Spring Projects in Kotlin

This section provides some specific hints and recommendations worth for developing Spring projects in Kotlin.

Final by Default

By default, all classes in Kotlin are `final`. The `open` modifier on a class is the opposite of Java's `final`: It allows others to inherit from this class. This also applies to member functions, in that they need to be marked as `open` to be overridden.

While Kotlin's JVM-friendly design is generally frictionless with Spring, this specific Kotlin feature can prevent the application from starting, if this fact is not taken into consideration. This is because

Spring beans (such as `@Configuration` annotated classes which by default need to be extended at runtime for technical reasons) are normally proxied by CGLIB. The workaround is to add an `open` keyword on each class and member function of Spring beans that are proxied by CGLIB, which can quickly become painful and is against the Kotlin principle of keeping code concise and predictable.

It is also possible to avoid CGLIB proxies for configuration classes by using `@Configuration(proxyBeanMethods = false)`. See `proxyBeanMethods` Javadoc for more details.

Fortunately, Kotlin provides a `kotlin-spring` plugin (a preconfigured version of the `kotlin-allopen` plugin) that automatically opens classes and their member functions for types that are annotated or meta-annotated with one of the following annotations:

- `@Component`
- `@Async`
- `@Transactional`
- `@Cacheable`

Meta-annotation support means that types annotated with `@Configuration`, `@Controller`, `@RestController`, `@Service`, or `@Repository` are automatically opened since these annotations are meta-annotated with `@Component`.

start.spring.io enables the `kotlin-spring` plugin by default. So, in practice, you can write your Kotlin beans without any additional `open` keyword, as in Java.

The Kotlin code samples in Spring Framework documentation do not explicitly specify `open` on the classes and their member functions. The samples are written for projects using the `kotlin-allopen` plugin, since this is the most commonly used setup.

Using Immutable Class Instances for Persistence

In Kotlin, it is convenient and considered to be a best practice to declare read-only properties within the primary constructor, as in the following example:

```
class Person(val name: String, val age: Int)
```

You can optionally add the `data` keyword to make the compiler automatically derive the following members from all properties declared in the primary constructor:

- `equals()` and `hashCode()`
- `toString()` of the form `"User(name=John, age=42)"`
- `componentN()` functions that correspond to the properties in their order of declaration
- `copy()` function

As the following example shows, this allows for easy changes to individual properties, even if

`Person` properties are read-only:

```
data class Person(val name: String, val age: Int)

val jack = Person(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

Common persistence technologies (such as JPA) require a default constructor, preventing this kind of design. Fortunately, there is a workaround for this "default constructor hell", since Kotlin provides a `kotlin-jpa` plugin that generates synthetic no-arg constructor for classes annotated with JPA annotations.

If you need to leverage this kind of mechanism for other persistence technologies, you can configure the `kotlin-noarg` plugin.

As of the Kay release train, Spring Data supports Kotlin immutable class instances and does not require the `kotlin-noarg` plugin if the module uses Spring Data object mappings (such as MongoDB, Redis, Cassandra, and others).

Injecting Dependencies

Our recommendation is to try to favor constructor injection with `val` read-only (and non-nullable when possible) properties, as the following example shows:

```
@Component
class YourBean(
    private val mongoTemplate: MongoTemplate,
    private val solrClient: SolrClient
)
```

Classes with a single constructor have their parameters automatically autowired. That's why there is no need for an explicit `@Autowired constructor` in the example shown above.

If you really need to use field injection, you can use the `lateinit var` construct, as the following example shows:

```
@Component
class YourBean {

    @Autowired
    lateinit var mongoTemplate: MongoTemplate

    @Autowired
    lateinit var solrClient: SolrClient
}
```

Injecting Configuration Properties

In Java, you can inject configuration properties by using annotations (such as `@Value("${property}")`). However, in Kotlin, `$` is a reserved character that is used for string interpolation.

Therefore, if you wish to use the `@Value` annotation in Kotlin, you need to escape the `$` character by writing `@Value("\${property}")`.

If you use Spring Boot, you should probably use `@ConfigurationProperties` instead of `@Value` annotations.

As an alternative, you can customize the property placeholder prefix by declaring the following configuration beans:

```
@Bean
fun propertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {
    setPlaceholderPrefix("%{")
}
```

You can customize existing code (such as Spring Boot actuators or `@LocalServerPort`) that uses the `${…}` syntax, with configuration beans, as the following example shows:

```
@Bean
fun kotlinPropertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {
    setPlaceholderPrefix("%{")
    setIgnoreUnresolvablePlaceholders(true)
}

@Bean
fun defaultPropertyConfigurer() = PropertySourcesPlaceholderConfigurer()
```

Checked Exceptions

Java and Kotlin exception handling are pretty close, with the main difference being that Kotlin treats all exceptions as unchecked exceptions. However, when using proxied objects (for example classes or methods annotated with `@Transactional`), checked exceptions thrown will be wrapped by default in an `UndeclaredThrowableException`.

To get the original exception thrown like in Java, methods should be annotated with `@Throws` to specify explicitly the checked exceptions thrown (for example `@Throws(IOException::class)`).

Annotation Array Attributes

Kotlin annotations are mostly similar to Java annotations, but array attributes (which are extensively used in Spring) behave differently. As explained in the Kotlin documentation you can omit the `value` attribute name, unlike other attributes, and specify it as a `vararg` parameter.

To understand what that means, consider `@RequestMapping` (which is one of the most widely used Spring annotations) as an example. This Java annotation is declared as follows:

```
public @interface RequestMapping {

    @AliasFor("path")
    String[] value() default {};

    @AliasFor("value")
    String[] path() default {};

    RequestMethod[] method() default {};

    // ...
}
```

The typical use case for `@RequestMapping` is to map a handler method to a specific path and method. In Java, you can specify a single value for the annotation array attribute, and it is automatically converted to an array.

That is why one can write `@RequestMapping(value = "/toys", method = RequestMethod.GET)` or `@RequestMapping(path = "/toys", method = RequestMethod.GET)`.

However, in Kotlin, you must write `@RequestMapping("/toys", method = [RequestMethod.GET])` or `@RequestMapping(path = ["/toys"], method = [RequestMethod.GET])` (square brackets need to be specified with named array attributes).

An alternative for this specific `method` attribute (the most common one) is to use a shortcut annotation, such as `@GetMapping`, `@PostMapping`, and others.

If the `@RequestMapping` `method` attribute is not specified, all HTTP methods will be matched, not only the `GET` method.

Testing

This section addresses testing with the combination of Kotlin and Spring Framework. The recommended testing framework is JUnit 5 along with Mockk for mocking.

If you are using Spring Boot, see this related documentation.

Constructor injection

As described in the dedicated section, JUnit 5 allows constructor injection of beans which is pretty useful with Kotlin in order to use `val` instead of `lateinit var`. You can use `@TestConstructor(autowireMode = AutowireMode.ALL)` to enable autowiring for all parameters.

```
@SpringJUnitConfig(TestConfig::class)
@TestConstructor(autowireMode = AutowireMode.ALL)
class OrderServiceIntegrationTests(val orderService: OrderService,
                                   val customerService: CustomerService) {

    // tests that use the injected OrderService and CustomerService
}
```

`PER_CLASS` Lifecycle

Kotlin lets you specify meaningful test function names between backticks (`). As of JUnit 5, Kotlin test classes can use the `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` annotation to enable single instantiation of test classes, which allows the use of `@BeforeAll` and `@AfterAll` annotations on non-static methods, which is a good fit for Kotlin.

You can also change the default behavior to `PER_CLASS` thanks to a `junit-platform.properties` file with a `junit.jupiter.testinstance.lifecycle.default = per_class` property.

The following example demonstrates `@BeforeAll` and `@AfterAll` annotations on non-static methods:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class IntegrationTests {

  val application = Application(8181)
  val client = WebClient.create("http://localhost:8181")

  @BeforeAll
  fun beforeAll() {
    application.start()
  }

  @Test
  fun `Find all users on HTML page`() {
    client.get().uri("/users")
        .accept(TEXT_HTML)
        .retrieve()
        .bodyToMono<String>()
        .test()
        .expectNextMatches { it.contains("Foo") }
        .verifyComplete()
  }

  @AfterAll
  fun afterAll() {
    application.stop()
  }
}
```

Specification-like Tests

You can create specification-like tests with JUnit 5 and Kotlin. The following example shows how to do so:

```
class SpecificationLikeTests {

  @Nested
  @DisplayName("a calculator")
  inner class Calculator {
     val calculator = SampleCalculator()

     @Test
     fun `should return the result of adding the first number to the second number`()
{
        val sum = calculator.sum(2, 4)
        assertEquals(6, sum)
     }

     @Test
     fun `should return the result of subtracting the second number from the first
number`() {
        val subtract = calculator.subtract(4, 2)
        assertEquals(2, subtract)
     }
  }
}
```

`WebTestClient` Type Inference Issue in Kotlin

Due to a type inference issue, you must use the Kotlin `expectBody` extension (such as `.expectBody<String>().isEqualTo("toys")`), since it provides a workaround for the Kotlin issue with the Java API.

See also the related SPR-16057 issue.

8.1.10. Getting Started

The easiest way to learn how to build a Spring application with Kotlin is to follow the dedicated tutorial.

`start.spring.io`

The easiest way to start a new Spring Framework project in Kotlin is to create a new Spring Boot 2 project on start.spring.io.

Choosing the Web Flavor

Spring Framework now comes with two different web stacks: Spring MVC and Spring WebFlux.

Spring WebFlux is recommended if you want to create applications that will deal with latency, long-lived connections, streaming scenarios or if you want to use the web functional Kotlin DSL.

For other use cases, especially if you are using blocking technologies such as JPA, Spring MVC and its annotation-based programming model is the recommended choice.

8.1.11. Resources

We recommend the following resources for people learning how to build applications with Kotlin and the Spring Framework:

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Stackoverflow, with `spring` and `kotlin` tags
- Try Kotlin in your browser
- Kotlin blog
- Awesome Kotlin

Examples

The following Github projects offer examples that you can learn from and possibly even extend:

- spring-boot-kotlin-demo: Regular Spring Boot and Spring Data JPA project
- mixit: Spring Boot 2, WebFlux, and Reactive Spring Data MongoDB
- spring-kotlin-functional: Standalone WebFlux and functional bean definition DSL
- spring-kotlin-fullstack: WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript
- spring-petclinic-kotlin: Kotlin version of the Spring PetClinic Sample Application
- spring-kotlin-deepdive: A step-by-step migration guide for Boot 1.0 and Java to Boot 2.0 and Kotlin
- spring-cloud-gcp-kotlin-app-sample: Spring Boot with Google Cloud Platform Integrations

Issues

The following list categorizes the pending issues related to Spring and Kotlin support:

- Spring Framework
  - Unable to use WebTestClient with mock server in Kotlin
  - Support null-safety at generics, varargs and array elements level
- Kotlin
  - Parent issue for Spring Framework support
  - Kotlin requires type inference where Java doesn't
  - Smart cast regression with open classes

- Impossible to pass not all SAM argument as function
- Support JSR 223 bindings directly via script variables
- Kotlin properties do not override Java-style getters and setters

8.2. Apache Groovy

Groovy is a powerful, optionally typed, and dynamic language, with static-typing and static compilation capabilities. It offers a concise syntax and integrates smoothly with any existing Java application.

The Spring Framework provides a dedicated `ApplicationContext` that supports a Groovy-based Bean Definition DSL. For more details, see The Groovy Bean Definition DSL.

Further support for Groovy, including beans written in Groovy, refreshable script beans, and more is available in Dynamic Language Support.

8.3. Dynamic Language Support

Spring provides comprehensive support for using classes and objects that have been defined by using a dynamic language (such as Groovy) with Spring. This support lets you write any number of classes in a supported dynamic language and have the Spring container transparently instantiate, configure, and dependency inject the resulting objects.

Spring's scripting support primarily targets Groovy and BeanShell. Beyond those specifically supported languages, the JSR-223 scripting mechanism is supported for integration with any JSR-223 capable language provider (as of Spring 4.2), e.g. JRuby.

You can find fully working examples of where this dynamic language support can be immediately useful in Scenarios.

8.3.1. A First Example

The bulk of this chapter is concerned with describing the dynamic language support in detail. Before diving into all of the ins and outs of the dynamic language support, we look at a quick example of a bean defined in a dynamic language. The dynamic language for this first bean is Groovy. (The basis of this example was taken from the Spring test suite. If you want to see equivalent examples in any of the other supported languages, take a look at the source code).

The next example shows the `Messenger` interface, which the Groovy bean is going to implement. Note that this interface is defined in plain Java. Dependent objects that are injected with a reference to the `Messenger` do not know that the underlying implementation is a Groovy script. The following listing shows the `Messenger` interface:

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

The following example defines a class that has a dependency on the `Messenger` interface:

```
package org.springframework.scripting;

public class DefaultBookingService implements BookingService {

    private Messenger messenger;

    public void setMessenger(Messenger messenger) {
        this.messenger = messenger;
    }

    public void processBooking() {
        // use the injected Messenger object...
    }
}
```

The following example implements the `Messenger` interface in Groovy:

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

// import the Messenger interface (written in Java) that is to be implemented
import org.springframework.scripting.Messenger

// define the implementation in Groovy
class GroovyMessenger implements Messenger {

    String message
}
```

To use the custom dynamic language tags to define dynamic-language-backed beans, you need to have the XML Schema preamble at the top of your Spring XML configuration file. You also need to use a Spring `ApplicationContext` implementation as your IoC container. Using the dynamic-language-backed beans with a plain `BeanFactory` implementation is supported, but you have to manage the plumbing of the Spring internals to do so.

For more information on schema-based configuration, see XML Schema-based Configuration.

Finally, the following example shows the bean definitions that effect the injection of the Groovy-defined `Messenger` implementation into an instance of the `DefaultBookingService` class:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://www.springframework.org/schema/lang/spring-lang.xsd">

    <!-- this is the bean definition for the Groovy-backed Messenger implementation
-->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger
-->
    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

The `bookingService` bean (a `DefaultBookingService`) can now use its private `messenger` member variable as normal, because the `Messenger` instance that was injected into it is a `Messenger` instance. There is nothing special going on here — just plain Java and plain Groovy.

Hopefully, the preceding XML snippet is self-explanatory, but do not worry unduly if it is not. Keep reading for the in-depth detail on the whys and wherefores of the preceding configuration.

8.3.2. Defining Beans that Are Backed by Dynamic Languages

This section describes exactly how you define Spring-managed beans in any of the supported dynamic languages.

Note that this chapter does not attempt to explain the syntax and idioms of the supported dynamic languages. For example, if you want to use Groovy to write certain of the classes in your application, we assume that you already know Groovy. If you need further details about the dynamic languages themselves, see Further Resources at the end of this chapter.

Common Concepts

The steps involved in using dynamic-language-backed beans are as follows:

1. Write the test for the dynamic language source code (naturally).
2. Then write the dynamic language source code itself.
3. Define your dynamic-language-backed beans by using the appropriate `<lang:language/>` element in the XML configuration (you can define such beans programmatically by using the Spring API, although you will have to consult the source code for directions on how to do this, as this chapter does not cover this type of advanced configuration). Note that this is an iterative step. You need at least one bean definition for each dynamic language source file (although multiple bean definitions can reference the same source file).

The first two steps (testing and writing your dynamic language source files) are beyond the scope of this chapter. See the language specification and reference manual for your chosen dynamic language and crack on with developing your dynamic language source files. You first want to read the rest of this chapter, though, as Spring's dynamic language support does make some (small) assumptions about the contents of your dynamic language source files.

The <lang:language/> element

The final step in the list in the preceding section involves defining dynamic-language-backed bean definitions, one for each bean that you want to configure (this is no different from normal JavaBean configuration). However, instead of specifying the fully qualified class name of the class that is to be instantiated and configured by the container, you can use the `<lang:language/>` element to define the dynamic language-backed bean.

Each of the supported languages has a corresponding `<lang:language/>` element:

- `<lang:groovy/>` (Groovy)
- `<lang:bsh/>` (BeanShell)
- `<lang:std/>` (JSR-223, e.g. with JRuby)

The exact attributes and child elements that are available for configuration depends on exactly which language the bean has been defined in (the language-specific sections later in this chapter detail this).

Refreshable Beans

One of the (and perhaps the single) most compelling value adds of the dynamic language support in Spring is the "refreshable bean" feature.

A refreshable bean is a dynamic-language-backed bean. With a small amount of configuration, a dynamic-language-backed bean can monitor changes in its underlying source file resource and

then reload itself when the dynamic language source file is changed (for example, when you edit and save changes to the file on the file system).

This lets you deploy any number of dynamic language source files as part of an application, configure the Spring container to create beans backed by dynamic language source files (using the mechanisms described in this chapter), and (later, as requirements change or some other external factor comes into play) edit a dynamic language source file and have any change they make be reflected in the bean that is backed by the changed dynamic language source file. There is no need to shut down a running application (or redeploy in the case of a web application). The dynamic-language-backed bean so amended picks up the new state and logic from the changed dynamic language source file.

This feature is off by default.

Now we can take a look at an example to see how easy it is to start using refreshable beans. To turn on the refreshable beans feature, you have to specify exactly one additional attribute on the `<lang:language/>` element of your bean definition. So, if we stick with the example from earlier in this chapter, the following example shows what we would change in the Spring XML configuration to effect refreshable beans:

```
<beans>

    <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-
delay' attribute -->
    <lang:groovy id="messenger"
            refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds
between checks -->
            script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

That really is all you have to do. The `refresh-check-delay` attribute defined on the `messenger` bean definition is the number of milliseconds after which the bean is refreshed with any changes made to the underlying dynamic language source file. You can turn off the refresh behavior by assigning a negative value to the `refresh-check-delay` attribute. Remember that, by default, the refresh behavior is disabled. If you do not want the refresh behavior, do not define the attribute.

If we then run the following application, we can exercise the refreshable feature. (Please excuse the “jumping-through-hoops-to-pause-the-execution” shenanigans in this next slice of code.) The `System.in.read()` call is only there so that the execution of the program pauses while you (the developer in this scenario) go off and edit the underlying dynamic language source file so that the refresh triggers on the dynamic-language-backed bean when the program resumes execution.

The following listing shows this sample application:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger.getMessage());
        // pause execution while I go off and make changes to the source file...
        System.in.read();
        System.out.println(messenger.getMessage());
    }
}
```

Assume then, for the purposes of this example, that all calls to the `getMessage()` method of `Messenger` implementations have to be changed such that the message is surrounded by quotation marks. The following listing shows the changes that you (the developer) should make to the `Messenger.groovy` source file when the execution of the program is paused:

```
package org.springframework.scripting

class GroovyMessenger implements Messenger {

    private String message = "Bingo"

    public String getMessage() {
        // change the implementation to surround the message in quotes
        return "'" + this.message + "'"
    }

    public void setMessage(String message) {
        this.message = message
    }
}
```

When the program runs, the output before the input pause will be `I Can Do The Frug`. After the change to the source file is made and saved and the program resumes execution, the result of calling the `getMessage()` method on the dynamic-language-backed `Messenger` implementation is `'I Can Do The Frug'` (notice the inclusion of the additional quotation marks).

Changes to a script do not trigger a refresh if the changes occur within the window of the `refresh-check-delay` value. Changes to the script are not actually picked up until a method is called on the dynamic-language-backed bean. It is only when a method is called on a dynamic-language-backed bean that it checks to see if its underlying script source has changed. Any exceptions that relate to

refreshing the script (such as encountering a compilation error or finding that the script file has been deleted) results in a fatal exception being propagated to the calling code.

The refreshable bean behavior described earlier does not apply to dynamic language source files defined with the `<lang:inline-script/>` element notation (see Inline Dynamic Language Source Files). Additionally, it applies only to beans where changes to the underlying source file can actually be detected (for example, by code that checks the last modified date of a dynamic language source file that exists on the file system).

Inline Dynamic Language Source Files

The dynamic language support can also cater to dynamic language source files that are embedded directly in Spring bean definitions. More specifically, the `<lang:inline-script/>` element lets you define dynamic language source immediately inside a Spring configuration file. An example might clarify how the inline script feature works:

```
<lang:groovy id="messenger">
    <lang:inline-script>

package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {
    String message
}

    </lang:inline-script>
    <lang:property name="message" value="I Can Do The Frug" />
</lang:groovy>
```

If we put to one side the issues surrounding whether it is good practice to define dynamic language source inside a Spring configuration file, the `<lang:inline-script/>` element can be useful in some scenarios. For instance, we might want to quickly add a Spring `Validator` implementation to a Spring MVC `Controller`. This is but a moment's work using inline source. (See Scripted Validators for such an example.)

Understanding Constructor Injection in the Context of Dynamic-language-backed Beans

There is one very important thing to be aware of with regard to Spring's dynamic language support. Namely, you can not (currently) supply constructor arguments to dynamic-language-backed beans (and, hence, constructor-injection is not available for dynamic-language-backed beans). In the interests of making this special handling of constructors and properties 100% clear, the following mixture of code and configuration does not work:

An approach that cannot work

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    GroovyMessenger() {}

    // this constructor is not available for Constructor Injection
    GroovyMessenger(String message) {
        this.message = message;
    }

    String message

    String anotherMessage
}
```

```
<lang:groovy id="badMessenger"
    script-source="classpath:Messenger.groovy">
    <!-- this next constructor argument will not be injected into the GroovyMessenger
-->
    <!-- in fact, this isn't even allowed according to the schema -->
    <constructor-arg value="This will not work" />

    <!-- only property values are injected into the dynamic-language-backed object -->
    <lang:property name="anotherMessage" value="Passed straight through to the
dynamic-language-backed object" />

</lang>
```

In practice this limitation is not as significant as it first appears, since setter injection is the injection style favored by the overwhelming majority of developers (we leave the discussion as to whether that is a good thing to another day).

Groovy Beans

This section describes how to use beans defined in Groovy in Spring.

The Groovy homepage includes the following description:

"Groovy is an agile dynamic language for the Java 2 Platform that has many of the features that people like so much in languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax."

If you have read this chapter straight from the top, you have already seen an example of a Groovy-

dynamic-language-backed bean. Now consider another example (again using an example from the Spring test suite):

```
package org.springframework.scripting;

public interface Calculator {

    int add(int x, int y);
}
```

The following example implements the `Calculator` interface in Groovy:

```
// from the file 'calculator.groovy'
package org.springframework.scripting.groovy

class GroovyCalculator implements Calculator {

    int add(int x, int y) {
        x + y
    }
}
```

The following bean definition uses the calculator defined in Groovy:

```
<!-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

Finally, the following small application exercises the preceding configuration:

```
package org.springframework.scripting;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Calculator calc = ctx.getBean("calculator", Calculator.class);
        System.out.println(calc.add(2, 8));
    }
}
```

The resulting output from running the above program is (unsurprisingly) `10`. (For more interesting

examples, see the dynamic language showcase project for a more complex example or see the examples Scenarios later in this chapter).

You must not define more than one class per Groovy source file. While this is perfectly legal in Groovy, it is (arguably) a bad practice. In the interests of a consistent approach, you should (in the opinion of the Spring team) respect the standard Java conventions of one (public) class per source file.

Customizing Groovy Objects by Using a Callback

The `GroovyObjectCustomizer` interface is a callback that lets you hook additional creation logic into the process of creating a Groovy-backed bean. For example, implementations of this interface could invoke any required initialization methods, set some default property values, or specify a custom `MetaClass`. The following listing shows the `GroovyObjectCustomizer` interface definition:

```
public interface GroovyObjectCustomizer {

    void customize(GroovyObject goo);
}
```

The Spring Framework instantiates an instance of your Groovy-backed bean and then passes the created `GroovyObject` to the specified `GroovyObjectCustomizer` (if one has been defined). You can do whatever you like with the supplied `GroovyObject` reference. We expect that most people want to set a custom `MetaClass` with this callback, and the following example shows how to do so:

```
public final class SimpleMethodTracingCustomizer implements GroovyObjectCustomizer {

    public void customize(GroovyObject goo) {
        DelegatingMetaClass metaClass = new DelegatingMetaClass(goo.getMetaClass()) {

            public Object invokeMethod(Object object, String methodName, Object[]
arguments) {
                System.out.println("Invoking '" + methodName + "'.");
                return super.invokeMethod(object, methodName, arguments);
            }
        };
        metaClass.initialize();
        goo.setMetaClass(metaClass);
    }

}
```

A full discussion of meta-programming in Groovy is beyond the scope of the Spring reference manual. See the relevant section of the Groovy reference manual or do a search online. Plenty of articles address this topic. Actually, making use of a `GroovyObjectCustomizer` is easy if you use the Spring namespace support, as the following example shows:

```
<!-- define the GroovyObjectCustomizer just like any other bean -->
<bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer"/>

    <!-- ... and plug it into the desired Groovy bean via the 'customizer-ref'
attribute -->
    <lang:groovy id="calculator"
        script-
source="classpath:org/springframework/scripting/groovy/Calculator.groovy"
        customizer-ref="tracingCustomizer"/>
```

If you do not use the Spring namespace support, you can still use the `GroovyObjectCustomizer` functionality, as the following example shows:

```
<bean id="calculator"
class="org.springframework.scripting.groovy.GroovyScriptFactory">
    <constructor-arg
value="classpath:org/springframework/scripting/groovy/Calculator.groovy"/>
    <!-- define the GroovyObjectCustomizer (as an inner bean) -->
    <constructor-arg>
        <bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer"/>
    </constructor-arg>
</bean>

<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/>
```

You may also specify a Groovy `CompilationCustomizer` (such as an `ImportCustomizer`) or even a full Groovy `CompilerConfiguration` object in the same place as Spring's `GroovyObjectCustomizer`. Furthermore, you may set a common `GroovyClassLoader` with custom configuration for your beans at the `ConfigurableApplicationContext.setClassLoader` level; this also leads to shared `GroovyClassLoader` usage and is therefore recommendable in case of a large number of scripted beans (avoiding an isolated `GroovyClassLoader` instance per bean).

BeanShell Beans

This section describes how to use BeanShell beans in Spring.

The BeanShell homepage includes the following description:

```
BeanShell is a small, free, embeddable Java source interpreter with dynamic language
features, written in Java. BeanShell dynamically runs standard Java syntax and
extends it with common scripting conveniences such as loose types, commands, and
method
closures like those in Perl and JavaScript.
```

In contrast to Groovy, BeanShell-backed bean definitions require some (small) additional configuration. The implementation of the BeanShell dynamic language support in Spring is interesting, because Spring creates a JDK dynamic proxy that implements all of the interfaces that are specified in the `script-interfaces` attribute value of the `<lang:bsh>` element (this is why you must supply at least one interface in the value of the attribute, and, consequently, program to interfaces when you use BeanShell-backed beans). This means that every method call on a BeanShell-backed object goes through the JDK dynamic proxy invocation mechanism.

Now we can show a fully working example of using a BeanShell-based bean that implements the `Messenger` interface that was defined earlier in this chapter. We again show the definition of the `Messenger` interface:

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

The following example shows the BeanShell "implementation" (we use the term loosely here) of the `Messenger` interface:

```
String message;

String getMessage() {
    return message;
}

void setMessage(String aMessage) {
    message = aMessage;
}
```

The following example shows the Spring XML that defines an "instance" of the above "class" (again, we use these terms very loosely here):

```
<lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"
    script-interfaces="org.springframework.scripting.Messenger">

    <lang:property name="message" value="Hello World!" />
</lang:bsh>
```

See Scenarios for some scenarios where you might want to use BeanShell-based beans.

8.3.3. Scenarios

The possible scenarios where defining Spring managed beans in a scripting language would be beneficial are many and varied. This section describes two possible use cases for the dynamic

language support in Spring.

Scripted Spring MVC Controllers

One group of classes that can benefit from using dynamic-language-backed beans is that of Spring MVC controllers. In pure Spring MVC applications, the navigational flow through a web application is, to a large extent, determined by code encapsulated within your Spring MVC controllers. As the navigational flow and other presentation layer logic of a web application needs to be updated to respond to support issues or changing business requirements, it may well be easier to effect any such required changes by editing one or more dynamic language source files and seeing those changes being immediately reflected in the state of a running application.

Remember that, in the lightweight architectural model espoused by projects such as Spring, you typically aim to have a really thin presentation layer, with all the meaty business logic of an application being contained in the domain and service layer classes. Developing Spring MVC controllers as dynamic-language-backed beans lets you change presentation layer logic by editing and saving text files. Any changes to such dynamic language source files is (depending on the configuration) automatically reflected in the beans that are backed by dynamic language source files.

To effect this automatic “pickup” of any changes to dynamic-language-backed beans, you have to enable the “refreshable beans” functionality. See Refreshable Beans for a full treatment of this feature.

The following example shows an `org.springframework.web.servlet.mvc.Controller` implemented by using the Groovy dynamic language:

```
// from the file '/WEB-INF/groovy/FortuneController.groovy'
package org.springframework.showcase.fortune.web

import org.springframework.showcase.fortune.service.FortuneService
import org.springframework.showcase.fortune.domain.Fortune
import org.springframework.web.servlet.ModelAndView
import org.springframework.web.servlet.mvc.Controller

import jakarta.servlet.http.HttpServletRequest
import jakarta.servlet.http.HttpServletResponse

class FortuneController implements Controller {

    @Property FortuneService fortuneService

    ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse httpServletResponse) {
        return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())
    }
}
```

```
<lang:groovy id="fortune"
        refresh-check-delay="3000"
        script-source="/WEB-INF/groovy/FortuneController.groovy">
    <lang:property name="fortuneService" ref="fortuneService"/>
</lang:groovy>
```

Scripted Validators

Another area of application development with Spring that may benefit from the flexibility afforded by dynamic-language-backed beans is that of validation. It can be easier to express complex validation logic by using a loosely typed dynamic language (that may also have support for inline regular expressions) as opposed to regular Java.

Again, developing validators as dynamic-language-backed beans lets you change validation logic by editing and saving a simple text file. Any such changes is (depending on the configuration) automatically reflected in the execution of a running application and would not require the restart of an application.

To effect the automatic “pickup” of any changes to dynamic-language-backed beans, you have to enable the 'refreshable beans' feature. See Refreshable Beans for a full and detailed treatment of this feature.

The following example shows a Spring `org.springframework.validation.Validator` implemented by using the Groovy dynamic language (see Validation using Spring’s Validator interface for a discussion of the `Validator` interface):

```
import org.springframework.validation.Validator
import org.springframework.validation.Errors
import org.springframework.beans.TestBean

class TestBeanValidator implements Validator {

    boolean supports(Class clazz) {
        return TestBean.class.isAssignableFrom(clazz)
    }

    void validate(Object bean, Errors errors) {
        if(bean.name?.trim()?.size() > 0) {
            return
        }
        errors.reject("whitespace", "Cannot be composed wholly of whitespace.")
    }
}
```

8.3.4. Additional Details

This last section contains some additional details related to the dynamic language support.

AOP — Advising Scripted Beans

You can use the Spring AOP framework to advise scripted beans. The Spring AOP framework actually is unaware that a bean that is being advised might be a scripted bean, so all of the AOP use cases and functionality that you use (or aim to use) work with scripted beans. When you advise scripted beans, you cannot use class-based proxies. You must use interface-based proxies.

You are not limited to advising scripted beans. You can also write aspects themselves in a supported dynamic language and use such beans to advise other Spring beans. This really would be an advanced use of the dynamic language support though.

Scoping

In case it is not immediately obvious, scripted beans can be scoped in the same way as any other bean. The `scope` attribute on the various `<lang:language/>` elements lets you control the scope of the underlying scripted bean, as it does with a regular bean. (The default scope is singleton, as it is with "regular" beans.)

The following example uses the `scope` attribute to define a Groovy bean scoped as a prototype:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://www.springframework.org/schema/lang/spring-lang.xsd">

    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy"
scope="prototype">
        <lang:property name="message" value="I Can Do The RoboCop" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

See Bean Scopes in The IoC Container for a full discussion of the scoping support in the Spring Framework.

The `lang` XML schema

The `lang` elements in Spring XML configuration deal with exposing objects that have been written in a dynamic language (such as Groovy or BeanShell) as beans in the Spring container.

These elements (and the dynamic language support) are comprehensively covered in Dynamic

Language Support. See that section for full details on this support and the `lang` elements.

To use the elements in the `lang` schema, you need to have the following preamble at the top of your Spring XML configuration file. The text in the following snippet references the correct schema so that the tags in the `lang` namespace are available to you:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://www.springframework.org/schema/lang/spring-lang.xsd">

    <!-- bean definitions here -->

</beans>
```

8.3.5. Further Resources

The following links go to further resources about the various dynamic languages referenced in this chapter:

- The Groovy homepage
- The BeanShell homepage
- The JRuby homepage

Chapter 9. Appendix

This part of the reference documentation covers topics that apply to multiple modules within the core Spring Framework.

9.1. Spring Properties

`SpringProperties` is a static holder for properties that control certain low-level aspects of the Spring Framework. Users can configure these properties via JVM system properties or programmatically via the `SpringProperties.setProperty(String key, String value)` method. The latter may be necessary if the deployment environment disallows custom JVM system properties. As an alternative, these properties may be configured in a `spring.properties` file in the root of the classpath — for example, deployed within the application's JAR file.

The following table lists all currently supported Spring properties.

Table 38. Supported Spring Properties

Name	Description
`spring.beaninfo.ignore`	Instructs Spring to use the `Introspector.IGNORE_ALL_BEANINFO` mode when calling the JavaBeans `Introspector`. See `CachedIntrospectionResults` for details.
`spring.expression.compiler.mode`	The mode to use when compiling expressions for the Spring Expression Language.
`spring.getenv.ignore`	Instructs Spring to ignore operating system environment variables if a Spring `Environment` property — for example, a placeholder in a configuration String — isn't resolvable otherwise. See `AbstractEnvironment` for details.
`spring.index.ignore`	Instructs Spring to ignore the components index located in `META-INF/spring.components`. See Generating an Index of Candidate Components.
`spring.jdbc.getParameterType.ignore`	Instructs Spring to ignore `java.sql.ParameterMetaData.getParameterType` completely. See the note in Batch Operations with a List of Objects.
`spring.jndi.ignore`	Instructs Spring to ignore a default JNDI environment, as an optimization for scenarios where nothing is ever to be found for such JNDI fallback searches to begin with, avoiding the repeated JNDI lookup overhead. See `JndiLocatorDelegate` for details.

Name	Description
`spring.objenesis.ignore`	Instructs Spring to ignore Objenesis, not even attempting to use it. See `SpringObjenesis` for details.
`spring.test.constructor.autowire.mode`	The default *test constructor autowire mode* to use if `@TestConstructor` is not present on a test class. See Changing the default test constructor autowire mode.
`spring.test.context.cache.maxSize`	The maximum size of the context cache in the *Spring TestContext Framework*. See Context Caching.
`spring.test.enclosing.configuration`	The default *enclosing configuration inheritance mode* to use if `@NestedTestConfiguration` is not present on a test class. See Changing the default enclosing configuration inheritance mode.

Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, Phillip Webb, Rob Winch, Brian Clozel, Stephane Nicoll, Sebastien Deleuze, Jay Bryant, Mark Paluch