Spring Boot Reference Guide

2.0.0.RC1

Phillip Webb , Dave Syer , Josh Long , Stéphane Nicoll , Rob Winch , Andy Wilkinson , Marcel Overdijk , Christian Dupuis , Sébastien Deleuze , Michael Simons , Vedran Pavi# , Jay Bryant

Table of Contents

Part I. Spring Boot Documentation

This section provides a brief overview of Spring Boot reference documentation. It serves as a map for the rest of the document.

1. About the Documentation

The Spring Boot reference guide is available as

- HTML
- PDF
- EPUB

The latest copy is available at docs.spring.io/spring-boot/docs/current/reference.

2. Getting Help

If you have trouble with Spring Boot, we would like to help.

- Try the How-to documents. They provide solutions to the most common questions.
- Learn the Spring basics. Spring Boot builds on many other Spring projects. Check the spring.io website for a wealth of reference documentation. If you are starting out with Spring, try one of the guides.
- Ask a question. We monitor stackoverflow.com for questions tagged with `spring-boot`.
- Report bugs with Spring Boot at github.com/spring-projects/spring-boot/issues.

> **Note**
>
> All of Spring Boot is open source, including the documentation. If you find problems with the docs or if you want to improve them, please get involved.

3. First Steps

If you are getting started with Spring Boot or 'Spring' in general, start with the following topics:

- **From scratch:** Overview | Requirements | Installation
- **Tutorial:** Part 1 | Part 2
- **Running your example:** Part 1 | Part 2

4. Working with Spring Boot

Ready to actually start using Spring Boot? We have you covered:

- **Build systems:** Maven | Gradle | Ant | Starters
- **Best practices:** Code Structure | @Configuration | @EnableAutoConfiguration | Beans and Dependency Injection
- **Running your code** IDE | Packaged | Maven | Gradle
- **Packaging your app:** Production jars
- **Spring Boot CLI:** Using the CLI

5. Learning about Spring Boot Features

Need more details about Spring Boot’s core features? The following content is for you:

- **Core Features:** SpringApplication | External Configuration | Profiles | Logging
- **Web Applications:** MVC | Embedded Containers
- **Working with data:** SQL | NO-SQL
- **Messaging:** Overview | JMS
- **Testing:** Overview | Boot Applications | Utils
- **Extending:** Auto-configuration | @Conditions

6. Moving to Production

When you are ready to push your Spring Boot application to production, we have some tricks that you might like:

- **Management endpoints:** Overview | Customization
- **Connection options:** HTTP | JMX
- **Monitoring:** Metrics | Auditing | Tracing | Process

7. Advanced Topics

Finally, we have a few topics for more advanced users:

- **Spring Boot Applications Deployment:** Cloud Deployment | OS Service
- **Build tool plugins:** Maven | Gradle
- **Appendix:** Application Properties | Auto-configuration classes | Executable Jars

Part II. Getting Started

If you are getting started with Spring Boot, or “Spring” in general, start by reading this section. It answers the basic “what?”, “how?” and “why?” questions. It includes an introduction to Spring Boot, along with installation instructions. We then walk you through building your first Spring Boot application, discussing some core principles as we go.

8. Introducing Spring Boot

Spring Boot makes it easy to create stand-alone, production-grade Spring-based Applications that you can run. We take an opinionated view of the Spring platform and third-party libraries, so that you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

You can use Spring Boot to create Java applications that can be started by using `java -jar` or more traditional war deployments. We also provide a command line tool that runs “spring scripts”.

Our primary goals are:

- Provide a radically faster and widely accessible getting-started experience for all Spring development.
- Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (such as embedded servers, security, metrics, health checks, and externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

9. System Requirements

Spring Boot 2.0.0.RC1 requires Java 8 and Spring Framework 5.0.3.RELEASE or above. Explicit build support is provided for Maven 3.2+ and Gradle 4.

9.1 Servlet Containers

Spring Boot supports the following embedded servlet containers:

Name	Servlet Version
Tomcat 8.5	3.1
Jetty 9.4	3.1
Undertow 1.3	3.1

You can also deploy Spring Boot applications to any Servlet 3.0+ compatible container.

10. Installing Spring Boot

Spring Boot can be used with “classic” Java development tools or installed as a command line tool. Either way, you need Java SDK v1.8 or higher. Before you begin, you should check your current Java installation by using the following command:

```
$ java -version
```

If you are new to Java development or if you want to experiment with Spring Boot, you might want to try the Spring Boot CLI (Command Line Interface) first. Otherwise, read on for “classic” installation instructions.

10.1 Installation Instructions for the Java Developer

You can use Spring Boot in the same way as any standard Java library. To do so, include the appropriate `spring-boot-*.jar` files on your classpath. Spring Boot does not require any special tools integration, so you can use any IDE or text editor. Also, there is nothing special about a Spring Boot application, so you can run and debug a Spring Boot application as you would any other Java program.

Although you *could* copy Spring Boot jars, we generally recommend that you use a build tool that supports dependency management (such as Maven or Gradle).

Maven Installation

Spring Boot is compatible with Apache Maven 3.2 or above. If you do not already have Maven installed, you can follow the instructions at maven.apache.org.

Tip

On many operating systems, Maven can be installed with a package manager. If you use OSX Homebrew, try `brew install maven`. Ubuntu users can run `sudo apt-get install maven`. Windows users with Chocolatey can run `choco install maven` from an elevated (administrator) prompt.

Spring Boot dependencies use the `org.springframework.boot` `groupId`. Typically, your Maven POM file inherits from the `spring-boot-starter-parent` project and declares dependencies to one or more “Starters”. Spring Boot also provides an optional Maven plugin to create executable jars.

The following listing shows a typical `pom.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.example</groupId>
 <artifactId>myproject</artifactId>
 <version>0.0.1-SNAPSHOT</version>

 <!-- Inherit defaults from Spring Boot -->
 <parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RC1</version>
 </parent>

 <!-- Add typical dependencies for a web application -->
```

```
  <dependencies>
   <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
  </dependencies>

  <!-- Package as an executable jar -->
  <build>
   <plugins>
    <plugin>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
   </plugins>
  </build>

  <!-- Add Spring repositories -->
  <!-- (you don't need this if you are using a .RELEASE version) -->
  <repositories>
   <repository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/snapshot</url>
    <snapshots><enabled>true</enabled></snapshots>
   </repository>
   <repository>
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
   </repository>
  </repositories>
  <pluginRepositories>
   <pluginRepository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/snapshot</url>
   </pluginRepository>
   <pluginRepository>
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
   </pluginRepository>
  </pluginRepositories>
</project>
```

Tip

The `spring-boot-starter-parent` is a great way to use Spring Boot, but it might not be suitable all of the time. Sometimes you may need to inherit from a different parent POM, or you might not like our default settings. In those cases, see the section called "Using Spring Boot without the Parent POM" for an alternative solution that uses an `import` scope.

Gradle Installation

Spring Boot is compatible with Gradle 4. If you do not already have Gradle installed, you can follow the instructions at www.gradle.org/.

Spring Boot dependencies can be declared by using the `org.springframework.boot` `group`. Typically, your project declares dependencies to one or more "Starters". Spring Boot provides a useful Gradle plugin that can be used to simplify dependency declarations and to create executable jars.

Gradle Wrapper

The Gradle Wrapper provides a nice way of "obtaining" Gradle when you need to build a project. It is a small script and library that you commit alongside your code to bootstrap the build process. See docs.gradle.org/4.2.1/userguide/gradle_wrapper.html for details.

The following example shows a typical `build.gradle` file:

```
buildscript {
 repositories {
  jcenter()
  maven { url 'http://repo.spring.io/snapshot' }
  maven { url 'http://repo.spring.io/milestone' }
 }
 dependencies {
  classpath 'org.springframework.boot:spring-boot-gradle-plugin:2.0.0.RC1'
 }
}

apply plugin: 'java'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

jar {
 baseName = 'myproject'
 version =  '0.0.1-SNAPSHOT'
}

repositories {
 jcenter()
 maven { url "http://repo.spring.io/snapshot" }
 maven { url "http://repo.spring.io/milestone" }
}

dependencies {
 compile("org.springframework.boot:spring-boot-starter-web")
 testCompile("org.springframework.boot:spring-boot-starter-test")
}
```

10.2 Installing the Spring Boot CLI

The Spring Boot CLI (Command Line Interface) is a command line tool that you can use to quickly prototype with Spring. It lets you run Groovy scripts, which means that you have a familiar Java-like syntax without so much boilerplate code.

You do not need to use the CLI to work with Spring Boot, but it is definitely the quickest way to get a Spring application off the ground.

Manual Installation

You can download the Spring CLI distribution from the Spring software repository:

- spring-boot-cli-2.0.0.RC1-bin.zip
- spring-boot-cli-2.0.0.RC1-bin.tar.gz

Cutting edge snapshot distributions are also available.

Once downloaded, follow the INSTALL.txt instructions from the unpacked archive. In summary, there is a `spring` script (`spring.bat` for Windows) in a `bin/` directory in the `.zip` file. Alternatively, you can use `java -jar` with the `.jar` file (the script helps you to be sure that the classpath is set correctly).

Installation with SDKMAN!

SDKMAN! (The Software Development Kit Manager) can be used for managing multiple versions of various binary SDKs, including Groovy and the Spring Boot CLI. Get SDKMAN! from sdkman.io and install Spring Boot by using the following commands:

```
$ sdk install springboot
$ spring --version
Spring Boot v2.0.0.RC1
```

If you develop features for the CLI and want easy access to the version you built, use the following commands:

```
$ sdk install springboot dev /path/to/spring-boot/spring-boot-cli/target/spring-boot-cli-2.0.0.RC1-bin/
spring-2.0.0.RC1/
$ sdk default springboot dev
$ spring --version
Spring CLI v2.0.0.RC1
```

The preceding instructions install a local instance of `spring` called the `dev` instance. It points at your target build location, so every time you rebuild Spring Boot, `spring` is up-to-date.

You can see it by running the following command:

```
$ sdk ls springboot

================================================================================
Available Springboot Versions
================================================================================
> + dev
* 2.0.0.RC1

================================================================================
+ - local version
* - installed
> - currently in use
================================================================================
```

OSX Homebrew Installation

If you are on a Mac and use Homebrew, you can install the Spring Boot CLI by using the following commands:

```
$ brew tap pivotal/tap
$ brew install springboot
```

Homebrew installs `spring` to `/usr/local/bin`.

Note

If you do not see the formula, your installation of brew might be out-of-date. In that case, run `brew update` and try again.

MacPorts Installation

If you are on a Mac and use MacPorts, you can install the Spring Boot CLI by using the following command:

```
$ sudo port install spring-boot-cli
```

Command-line Completion

The Spring Boot CLI includes scripts that provide command completion for the BASH and zsh shells. You can `source` the script (also named `spring`) in any shell or put it in your personal or system-wide bash completion initialization. On a Debian system, the system-wide scripts are in `/shell-completion/`

bash and all scripts in that directory are executed when a new shell starts. For example, to run the script manually if you have installed by using SDKMAN!, use the following commands:

```
$ . ~/.sdkman/candidates/springboot/current/shell-completion/bash/spring
$ spring <HIT TAB HERE>
  grab  help  jar  run  test  version
```

Note

If you install the Spring Boot CLI by using Homebrew or MacPorts, the command-line completion scripts are automatically registered with your shell.

Quick-start Spring CLI Example

You can use the following web application to test your installation. To start, create a file called `app.groovy`, as follows:

```
@RestController
class ThisWillActuallyRun {

 @RequestMapping("/")
 String home() {
  "Hello World!"
 }

}
```

Then run it from a shell, as follows:

```
$ spring run app.groovy
```

Note

The first run of your application is slow, as dependencies are downloaded. Subsequent runs are much quicker.

Open `localhost:8080` in your favorite web browser. You should see the following output:

```
Hello World!
```

10.3 Upgrading from an Earlier Version of Spring Boot

If you are upgrading from an earlier release of Spring Boot, check the "migration guide" on the project wiki that provides detailed upgrade instructions. Check also the "release notes" for a list of "new and noteworthy" features for each release.

To upgrade an existing CLI installation, use the appropriate package manager command (for example, `brew upgrade`) or, if you manually installed the CLI, follow the standard instructions, remembering to update your PATH environment variable to remove any older references.

11. Developing Your First Spring Boot Application

This section describes how to develop a simple “Hello World!” web application that highlights some of Spring Boot’s key features. We use Maven to build this project, since most IDEs support it.

> **Tip**
>
> The spring.io web site contains many “Getting Started” guides that use Spring Boot. If you need to solve a specific problem, check there first.
>
> You can shortcut the steps below by going to start.spring.io and choosing the "Web" starter from the dependencies searcher. Doing so generates a new project structure so that you can start coding right away. Check the Spring Initializr documentation for more details.

Before we begin, open a terminal and run the following commands to ensure that you have valid versions of Java and Maven installed:

```
$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

```
$ mvn -v
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T16:41:47+00:00)
Maven home: /usr/local/Cellar/maven/3.3.9/libexec
Java version: 1.8.0_102, vendor: Oracle Corporation
```

> **Note**
>
> This sample needs to be created in its own folder. Subsequent instructions assume that you have created a suitable folder and that it is your current directory.

11.1 Creating the POM

We need to start by creating a Maven `pom.xml` file. The `pom.xml` is the recipe that is used to build your project. Open your favorite text editor and add the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>com.example</groupId>
 <artifactId>myproject</artifactId>
 <version>0.0.1-SNAPSHOT</version>

 <parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RC1</version>
 </parent>

 <!-- Additional lines to be added here... -->

 <!-- (you don't need this if you are using a .RELEASE version) -->
 <repositories>
  <repository>
   <id>spring-snapshots</id>
   <url>http://repo.spring.io/snapshot</url>
```

```
    <snapshots><enabled>true</enabled></snapshots>
   </repository>
   <repository>
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
   </repository>
  </repositories>
  <pluginRepositories>
   <pluginRepository>
    <id>spring-snapshots</id>
    <url>http://repo.spring.io/snapshot</url>
   </pluginRepository>
   <pluginRepository>
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
   </pluginRepository>
  </pluginRepositories>
</project>
```

The preceding listing should give you a working build. You can test it by running `mvn package` (for now, you can ignore the “jar will be empty - no content was marked for inclusion!” warning).

Note

At this point, you could import the project into an IDE (most modern Java IDEs include built-in support for Maven). For simplicity, we continue to use a plain text editor for this example.

11.2 Adding Classpath Dependencies

Spring Boot provides a number of “Starters” that let you add jars to your classpath. Our sample application has already used `spring-boot-starter-parent` in the `parent` section of the POM. The `spring-boot-starter-parent` is a special starter that provides useful Maven defaults. It also provides a `dependency-management` section so that you can omit `version` tags for “blessed” dependencies.

Other “Starters” provide dependencies that you are likely to need when developing a specific type of application. Since we are developing a web application, we add a `spring-boot-starter-web` dependency. Before that, we can look at what we currently have by running the following command:

```
$ mvn dependency:tree

[INFO] com.example:myproject:jar:0.0.1-SNAPSHOT
```

The `mvn dependency:tree` command prints a tree representation of your project dependencies. You can see that `spring-boot-starter-parent` provides no dependencies by itself. To add the necessary dependencies, edit your `pom.xml` and add the `spring-boot-starter-web` dependency immediately below the `parent` section:

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
 </dependency>
</dependencies>
```

If you run `mvn dependency:tree` again, you see that there are now a number of additional dependencies, including the Tomcat web server and Spring Boot itself.

11.3 Writing the Code

To finish our application, we need to create a single Java file. By default, Maven compiles sources from `src/main/java`, so you need to create that folder structure and then add a file named `src/main/java/Example.java` to contain the following code:

```
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

 @RequestMapping("/")
 String home() {
  return "Hello World!";
 }

 public static void main(String[] args) throws Exception {
  SpringApplication.run(Example.class, args);
 }

}
```

Although there is not much code here, quite a lot is going on. We step through the important parts in the next few sections.

The @RestController and @RequestMapping Annotations

The first annotation on our `Example` class is `@RestController`. This is known as a *stereotype* annotation. It provides hints for people reading the code and for Spring that the class plays a specific role. In this case, our class is a web `@Controller`, so Spring considers it when handling incoming web requests.

The `@RequestMapping` annotation provides “routing” information. It tells Spring that any HTTP request with the `/` path should be mapped to the `home` method. The `@RestController` annotation tells Spring to render the resulting string directly back to the caller.

Tip

The `@RestController` and `@RequestMapping` annotations are Spring MVC annotations. (They are not specific to Spring Boot.) See the MVC section in the Spring Reference Documentation for more details.

The @EnableAutoConfiguration Annotation

The second class-level annotation is `@EnableAutoConfiguration`. This annotation tells Spring Boot to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since `spring-boot-starter-web` added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

Starters and Auto-Configuration

Auto-configuration is designed to work well with “Starters”, but the two concepts are not directly tied. You are free to pick and choose jar dependencies outside of the starters. Spring Boot still does its best to auto-configure your application.

The “main” Method

The final part of our application is the `main` method. This is just a standard method that follows the Java convention for an application entry point. Our main method delegates to Spring Boot’s `SpringApplication` class by calling `run`. `SpringApplication` bootstraps our application, starting Spring, which, in turn, starts the auto-configured Tomcat web server. We need to pass `Example.class` as an argument to the `run` method to tell `SpringApplication` which is the primary Spring component. The `args` array is also passed through to expose any command-line arguments.

11.4 Running the Example

At this point, your application should work. Since you used the `spring-boot-starter-parent` POM, you have a useful `run` goal that you can use to start the application. Type `mvn spring-boot:run` from the root project directory to start the application. You should see output similar to the following:

```
$ mvn spring-boot:run

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v2.0.0.RC1)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

To gracefully exit the application, press `ctrl-c`.

11.5 Creating an Executable Jar

We finish our example by creating a completely self-contained executable jar file that we could run in production. Executable jars (sometimes called “fat jars”) are archives containing your compiled classes along with all of the jar dependencies that your code needs to run.

Executable jars and Java

Java does not provide a standard way to load nested jar files (jar files that are themselves contained within a jar). This can be problematic if you are looking to distribute a self-contained application.

To solve this problem, many developers use “uber” jars. An uber jar packages all the classes from all the application’s dependencies into a single archive. The problem with this approach is that it becomes hard to see which libraries are in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars.

Spring Boot takes a different approach and lets you actually nest jars directly.

To create an executable jar, we need to add the `spring-boot-maven-plugin` to our `pom.xml`. To do so, insert the following lines just below the `dependencies` section:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
  </plugin>
 </plugins>
</build>
```

> **Note**
>
> The `spring-boot-starter-parent` POM includes `<executions>` configuration to bind the `repackage` goal. If you do not use the parent POM, you need to declare this configuration yourself. See the plugin documentation for details.

Save your `pom.xml` and run `mvn package` from the command line, as follows:

```
$ mvn package

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building myproject 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] .... ..
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ myproject ---
[INFO] Building jar: /Users/developer/example/spring-boot-example/target/myproject-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.0.RC1:repackage (default) @ myproject ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

If you look in the `target` directory, you should see `myproject-0.0.1-SNAPSHOT.jar`. The file should be around 10 MB in size. If you want to peek inside, you can use `jar tvf`, as follows:

```
$ jar tvf target/myproject-0.0.1-SNAPSHOT.jar
```

You should also see a much smaller file named `myproject-0.0.1-SNAPSHOT.jar.original` in the `target` directory. This is the original jar file that Maven created before it was repackaged by Spring Boot.

To run that application, use the `java -jar` command, as follows:

```
$ java -jar target/myproject-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v2.0.0.RC1)
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.536 seconds (JVM running for 2.864)
```

As before, to exit the application, press `ctrl-c`.

12. What to Read Next

Hopefully, this section provided some of the Spring Boot basics and got you on your way to writing your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and check out some of the getting started guides that solve specific “How do I do that with Spring?” problems. We also have Spring Boot-specific “How-to” reference documentation.

The Spring Boot repository also has a bunch of samples you can run. The samples are independent of the rest of the code (that is, you do not need to build the rest to run or use the samples).

Otherwise, the next logical step is to read *Part III, “Using Spring Boot”*. If you are really impatient, you could also jump ahead and read about *Spring Boot features*.

Part III. Using Spring Boot

This section goes into more detail about how you should use Spring Boot. It covers topics such as build systems, auto-configuration, and how to run your applications. We also cover some Spring Boot best practices. Although there is nothing particularly special about Spring Boot (it is just another library that you can consume), there are a few recommendations that, when followed, make your development process a little easier.

If you are starting out with Spring Boot, you should probably read the *Getting Started* guide before diving into this section.

13. Build Systems

It is strongly recommended that you choose a build system that supports *dependency management* and that can consume artifacts published to the “Maven Central” repository. We would recommend that you choose Maven or Gradle. It is possible to get Spring Boot to work with other build systems (Ant, for example), but they are not particularly well supported.

13.1 Dependency Management

Each release of Spring Boot provides a curated list of dependencies that it supports. In practice, you do not need to provide a version for any of these dependencies in your build configuration, as Spring Boot manages that for you. When you upgrade Spring Boot itself, these dependencies are upgraded as well in a consistent way.

> **Note**
>
> You can still specify a version and override Spring Boot’s recommendations if you need to do so.

The curated list contains all the spring modules that you can use with Spring Boot as well as a refined list of third party libraries. The list is available as a standard Bills of Materials (`spring-boot-dependencies`) that can be used with both Maven and Gradle.

> **Warning**
>
> Each release of Spring Boot is associated with a base version of the Spring Framework. We **highly** recommend that you not specify its version.

13.2 Maven

Maven users can inherit from the `spring-boot-starter-parent` project to obtain sensible defaults. The parent project provides the following features:

- Java 1.8 as the default compiler level.
- UTF-8 source encoding.
- A Dependency Management section, inherited from the spring-boot-dependencies pom, that manages the versions of common dependencies. This dependency management lets you omit <version> tags for those dependencies when used in your own pom.
- Sensible resource filtering.
- Sensible plugin configuration (exec plugin, Git commit ID, and shade).
- Sensible resource filtering for `application.properties` and `application.yml` including profile-specific files (for example, `application-dev.properties` and `application-dev.yml`)

Note that, since the `application.properties` and `application.yml` files accept Spring style placeholders (`${…}`), the Maven filtering is changed to use `@..@` placeholders. (You can override that by setting a Maven property called `resource.delimiter`.)

Inheriting the Starter Parent

To configure your project to inherit from the `spring-boot-starter-parent`, set the `parent` as follows:

```
<!-- Inherit defaults from Spring Boot -->
<parent>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-parent</artifactId>
 <version>2.0.0.RC1</version>
</parent>
```

Note

You should need to specify only the Spring Boot version number on this dependency. If you import additional starters, you can safely omit the version number.

With that setup, you can also override individual dependencies by overriding a property in your own project. For instance, to upgrade to another Spring Data release train, you would add the following to your `pom.xml`:

```
<properties>
 <spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>
</properties>
```

Tip

Check the `spring-boot-dependencies` pom for a list of supported properties.

Using Spring Boot without the Parent POM

Not everyone likes inheriting from the `spring-boot-starter-parent` POM. You may have your own corporate standard parent that you need to use or you may prefer to explicitly declare all your Maven configuration.

If you do not want to use the `spring-boot-starter-parent`, you can still keep the benefit of the dependency management (but not the plugin management) by using a `scope=import` dependency, as follows:

```
<dependencyManagement>
  <dependencies>
  <dependency>
   <!-- Import dependency management from Spring Boot -->
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>2.0.0.RC1</version>
   <type>pom</type>
         <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

The preceding sample setup does not let you override individual dependencies by using a property, as explained above. To achieve the same result, you need to add an entry in the `dependencyManagement` of your project **before** the `spring-boot-dependencies` entry. For instance, to upgrade to another Spring Data release train, you could add the following element to your `pom.xml`:

```
<dependencyManagement>
```

```
 <dependencies>
  <!-- Override Spring Data release train provided by Spring Boot -->
  <dependency>
   <groupId>org.springframework.data</groupId>
   <artifactId>spring-data-releasetrain</artifactId>
   <version>Fowler-SR2</version>
   <scope>import</scope>
   <type>pom</type>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-dependencies</artifactId>
   <version>2.0.0.RC1</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

Note

In the preceding example, we specify a *BOM*, but any dependency type can be overridden in the same way.

Using the Spring Boot Maven Plugin

Spring Boot includes a Maven plugin that can package the project as an executable jar. Add the plugin to your `<plugins>` section if you want to use it, as shown in the following example:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
  </plugin>
 </plugins>
</build>
```

Note

If you use the Spring Boot starter parent pom, you need to add only the plugin. There is no need to configure it unless you want to change the settings defined in the parent.

13.3 Gradle

To learn about using Spring Boot with Gradle, please refer to the documentation for Spring Boot's Gradle plugin:

- Reference (HTML and PDF)
- API

13.4 Ant

It is possible to build a Spring Boot project using Apache Ant+Ivy. The `spring-boot-antlib` "AntLib" module is also available to help Ant create executable jars.

To declare dependencies, a typical `ivy.xml` file looks something like the following example:

```
<ivy-module version="2.0">
 <info organisation="org.springframework.boot" module="spring-boot-sample-ant" />
 <configurations>
  <conf name="compile" description="everything needed to compile this module" />
  <conf name="runtime" extends="compile" description="everything needed to run this module" />
 </configurations>
 <dependencies>
  <dependency org="org.springframework.boot" name="spring-boot-starter"
   rev="${spring-boot.version}" conf="compile" />
 </dependencies>
</ivy-module>
```

A typical `build.xml` looks like the following example:

```
<project
 xmlns:ivy="antlib:org.apache.ivy.ant"
 xmlns:spring-boot="antlib:org.springframework.boot.ant"
 name="myapp" default="build">

 <property name="spring-boot.version" value="2.0.0.RC1" />

 <target name="resolve" description="--> retrieve dependencies with ivy">
  <ivy:retrieve pattern="lib/[conf]/[artifact]-[type]-[revision].[ext]" />
 </target>

 <target name="classpaths" depends="resolve">
  <path id="compile.classpath">
   <fileset dir="lib/compile" includes="*.jar" />
  </path>
 </target>

 <target name="init" depends="classpaths">
  <mkdir dir="build/classes" />
 </target>

 <target name="compile" depends="init" description="compile">
  <javac srcdir="src/main/java" destdir="build/classes" classpathref="compile.classpath" />
 </target>

 <target name="build" depends="compile">
  <spring-boot:exejar destfile="build/myapp.jar" classes="build/classes">
   <spring-boot:lib>
    <fileset dir="lib/runtime" />
   </spring-boot:lib>
  </spring-boot:exejar>
 </target>
</project>
```

Tip

If you do not want to use the `spring-boot-antlib` module, see the *Section 85.9, “Build an Executable Archive from Ant without Using `spring-boot-antlib`”* “How-to” .

13.5 Starters

Starters are a set of convenient dependency descriptors that you can include in your application. You get a one-stop shop for all the Spring and related technologies that you need without having to hunt through sample code and copy-paste loads of dependency descriptors. For example, if you want to get started using Spring and JPA for database access, include the `spring-boot-starter-data-jpa` dependency in your project.

The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

What’s in a name

All **official** starters follow a similar naming pattern; `spring-boot-starter-*`, where `*` is a particular type of application. This naming structure is intended to help when you need to find a starter. The Maven integration in many IDEs lets you search dependencies by name. For example, with the appropriate Eclipse or STS plugin installed, you can press `ctrl-space` in the POM editor and type “spring-boot-starter” for a complete list.

As explained in the “Creating Your Own Starter” section, third party starters should not start with `spring-boot`, as it is reserved for official Spring Boot artifacts. Rather, a third-party starter typically starts with the name of the project. For example, a third-party starter project called `thirdpartyproject` would typically be named `thirdpartyproject-spring-boot-starter`.

The following application starters are provided by Spring Boot under the `org.springframework.boot` group:

Table 13.1. Spring Boot application starters

Name	Description	Pom
`spring-boot-starter`	Core starter, including auto-configuration support, logging and YAML	Pom
`spring-boot-starter-activemq`	Starter for JMS messaging using Apache ActiveMQ	Pom
`spring-boot-starter-amqp`	Starter for using Spring AMQP and Rabbit MQ	Pom
`spring-boot-starter-aop`	Starter for aspect-oriented programming with Spring AOP and AspectJ	Pom
`spring-boot-starter-artemis`	Starter for JMS messaging using Apache Artemis	Pom
`spring-boot-starter-batch`	Starter for using Spring Batch	Pom
`spring-boot-starter-cache`	Starter for using Spring Framework’s caching support	Pom
`spring-boot-starter-cloud-connectors`	Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku	Pom
`spring-boot-starter-data-cassandra`	Starter for using Cassandra distributed database and Spring Data Cassandra	Pom

Name	Description	Pom
`spring-boot-starter-data-cassandra-reactive`	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive	Pom
`spring-boot-starter-data-couchbase`	Starter for using Couchbase document-oriented database and Spring Data Couchbase	Pom
`spring-boot-starter-data-couchbase-reactive`	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive	Pom
`spring-boot-starter-data-elasticsearch`	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch	Pom
`spring-boot-starter-data-jpa`	Starter for using Spring Data JPA with Hibernate	Pom
`spring-boot-starter-data-ldap`	Starter for using Spring Data LDAP	Pom
`spring-boot-starter-data-mongodb`	Starter for using MongoDB document-oriented database and Spring Data MongoDB	Pom
`spring-boot-starter-data-mongodb-reactive`	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive	Pom
`spring-boot-starter-data-neo4j`	Starter for using Neo4j graph database and Spring Data Neo4j	Pom
`spring-boot-starter-data-redis`	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client	Pom
`spring-boot-starter-data-redis-reactive`	Starter for using Redis key-value data store with Spring Data Redis reactive and the Lettuce client	Pom
`spring-boot-starter-data-rest`	Starter for exposing Spring Data repositories over REST using Spring Data REST	Pom

Name	Description	Pom
`spring-boot-starter-data-solr`	Starter for using the Apache Solr search platform with Spring Data Solr	Pom
`spring-boot-starter-freemarker`	Starter for building MVC web applications using FreeMarker views	Pom
`spring-boot-starter-groovy-templates`	Starter for building MVC web applications using Groovy Templates views	Pom
`spring-boot-starter-hateoas`	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS	Pom
`spring-boot-starter-integration`	Starter for using Spring Integration	Pom
`spring-boot-starter-jdbc`	Starter for using JDBC with the Tomcat JDBC connection pool	Pom
`spring-boot-starter-jersey`	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to `spring-boot-starter-web`	Pom
`spring-boot-starter-jooq`	Starter for using jOOQ to access SQL databases. An alternative to `spring-boot-starter-data-jpa` or `spring-boot-starter-jdbc`	Pom
`spring-boot-starter-json`	Starter for reading and writing json	Pom
`spring-boot-starter-jta-atomikos`	Starter for JTA transactions using Atomikos	Pom
`spring-boot-starter-jta-bitronix`	Starter for JTA transactions using Bitronix	Pom
`spring-boot-starter-jta-narayana`	Spring Boot Narayana JTA Starter	Pom
`spring-boot-starter-mail`	Starter for using Java Mail and Spring Framework’s email sending support	Pom

Name	Description	Pom
`spring-boot-starter-mustache`	Starter for building web applications using Mustache views	Pom
`spring-boot-starter-quartz`	Spring Boot Quartz Starter	Pom
`spring-boot-starter-security`	Starter for using Spring Security	Pom
`spring-boot-starter-test`	Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito	Pom
`spring-boot-starter-thymeleaf`	Starter for building MVC web applications using Thymeleaf views	Pom
`spring-boot-starter-validation`	Starter for using Java Bean Validation with Hibernate Validator	Pom
`spring-boot-starter-web`	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container	Pom
`spring-boot-starter-web-services`	Starter for using Spring Web Services	Pom
`spring-boot-starter-webflux`	Starter for building WebFlux applications using Spring Framework's Reactive Web support	Pom
`spring-boot-starter-websocket`	Starter for building WebSocket applications using Spring Framework's WebSocket support	Pom

In addition to the application starters, the following starters can be used to add *production ready* features:

Table 13.2. Spring Boot production starters

Name	Description	Pom
`spring-boot-starter-actuator`	Starter for using Spring Boot's Actuator which provides production ready features to help you monitor and manage your application	Pom

Finally, Spring Boot also includes the following starters that can be used if you want to exclude or swap specific technical facets:

Table 13.3. Spring Boot technical starters

Name	Description	Pom
`spring-boot-starter-jetty`	Starter for using Jetty as the embedded servlet container. An alternative to `spring-boot-starter-tomcat`	Pom
`spring-boot-starter-log4j2`	Starter for using Log4j2 for logging. An alternative to `spring-boot-starter-logging`	Pom
`spring-boot-starter-logging`	Starter for logging using Logback. Default logging starter	Pom
`spring-boot-starter-reactor-netty`	Starter for using Reactor Netty as the embedded reactive HTTP server.	Pom
`spring-boot-starter-tomcat`	Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by `spring-boot-starter-web`	Pom
`spring-boot-starter-undertow`	Starter for using Undertow as the embedded servlet container. An alternative to `spring-boot-starter-tomcat`	Pom

Tip

For a list of additional community contributed starters, see the README file in the `spring-boot-starters` module on GitHub.

14. Structuring Your Code

Spring Boot does not require any specific code layout to work. However, there are some best practices that help.

14.1 Using the “default” Package

When a class does not include a `package` declaration, it is considered to be in the “default package”. The use of the “default package” is generally discouraged and should be avoided. It can cause particular problems for Spring Boot applications that use the `@ComponentScan`, `@EntityScan`, or `@SpringBootApplication` annotations, since every class from every jar is read.

Tip

We recommend that you follow Java’s recommended package naming conventions and use a reversed domain name (for example, `com.example.project`).

14.2 Locating the Main Application Class

We generally recommend that you locate your main application class in a root package above other classes. The `@EnableAutoConfiguration` annotation is often placed on your main class, and it implicitly defines a base “search package” for certain items. For example, if you are writing a JPA application, the package of the `@EnableAutoConfiguration` annotated class is used to search for `@Entity` items.

Using a root package also lets the `@ComponentScan` annotation be used without needing to specify a `basePackage` attribute. You can also use the `@SpringBootApplication` annotation if your main class is in the root package.

The following listing shows a typical layout:

```
com
 +- example
     +- myapplication
         +- Application.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

The `Application.java` file would declare the `main` method, along with the basic `@Configuration`, as follows:

```java
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {

 public static void main(String[] args) {
  SpringApplication.run(Application.class, args);
 }

}
```

15. Configuration Classes

Spring Boot favors Java-based configuration. Although it is possible to use `SpringApplication` with XML sources, we generally recommend that your primary source be a single `@Configuration` class. Usually the class that defines the `main` method is a good candidate as the primary `@Configuration`.

> **Tip**
>
> Many Spring configuration examples have been published on the Internet that use XML configuration. If possible, always try to use the equivalent Java-based configuration. Searching for `Enable*` annotations can be a good starting point.

15.1 Importing Additional Configuration Classes

You need not put all your `@Configuration` into a single class. The `@Import` annotation can be used to import additional configuration classes. Alternatively, you can use `@ComponentScan` to automatically pick up all Spring components, including `@Configuration` classes.

15.2 Importing XML Configuration

If you absolutely must use XML based configuration, we recommend that you still start with a `@Configuration` class. You can then use an `@ImportResource` annotation to load XML configuration files.

16. Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. For example, if `HSQLDB` is on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database.

You need to opt-in to auto-configuration by adding the `@EnableAutoConfiguration` or `@SpringBootApplication` annotations to one of your `@Configuration` classes.

> **Tip**
>
> You should only ever add one `@EnableAutoConfiguration` annotation. We generally recommend that you add it to your primary `@Configuration` class.

16.1 Gradually Replacing Auto-configuration

Auto-configuration is non-invasive. At any point, you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own `DataSource` bean, the default embedded database support backs away.

If you need to find out what auto-configuration is currently being applied, and why, start your application with the `--debug` switch. Doing so enables debug logs for a selection of core loggers and logs a conditions report to the console.

16.2 Disabling Specific Auto-configuration Classes

If you find that specific auto-configuration classes that you do not want are being applied, you can use the exclude attribute of `@EnableAutoConfiguration` to disable them, as shown in the following example:

```
import org.springframework.boot.autoconfigure.*;
import org.springframework.boot.autoconfigure.jdbc.*;
import org.springframework.context.annotation.*;

@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
public class MyConfiguration {
}
```

If the class is not on the classpath, you can use the `excludeName` attribute of the annotation and specify the fully qualified name instead. Finally, you can also control the list of auto-configuration classes to exclude by using the `spring.autoconfigure.exclude` property.

> **Tip**
>
> You can define exclusions both at the annotation level and by using the property.

17. Spring Beans and Dependency Injection

You are free to use any of the standard Spring Framework techniques to define your beans and their injected dependencies. For simplicity, we often find that using `@ComponentScan` (to find your beans) and using `@Autowired` (to do constructor injection) works well.

If you structure your code as suggested above (locating your application class in a root package), you can add `@ComponentScan` without any arguments. All of your application components (`@Component`, `@Service`, `@Repository`, `@Controller` etc.) are automatically registered as Spring Beans.

The following example shows a `@Service` Bean that uses constructor injection to obtain a required `RiskAssessor` bean:

```
package com.example.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class DatabaseAccountService implements AccountService {

 private final RiskAssessor riskAssessor;

 @Autowired
 public DatabaseAccountService(RiskAssessor riskAssessor) {
  this.riskAssessor = riskAssessor;
 }

 // ...

}
```

If a bean has one constructor, you can omit the `@Autowired`, as shown in the following example:

```
@Service
public class DatabaseAccountService implements AccountService {

 private final RiskAssessor riskAssessor;

 public DatabaseAccountService(RiskAssessor riskAssessor) {
  this.riskAssessor = riskAssessor;
 }

 // ...

}
```

Tip

Notice how using constructor injection lets the `riskAssessor` field be marked as `final`, indicating that it cannot be subsequently changed.

18. Using the @SpringBootApplication Annotation

Many Spring Boot developers always have their main class annotated with `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan`. Since these annotations are so frequently used together (especially if you follow the best practices above), Spring Boot provides a convenient `@SpringBootApplication` alternative.

The `@SpringBootApplication` annotation is equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes, as shown in the following example:

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {

 public static void main(String[] args) {
  SpringApplication.run(Application.class, args);
 }

}
```

Note

`@SpringBootApplication` also provides aliases to customize the attributes of `@EnableAutoConfiguration` and `@ComponentScan`.

19. Running Your Application

One of the biggest advantages of packaging your application as a jar and using an embedded HTTP server is that you can run your application as you would any other. Debugging Spring Boot applications is also easy. You do not need any special IDE plugins or extensions.

Note

This section only covers jar based packaging. If you choose to package your application as a war file, you should refer to your server and IDE documentation.

19.1 Running from an IDE

You can run a Spring Boot application from your IDE as a simple Java application. However, you first need to import your project. Import steps vary depending on your IDE and build system. Most IDEs can import Maven projects directly. For example, Eclipse users can select `Import…` → `Existing Maven Projects` from the `File` menu.

If you cannot directly import your project into your IDE, you may be able to generate IDE metadata by using a build plugin. Maven includes plugins for Eclipse and IDEA. Gradle offers plugins for various IDEs.

Tip

If you accidentally run a web application twice, you see a “Port already in use” error. STS users can use the `Relaunch` button rather than the `Run` button to ensure that any existing instance is closed.

19.2 Running as a Packaged Application

If you use the Spring Boot Maven or Gradle plugins to create an executable jar, you can run your application using `java -jar`, as shown in the following example:

```
$ java -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

It is also possible to run a packaged application with remote debugging support enabled. Doing so lets you attach a debugger to your packaged application, as shown in the following example:

```
$ java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n \
       -jar target/myapplication-0.0.1-SNAPSHOT.jar
```

19.3 Using the Maven Plugin

The Spring Boot Maven plugin includes a `run` goal that can be used to quickly compile and run your application. Applications run in an exploded form, as they do in your IDE. The following example shows a typical Maven command to run a Spring Boot application:

```
$ mvn spring-boot:run
```

You might also want to use the `MAVEN_OPTS` operating system environment variable, as shown in the following example:

```
$ export MAVEN_OPTS=-Xmx1024m
```

19.4 Using the Gradle Plugin

The Spring Boot Gradle plugin also includes a `bootRun` task that can be used to run your application in an exploded form. The `bootRun` task is added whenever you apply the `org.springframework.boot` and `java` plugins and is shown in the following example:

```
$ gradle bootRun
```

You might also want to use the `JAVA_OPTS` operating system environment variable, as shown in the following example:

```
$ export JAVA_OPTS=-Xmx1024m
```

19.5 Hot Swapping

Since Spring Boot applications are just plain Java applications, JVM hot-swapping should work out of the box. JVM hot swapping is somewhat limited with the bytecode that it can replace. For a more complete solution, JRebel can be used.

The `spring-boot-devtools` module also includes support for quick application restarts. See the Chapter 20, *Developer Tools* section later in this chapter and the Hot swapping “How-to” for details.

20. Developer Tools

Spring Boot includes an additional set of tools that can make the application development experience a little more pleasant. The `spring-boot-devtools` module can be included in any project to provide additional development-time features. To include devtools support, add the module dependency to your build, as shown in the following listings for Maven and Gradle:

Maven.

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-devtools</artifactId>
  <optional>true</optional>
 </dependency>
</dependencies>
```

Gradle.

```
dependencies {
 compile("org.springframework.boot:spring-boot-devtools")
}
```

Note

Developer tools are automatically disabled when running a fully packaged application. If your application is launched from `java -jar` or if it is started from a special classloader, then it is considered a “production application”. Flagging the dependency as optional is a best practice that prevents devtools from being transitively applied to other modules that use your project. Gradle does not support `optional` dependencies out-of-the-box, so you may want to have a look at the `propdeps-plugin`.

Tip

Repackaged archives do not contain devtools by default. If you want to use a certain remote devtools feature, you need to disable the `excludeDevtools` build property to include it. The property is supported with both the Maven and Gradle plugins.

20.1 Property Defaults

Several of the libraries supported by Spring Boot use caches to improve performance. For example, template engines cache compiled templates to avoid repeatedly parsing template files. Also, Spring MVC can add HTTP caching headers to responses when serving static resources.

While caching is very beneficial in production, it can be counter-productive during development, preventing you from seeing the changes you just made in your application. For this reason, spring-boot-devtools disables the caching options by default.

Cache options are usually configured by settings in your `application.properties` file. For example, Thymeleaf offers the `spring.thymeleaf.cache` property. Rather than needing to set these properties manually, the `spring-boot-devtools` module automatically applies sensible development-time configuration.

Tip

For a complete list of the properties that are applied by the devtools, see DevToolsPropertyDefaultsPostProcessor.

20.2 Automatic Restart

Applications that use `spring-boot-devtools` automatically restart whenever files on the classpath change. This can be a useful feature when working in an IDE, as it gives a very fast feedback loop for code changes. By default, any entry on the classpath that points to a folder is monitored for changes. Note that certain resources, such as static assets and view templates, do not need to restart the application.

Triggering a restart

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using. In Eclipse, saving a modified file causes the classpath to be updated and triggers a restart. In IntelliJ IDEA, building the project (`Build -> Make Project`) has the same effect.

Note

As long as forking is enabled, you can also start your application by using the supported build plugins (Maven and Gradle), since DevTools needs an isolated application classloader to operate properly. By default, Gradle and Maven do that when they detect DevTools on the classpath.

Tip

Automatic restart works very well when used with LiveReload. See the LiveReload section for details. If you use JRebel, automatic restarts are disabled in favor of dynamic class reloading. Other devtools features (such as LiveReload and property overrides) can still be used.

Note

DevTools relies on the application context's shutdown hook to close it during a restart. It does not work correctly if you have disabled the shutdown hook (`SpringApplication.setRegisterShutdownHook(false)`).

Note

When deciding if an entry on the classpath should trigger a restart when it changes, DevTools automatically ignores projects named `spring-boot`, `spring-boot-devtools`, `spring-boot-autoconfigure`, `spring-boot-actuator`, and `spring-boot-starter`.

Note

DevTools needs to customize the `ResourceLoader` used by the `ApplicationContext`. If your application provides one already, it is going to be wrapped. Direct override of the `getResource` method on the `ApplicationContext` is not supported.

Restart vs Reload

The restart technology provided by Spring Boot works by using two classloaders. Classes that do not change (for example, those from third-party jars) are loaded into a *base* classloader. Classes that you are actively developing are loaded into a *restart* classloader. When the application is restarted, the *restart* classloader is thrown away and a new one is created. This approach means that application restarts are typically much faster than “cold starts”, since the *base* classloader is already available and populated.

If you find that restarts are not quick enough for your applications or you encounter classloading issues, you could consider reloading technologies such as JRebel from ZeroTurnaround. These work by rewriting classes as they are loaded to make them more amenable to reloading.

Logging changes in condition evaluation

By default, each time your application restarts, a report showing the condition evaluation delta is logged. The report shows the changes to your application’s auto-configuration as you make changes such as adding or removing beans and setting configuration properties.

To disable the logging of the report, set the following property:

```
spring.devtools.restart.log-condition-evaluation-delta=false
```

Excluding Resources

Certain resources do not necessarily need to trigger a restart when they are changed. For example, Thymeleaf templates can be edited in-place. By default, changing resources in `/META-INF/maven`, `/META-INF/resources`, `/resources`, `/static`, `/public`, or `/templates` does not trigger a restart but does trigger a live reload. If you want to customize these exclusions, you can use the `spring.devtools.restart.exclude` property. For example, to exclude only `/static` and `/public` you would set the following property:

```
spring.devtools.restart.exclude=static/**,public/**
```

Tip

If you want to keep those defaults and *add* additional exclusions, use the `spring.devtools.restart.additional-exclude` property instead.

Watching Additional Paths

You may want your application to be restarted or reloaded when you make changes to files that are not on the classpath. To do so, use the `spring.devtools.restart.additional-paths` property to configure additional paths to watch for changes. You can use the `spring.devtools.restart.exclude` property described earlier to control whether changes beneath the additional paths trigger a full restart or a live reload.

Disabling Restart

If you do not want to use the restart feature, you can disable it by using the `spring.devtools.restart.enabled` property. In most cases, you can set this property in your `application.properties` (doing so still initializes the restart classloader, but it does not watch for file changes).

If you need to *completely* disable restart support (for example, because it does not work with a specific library), you need to set the `spring.devtools.restart.enabled` `System` property to `false` before calling `SpringApplication.run(…)`, as shown in the following example:

```
public static void main(String[] args) {
 System.setProperty("spring.devtools.restart.enabled", "false");
 SpringApplication.run(MyApp.class, args);
}
```

Using a Trigger File

If you work with an IDE that continuously compiles changed files, you might prefer to trigger restarts only at specific times. To do so, you can use a "trigger file", which is a special file that must be modified when you want to actually trigger a restart check. Changing the file only triggers the check and the restart only occurs if Devtools has detected it has to do something. The trigger file can be updated manually or with an IDE plugin.

To use a trigger file, set the `spring.devtools.restart.trigger-file` property to the path of your trigger file.

Tip

You might want to set `spring.devtools.restart.trigger-file` as a global setting, so that all your projects behave in the same way.

Customizing the Restart Classloader

As described earlier in the Restart vs Reload section, restart functionality is implemented by using two classloaders. For most applications, this approach works well. However, it can sometimes cause classloading issues.

By default, any open project in your IDE is loaded with the "restart" classloader, and any regular `.jar` file is loaded with the "base" classloader. If you work on a multi-module project, and not every module is imported into your IDE, you may need to customize things. To do so, you can create a `META-INF/spring-devtools.properties` file.

The `spring-devtools.properties` file can contain properties prefixed with `restart.exclude` and `restart.include`. The `include` elements are items that should be pulled up into the "restart" classloader, and the `exclude` elements are items that should be pushed down into the "base" classloader. The value of the property is a regex pattern that is applied to the classpath, as shown in the following example:

```
restart.exclude.companycommonlibs=/mycorp-common-[\\w-]+\.jar
restart.include.projectcommon=/mycorp-myproj-[\\w-]+\.jar
```

Note

All property keys must be unique. As long as a property starts with `restart.include.` or `restart.exclude.` it is considered.

Tip

All `META-INF/spring-devtools.properties` from the classpath are loaded. You can package files inside your project, or in the libraries that the project consumes.

Known Limitations

Restart functionality does not work well with objects that are deserialized by using a standard `ObjectInputStream`. If you need to deserialize data, you may need to use Spring’s `ConfigurableObjectInputStream` in combination with `Thread.currentThread().getContextClassLoader()`.

Unfortunately, several third-party libraries deserialize without considering the context classloader. If you find such a problem, you need to request a fix with the original authors.

20.3 LiveReload

The `spring-boot-devtools` module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload browser extensions are freely available for Chrome, Firefox and Safari from livereload.com.

If you do not want to start the LiveReload server when your application runs, you can set the `spring.devtools.livereload.enabled` property to `false`.

Note

You can only run one LiveReload server at a time. Before starting your application, ensure that no other LiveReload servers are running. If you start multiple applications from your IDE, only the first has LiveReload support.

20.4 Global Settings

You can configure global devtools settings by adding a file named `.spring-boot-devtools.properties` to your `$HOME` folder (note that the filename starts with “.”). Any properties added to this file apply to *all* Spring Boot applications on your machine that use devtools. For example, to configure restart to always use a trigger file, you would add the following property:

~/.spring-boot-devtools.properties.

```
spring.devtools.reload.trigger-file=.reloadtrigger
```

20.5 Remote Applications

The Spring Boot developer tools are not limited to local development. You can also use several features when running applications remotely. Remote support is opt-in. To enable it, you need to make sure that `devtools` is included in the repackaged archive, as shown in the following listing:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
    <excludeDevtools>false</excludeDevtools>
   </configuration>
  </plugin>
 </plugins>
</build>
```

Then you need to set a `spring.devtools.remote.secret` property, as shown in the following example:

```
spring.devtools.remote.secret=mysecret
```

> **Warning**
>
> Enabling `spring-boot-devtools` on a remote application is a security risk. You should never enable support on a production deployment.

Remote devtools support is provided in two parts: a server-side endpoint that accepts connections and a client application that you run in your IDE. The server component is automatically enabled when the `spring.devtools.remote.secret` property is set. The client component must be launched manually.

Running the Remote Client Application

The remote client application is designed to be run from within your IDE. You need to run `org.springframework.boot.devtools.RemoteSpringApplication` with the same classpath as the remote project that you connect to. The application's single required argument is the remote URL to which it connects.

For example, if you are using Eclipse or STS and you have a project named `my-app` that you have deployed to Cloud Foundry, you would do the following:

- Select `Run Configurations…` from the `Run` menu.
- Create a new `Java Application` "launch configuration".
- Browse for the `my-app` project.
- Use `org.springframework.boot.devtools.RemoteSpringApplication` as the main class.
- Add `https://myapp.cfapps.io` to the `Program arguments` (or whatever your remote URL is).

A running remote client might resemble the following listing:

```
  .   ____          _                                              __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _          ___               _      \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` |        | _ \___ _ __  ___| |_ ___ \ \ \ \
 \\/  ___)| |_)| | | | | || (_| []::::::[]   / -_) '  \/ _ \  _/ -_) ) ) ) )
  '  |____| .__|_| |_|_| |_\__, |        |_|_\___|_|_|_\___/\__\___|/ / / /
 =========|_|==============|___/===================================/_/_/_/
 :: Spring Boot Remote :: 2.0.0.RC1

2015-06-10 18:25:06.632  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   :
 Starting RemoteSpringApplication on pwmbp with PID 14938 (/Users/pwebb/projects/spring-boot/code/
spring-boot-devtools/target/classes started by pwebb in /Users/pwebb/projects/spring-boot/code/spring-
boot-samples/spring-boot-sample-devtools)
2015-06-10 18:25:06.671  INFO 14938 --- [           main] s.c.a.AnnotationConfigApplicationContext :
 Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@2a17b7b6: startup
 date [Wed Jun 10 18:25:06 PDT 2015]; root of context hierarchy
2015-06-10 18:25:07.043  WARN 14938 --- [           main] o.s.b.d.r.c.RemoteClientConfiguration    : The
 connection to http://localhost:8080 is insecure. You should use a URL starting with 'https://'.
2015-06-10 18:25:07.074  INFO 14938 --- [           main] o.s.b.d.a.OptionalLiveReloadServer       :
 LiveReload server is running on port 35729
2015-06-10 18:25:07.130  INFO 14938 --- [           main] o.s.b.devtools.RemoteSpringApplication   :
 Started RemoteSpringApplication in 0.74 seconds (JVM running for 1.105)
```

> **Note**
>
> Because the remote client is using the same classpath as the real application it can directly read application properties. This is how the `spring.devtools.remote.secret` property is read and passed to the server for authentication.

> **Tip**
>
> It is always advisable to use `https://` as the connection protocol, so that traffic is encrypted and passwords cannot be intercepted.

> **Tip**
>
> If you need to use a proxy to access the remote application, configure the `spring.devtools.remote.proxy.host` and `spring.devtools.remote.proxy.port` properties.

Remote Update

The remote client monitors your application classpath for changes in the same way as the local restart. Any updated resource is pushed to the remote application and (*if required*) triggers a restart. This can be helpful if you iterate on a feature that uses a cloud service that you do not have locally. Generally, remote updates and restarts are much quicker than a full rebuild and deploy cycle.

> **Note**
>
> Files are only monitored when the remote client is running. If you change a file before starting the remote client, it is not pushed to the remote server.

21. Packaging Your Application for Production

Executable jars can be used for production deployment. As they are self-contained, they are also ideally suited for cloud-based deployment.

For additional “production ready” features, such as health, auditing, and metric REST or JMX end-points, consider adding `spring-boot-actuator`. See *Part V, “Spring Boot Actuator: Production-ready features”* for details.

22. What to Read Next

You should now understand how you can use Spring Boot and some best practices that you should follow. You can now go on to learn about specific *Spring Boot features* in depth, or you could skip ahead and read about the “production ready” aspects of Spring Boot.

Part IV. Spring Boot features

This section dives into the details of Spring Boot. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "Part II, “Getting Started”" and "Part III, “Using Spring Boot”" sections, so that you have a good grounding of the basics.

23. SpringApplication

The `SpringApplication` class provides a convenient way to bootstrap a Spring application that is started from a `main()` method. In many situations, you can delegate to the static `SpringApplication.run` method, as shown in the following example:

```
public static void main(String[] args) {
 SpringApplication.run(MySpringConfiguration.class, args);
}
```

When your application starts, you should see something similar to the following output:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::   v2.0.0.RC1

2013-07-31 00:08:16.117  INFO 56603 --- [           main] o.s.b.s.app.SampleApplication            :
 Starting SampleApplication v0.1.0 on mycomputer with PID 56603 (/apps/myapp.jar started by pwebb)
2013-07-31 00:08:16.166  INFO 56603 --- [           main]
 ationConfigServletWebServerApplicationContext : Refreshing
 org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@6e5a8246:
 startup date [Wed Jul 31 00:08:16 PDT 2013]; root of context hierarchy
2014-03-04 13:09:54.912  INFO 41370 --- [           main] .t.TomcatServletWebServerFactory : Server
 initialized with port: 8080
2014-03-04 13:09:56.501  INFO 41370 --- [           main] o.s.b.s.app.SampleApplication            :
 Started SampleApplication in 2.992 seconds (JVM running for 3.658)
```

By default, `INFO` logging messages are shown, including some relevant startup details, such as the user that launched the application. If you need a log level other than `INFO`, you can set it, as described in Section 26.4, “Log Levels”,

23.1 Startup Failure

If your application fails to start, registered `FailureAnalyzers` get a chance to provide a dedicated error message and a concrete action to fix the problem. For instance, if you start a web application on port `8080` and that port is already in use, you should see something similar to the following message:

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Embedded servlet container failed to start. Port 8080 was already in use.

Action:

Identify and stop the process that's listening on port 8080 or configure this application to listen on
 another port.
```

Note

Spring Boot provides numerous `FailureAnalyzer` implementations, and you can add your own.

If no failure analyzers are able to handle the exception, you can still display the full conditions report to better understand what went wrong. To do so, you need to enable the debug property or enable DEBUG logging for org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

For instance, if you are running your application by using `java -jar`, you can enable the `debug` property as follows:

```
$ java -jar myproject-0.0.1-SNAPSHOT.jar --debug
```

23.2 Customizing the Banner

The banner that is printed on start up can be changed by adding a `banner.txt` file to your classpath or by setting the `spring.banner.location` property to the location of such a file. If the file has an encoding other than UTF-8, you can set `spring.banner.charset`. In addition to a text file, you can also add a `banner.gif`, `banner.jpg`, or `banner.png` image file to your classpath or set the `spring.banner.image.location` property. Images are converted into an ASCII art representation and printed above any text banner.

Inside your `banner.txt` file, you can use any of the following placeholders:

Table 23.1. Banner variables

Variable	Description
`${application.version}`	The version number of your application, as declared in `MANIFEST.MF`. For example, `Implementation-Version: 1.0` is printed as `1.0`.
`${application.formatted-version}`	The version number of your application, as declared in `MANIFEST.MF` and formatted for display (surrounded with brackets and prefixed with `v`). For example `(v1.0)`.
`${spring-boot.version}`	The Spring Boot version that you are using. For example `2.0.0.RC1`.
`${spring-boot.formatted-version}`	The Spring Boot version that you are using, formatted for display (surrounded with brackets and prefixed with `v`). For example `(v2.0.0.RC1)`.
`${Ansi.NAME}` (or `${AnsiColor.NAME}`, `${AnsiBackground.NAME}`, `${AnsiStyle.NAME}`)	Where `NAME` is the name of an ANSI escape code. See `AnsiPropertySource` for details.
`${application.title}`	The title of your application, as declared in `MANIFEST.MF`. For example `Implementation-Title: MyApp` is printed as `MyApp`.

Tip

The `SpringApplication.setBanner(…)` method can be used if you want to generate a banner programmatically. Use the `org.springframework.boot.Banner` interface and implement your own `printBanner()` method.

You can also use the `spring.main.banner-mode` property to determine if the banner has to be printed on `System.out` (`console`), sent to the configured logger (`log`), or not produced at all (`off`).

The printed banner is registered as a singleton bean under the following name: `springBootBanner`.

Note

YAML maps `off` to `false`, so be sure to add quotes if you want to disable the banner in your application, as shown in the following example:

```
spring:
 main:
  banner-mode: "off"
```

23.3 Customizing SpringApplication

If the `SpringApplication` defaults are not to your taste, you can instead create a local instance and customize it. For example, to turn off the banner, you could write:

```
public static void main(String[] args) {
 SpringApplication app = new SpringApplication(MySpringConfiguration.class);
 app.setBannerMode(Banner.Mode.OFF);
 app.run(args);
}
```

Note

The constructor arguments passed to `SpringApplication` are configuration sources for Spring beans. In most cases, these are references to `@Configuration` classes, but they could also be references to XML configuration or to packages that should be scanned.

It is also possible to configure the `SpringApplication` by using an `application.properties` file. See *Chapter 24, Externalized Configuration* for details.

For a complete list of the configuration options, see the `SpringApplication` Javadoc.

23.4 Fluent Builder API

If you need to build an `ApplicationContext` hierarchy (multiple contexts with a parent/child relationship) or if you prefer using a “fluent” builder API, you can use the `SpringApplicationBuilder`.

The `SpringApplicationBuilder` lets you chain together multiple method calls and includes `parent` and `child` methods that let you create a hierarchy, as shown in the following example:

```
new SpringApplicationBuilder()
  .sources(Parent.class)
  .child(Application.class)
  .bannerMode(Banner.Mode.OFF)
  .run(args);
```

Note

There are some restrictions when creating an `ApplicationContext` hierarchy. For example, Web components **must** be contained within the child context, and the same `Environment` is used for both parent and child contexts. See the `SpringApplicationBuilder` Javadoc for full details.

23.5 Application Events and Listeners

In addition to the usual Spring Framework events, such as `ContextRefreshedEvent`, a `SpringApplication` sends some additional application events.

Note

Some events are actually triggered before the `ApplicationContext` is created, so you cannot register a listener on those as a `@Bean`. You can register them with the `SpringApplication.addListeners(…)` method or the `SpringApplicationBuilder.listeners(…)` method.

If you want those listeners to be registered automatically, regardless of the way the application is created, you can add a `META-INF/spring.factories` file to your project and reference your listener(s) by using the `org.springframework.context.ApplicationListener` key, as shown in the following example:

```
org.springframework.context.ApplicationListener=com.example.project.MyListener
```

Application events are sent in the following order, as your application runs:

1. An `ApplicationStartingEvent` is sent at the start of a run but before any processing, except for the registration of listeners and initializers.
2. An `ApplicationEnvironmentPreparedEvent` is sent when the `Environment` to be used in the context is known but before the context is created.
3. An `ApplicationPreparedEvent` is sent just before the refresh is started but after bean definitions have been loaded.
4. An `ApplicationStartedEvent` is sent after the context has been refreshed but before any application and command-line runners have been called.
5. An `ApplicationReadyEvent` is sent after any application and command-line runners have been called. It indicates that the application is ready to service requests.
6. An `ApplicationFailedEvent` is sent if there is an exception on startup.

Tip

You often need not use application events, but it can be handy to know that they exist. Internally, Spring Boot uses events to handle a variety of tasks.

Application events are sent by using Spring Framework's event publishing mechanism. Part of this mechanism ensures that an event published to the listeners in a child context is also published to the listeners in any ancestor contexts. As a result of this, if your application uses a hierarchy of `SpringApplication` instances, a listener may receive multiple instances of the same type of application event.

To allow your listener to distinguish between an event for its context and an event for a descendant context, it should request that its application context is injected and then compare the injected context with the context of the event. The context can be injected by implementing `ApplicationContextAware` or, if the listener is a bean, by using `@Autowired`.

23.6 Web Environment

A `SpringApplication` attempts to create the right type of `ApplicationContext` on your behalf. By default, an `AnnotationConfigApplicationContext` or `AnnotationConfigServletWebServerApplicationContext` is used, depending on whether you are developing a web application or not.

The algorithm used to determine a “web environment” is fairly simplistic (it is based on the presence of a few classes). If you need to override the default, you can use `setWebEnvironment(boolean webEnvironment)`.

It is also possible to take complete control of the `ApplicationContext` type that is used by calling `setApplicationContextClass(…)`.

Tip

It is often desirable to call `setWebEnvironment(false)` when using `SpringApplication` within a JUnit test.

23.7 Accessing Application Arguments

If you need to access the application arguments that were passed to `SpringApplication.run(…)`, you can inject a `org.springframework.boot.ApplicationArguments` bean. The `ApplicationArguments` interface provides access to both the raw `String[]` arguments as well as parsed `option` and `non-option` arguments, as shown in the following example:

```java
import org.springframework.boot.*
import org.springframework.beans.factory.annotation.*
import org.springframework.stereotype.*

@Component
public class MyBean {

 @Autowired
 public MyBean(ApplicationArguments args) {
  boolean debug = args.containsOption("debug");
  List<String> files = args.getNonOptionArgs();
  // if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
 }

}
```

Tip

Spring Boot also registers a `CommandLinePropertySource` with the Spring `Environment`. This lets you also inject single application arguments by using the `@Value` annotation.

23.8 Using the ApplicationRunner or CommandLineRunner

If you need to run some specific code once the `SpringApplication` has started, you can implement the `ApplicationRunner` or `CommandLineRunner` interfaces. Both interfaces work in the same way and offer a single `run` method, which is called just before `SpringApplication.run(…)` completes.

The CommandLineRunner interfaces provides access to application arguments as a simple string array, whereas the ApplicationRunner uses the ApplicationArguments interface discussed earlier. The following example shows a CommandLineRunner with a run method:

```
import org.springframework.boot.*
import org.springframework.stereotype.*

@Component
public class MyBean implements CommandLineRunner {

 public void run(String... args) {
  // Do something...
 }

}
```

If several CommandLineRunner or ApplicationRunner beans are defined that must be called in a specific order, you can additionally implement the org.springframework.core.Ordered interface or use the org.springframework.core.annotation.Order annotation.

23.9 Application Exit

Each SpringApplication registers a shutdown hook with the JVM to ensure that the ApplicationContext closes gracefully on exit. All the standard Spring lifecycle callbacks (such as the DisposableBean interface or the @PreDestroy annotation) can be used.

In addition, beans may implement the org.springframework.boot.ExitCodeGenerator interface if they wish to return a specific exit code when SpringApplication.exit() is called. This exit code can then be passed to System.exit() to return it as a status code, as shown in the following example:

```
@SpringBootApplication
public class ExitCodeApplication {

 @Bean
 public ExitCodeGenerator exitCodeGenerator() {
  return () -> 42;
 }

 public static void main(String[] args) {
  System.exit(SpringApplication
    .exit(SpringApplication.run(ExitCodeApplication.class, args)));
 }

}
```

Also, the ExitCodeGenerator interface may be implemented by exceptions. When such an exception is encountered, Spring Boot returns the exit code provided by the implemented getExitCode() method.

23.10 Admin Features

It is possible to enable admin-related features for the application by specifying the spring.application.admin.enabled property. This exposes the SpringApplicationAdminMXBean on the platform MBeanServer. You could use this feature to administer your Spring Boot application remotely. This feature could also be useful for any service wrapper implementation.

Tip

If you want to know on which HTTP port the application is running, get the property with a key of `local.server.port`.

Caution

Take care when enabling this feature, as the MBean exposes a method to shutdown the application.

24. Externalized Configuration

Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, and command-line arguments to externalize configuration. Property values can be injected directly into your beans by using the `@Value` annotation, accessed through Spring’s `Environment` abstraction, or be bound to structured objects through `@ConfigurationProperties`.

Spring Boot uses a very particular `PropertySource` order that is designed to allow sensible overriding of values. Properties are considered in the following order:

1. Devtools global settings properties on your home directory (`~/.spring-boot-devtools.properties` when devtools is active).
2. `@TestPropertySource` annotations on your tests.
3. `@SpringBootTest#properties` annotation attribute on your tests.
4. Command line arguments.
5. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
6. `ServletConfig` init parameters.
7. `ServletContext` init parameters.
8. JNDI attributes from `java:comp/env`.
9. Java System properties (`System.getProperties()`).
10. OS environment variables.
11. A `RandomValuePropertySource` that has properties only in `random.*`.
12. Profile-specific application properties outside of your packaged jar (`application-{profile}.properties` and YAML variants).
13. Profile-specific application properties packaged inside your jar (`application-{profile}.properties` and YAML variants).
14. Application properties outside of your packaged jar (`application.properties` and YAML variants).
15. Application properties packaged inside your jar (`application.properties` and YAML variants).
16. `@PropertySource` annotations on your `@Configuration` classes.
17. Default properties (specified by setting `SpringApplication.setDefaultProperties`).

To provide a concrete example, suppose you develop a `@Component` that uses a `name` property, as shown in the following example:

```
import org.springframework.stereotype.*
import org.springframework.beans.factory.annotation.*
```

```
@Component
public class MyBean {

    @Value("${name}")
    private String name;

    // ...

}
```

On your application classpath (for example, inside your jar) you can have an `application.properties` file that provides a sensible default property value for `name`. When running in a new environment, an `application.properties` file can be provided outside of your jar that overrides the `name`. For one-off testing, you can launch with a specific command line switch (for example, `java -jar app.jar --name="Spring"`).

Tip

The `SPRING_APPLICATION_JSON` properties can be supplied on the command line with an environment variable. For example, you could use the following line in a UN*X shell:

```
$ SPRING_APPLICATION_JSON='{"acme":{"name":"test"}}' java -jar myapp.jar
```

In the preceding example, you end up with `acme.name=test` in the Spring `Environment`. You can also supply the JSON as `spring.application.json` in a System property, as shown in the following example:

```
$ java -Dspring.application.json='{"name":"test"}' -jar myapp.jar
```

You can also supply the JSON by using a command line argument, as shown in the following example:

```
$ java -jar myapp.jar --spring.application.json='{"name":"test"}'
```

You can also supply the JSON as a JNDI variable, as follows: `java:comp/env/spring.application.json`.

24.1 Configuring Random Values

The `RandomValuePropertySource` is useful for injecting random values (for example, into secrets or test cases). It can produce integers, longs, uuids, or strings, as shown in the following example:

```
my.secret=${random.value}
my.number=${random.int}
my.bignumber=${random.long}
my.uuid=${random.uuid}
my.number.less.than.ten=${random.int(10)}
my.number.in.range=${random.int[1024,65536]}
```

The `random.int*` syntax is `OPEN value (,max) CLOSE` where the `OPEN,CLOSE` are any character and `value,max` are integers. If `max` is provided, then `value` is the minimum value and `max` is the maximum value (exclusive).

24.2 Accessing Command Line Properties

By default, `SpringApplication` converts any command line option arguments (that is, arguments starting with `--`, such as `--server.port=9000`) to a `property` and adds them to the Spring

`Environment`. As mentioned previously, command line properties always take precedence over other property sources.

If you do not want command line properties to be added to the `Environment`, you can disable them by using `SpringApplication.setAddCommandLineProperties(false)`.

24.3 Application Property Files

`SpringApplication` loads properties from `application.properties` files in the following locations and adds them to the Spring `Environment`:

1. A `/config` subdirectory of the current directory
2. The current directory
3. A classpath `/config` package
4. The classpath root

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations).

> **Note**
>
> You can also use YAML ('.yml') files as an alternative to '.properties'.

If you do not like `application.properties` as the configuration file name, you can switch to another file name by specifying a `spring.config.name` environment property. You can also refer to an explicit location by using the `spring.config.location` environment property (which is a comma-separated list of directory locations or file paths). The following example shows how to specify a different file name:

```
$ java -jar myproject.jar --spring.config.name=myproject
```

The following example shows how to specify two locations:

```
$ java -jar myproject.jar --spring.config.location=classpath:/default.properties,classpath:/
override.properties
```

> **Warning**
>
> `spring.config.name` and `spring.config.location` are used very early to determine which files have to be loaded, so they must be defined as an environment property (typically an OS environment variable, a system property, or a command-line argument).

If `spring.config.location` contains directories (as opposed to files), they should end in `/` (and, at runtime, be appended with the names generated from `spring.config.name` before being loaded, including profile-specific file names). Files specified in `spring.config.location` are used as-is, with no support for profile-specific variants, and are overridden by any profile-specific properties.

Config locations are searched in reverse order. By default, the configured locations are `classpath:/,classpath:/config/,file:./,file:./config/`. The resulting search order is the following:

1. `file:./config/`

2. `file:./`
3. `classpath:/config/`
4. `classpath:/`

When custom config locations are configured by using `spring.config.location`, they replace the default locations. For example, if `spring.config.location` is configured with the value `classpath:/custom-config/,file:./custom-config/`, the search order becomes the following:

1. `file:./custom-config/`
2. `classpath:custom-config/`

Alternatively, when custom config locations are configured by using `spring.config.additional-location`, they are used in addition to the default locations. Additional locations are searched before the default locations. For example, if additional locations of `classpath:/custom-config/,file:./custom-config/` are configured, the search order becomes the following:

1. `file:./custom-config/`
2. `classpath:custom-config/`
3. `file:./config/`
4. `file:./`
5. `classpath:/config/`
6. `classpath:/`

This search ordering lets you specify default values in one configuration file and then selectively override those values in another. You can provide default values for your application in `application.properties` (or whatever other basename you choose with `spring.config.name`) in one of the default locations. These default values can then be overridden at runtime with a different file located in one of the custom locations.

> **Note**
>
> If you use environment variables rather than system properties, most operating systems disallow period-separated key names, but you can use underscores instead (for example, `SPRING_CONFIG_NAME` instead of `spring.config.name`).

> **Note**
>
> If your application runs in a container, then JNDI properties (in `java:comp/env`) or servlet context initialization parameters can be used instead of, or as well as, environment variables or system properties.

24.4 Profile-specific Properties

In addition to `application.properties` files, profile-specific properties can also be defined by using the following naming convention: `application-{profile}.properties`. The `Environment` has a set of default profiles (by default, `[default]`) that are used if no active profiles are set. In other words, if no profiles are explicitly activated, then properties from `application-default.properties` are loaded.

Profile-specific properties are loaded from the same locations as standard `application.properties`, with profile-specific files always overriding the non-specific ones, whether or not the profile-specific files are inside or outside your packaged jar.

If several profiles are specified, a last-wins strategy applies. For example, profiles specified by the `spring.profiles.active` property are added after those configured through the `SpringApplication` API and therefore take precedence.

> **Note**
>
> If you have specified any files in `spring.config.location`, profile-specific variants of those files are not considered. Use directories in `spring.config.location` if you want to also use profile-specific properties.

24.5 Placeholders in Properties

The values in `application.properties` are filtered through the existing `Environment` when they are used, so you can refer back to previously defined values (for example, from System properties).

```
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

> **Tip**
>
> You can also use this technique to create “short” variants of existing Spring Boot properties. See the *Section 73.4, “Use ‘Short’ Command Line Arguments”* how-to for details.

24.6 Using YAML Instead of Properties

YAML is a superset of JSON and, as such, is a convenient format for specifying hierarchical configuration data. The `SpringApplication` class automatically supports YAML as an alternative to properties whenever you have the SnakeYAML library on your classpath.

> **Note**
>
> If you use “Starters”, SnakeYAML is automatically provided by `spring-boot-starter`.

Loading YAML

Spring Framework provides two convenient classes that can be used to load YAML documents. The `YamlPropertiesFactoryBean` loads YAML as `Properties` and the `YamlMapFactoryBean` loads YAML as a `Map`.

For example, consider the following YAML document:

```
environments:
 dev:
  url: http://dev.example.com
  name: Developer Setup
 prod:
  url: http://another.example.com
  name: My Cool App
```

The preceding example would be transformed into the following properties:

```
environments.dev.url=http://dev.example.com
environments.dev.name=Developer Setup
environments.prod.url=http://another.example.com
environments.prod.name=My Cool App
```

YAML lists are represented as property keys with `[index]` dereferencers. For example, consider the following YAML:

```
my:
servers:
 - dev.example.com
 - another.example.com
```

The preceding example would be transformed into these properties:

```
my.servers[0]=dev.example.com
my.servers[1]=another.example.com
```

To bind to properties like that by using the Spring `DataBinder` utilities (which is what `@ConfigurationProperties` does), you need to have a property in the target bean of type `java.util.List` (or `Set`) and you either need to provide a setter or initialize it with a mutable value. For example, the following example binds to the properties shown previously:

```
@ConfigurationProperties(prefix="my")
public class Config {

 private List<String> servers = new ArrayList<String>();

 public List<String> getServers() {
  return this.servers;
 }
}
```

Note

When lists are configured in more than one place, overriding works by replacing the entire list. In the preceding example, when `my.servers` is defined in several places, the entire list from the `PropertySource` with higher precedence overrides any other configuration for that list. Both comma-separated lists and YAML lists can be used for completely overriding the contents of the list.

Exposing YAML as Properties in the Spring Environment

The `YamlPropertySourceLoader` class can be used to expose YAML as a `PropertySource` in the Spring `Environment`. Doing so lets you use the `@Value` annotation with placeholders syntax to access YAML properties.

Multi-profile YAML Documents

You can specify multiple profile-specific YAML documents in a single file by using a `spring.profiles` key to indicate when the document applies, as shown in the following example:

```
server:
 address: 192.168.1.100
---
spring:
 profiles: development
server:
 address: 127.0.0.1
---
spring:
 profiles: production
server:
 address: 192.168.1.120
```

In the preceding example, if the `development` profile is active, the `server.address` property is `127.0.0.1`. Similarly, if the `production` profile is active, the `server.address` property is `192.168.1.120`. If the `development` and `production` profiles are **not** enabled, then the value for the property is `192.168.1.100`.

If none are explicitly active when the application context starts, the default profiles are activated. So, in the following YAML, we set a value for `spring.security.user.password` that is available **only** in the "default" profile:

```
server:
  port: 8000
---
spring:
  profiles: default
  security:
    user:
      password: weak
```

Whereas, in the following example, the password is always set because it is not attached to any profile, and it would have to be explicitly reset in all other profiles as necessary:

```
server:
  port: 8000
spring:
  security:
    user:
      password: weak
```

Spring profiles designated by using the `spring.profiles` element may optionally be negated by using the `!` character. If both negated and non-negated profiles are specified for a single document, at least one non-negated profile must match, and no negated profiles may match.

YAML Shortcomings

YAML files cannot be loaded by using the `@PropertySource` annotation. So, in the case that you need to load values that way, you need to use a properties file.

Merging YAML Lists

As we showed earlier, any YAML content is ultimately transformed to properties. That process may be counter-intuitive when overriding “list” properties through a profile.

For example, assume a `MyPojo` object with `name` and `description` attributes that are `null` by default. The following example exposes a list of `MyPojo` objects from `AcmeProperties`:

```
@ConfigurationProperties("acme")
public class AcmeProperties {
 private final List<MyPojo> list = new ArrayList<>();

 public List<MyPojo> getList() {
  return this.list;
 }

}
```

Consider the following configuration:

```
acme:
  list:
    - name: my name
      description: my description
---
spring:
  profiles: dev
acme:
  list:
       - name: my another name
```

If the `dev` profile is not active, `AcmeProperties.list` contains one `MyPojo` entry, as previously defined. If the `dev` profile is enabled, however, the `list` *still* contains only one entry (with a name of `my another name` and a description of `null`). This configuration *does not* add a second `MyPojo` instance to the list, and it does not merge the items.

When a collection is specified in multiple profiles, the one with the highest priority (and only that one) is used. Consider the following example:

```
acme:
  list:
 - name: my name
   description: my description
 - name: another name
   description: another description
---
spring:
  profiles: dev
acme:
  list:
  - name: my another name
```

In the preceding example, if the `dev` profile is active, `AcmeProperties.list` contains *one* `MyPojo` entry (with a name of `my another name` and a description of `null`).

24.7 Type-safe Configuration Properties

Using the `@Value("${property}")` annotation to inject configuration properties can sometimes be cumbersome, especially if you are working with multiple properties or your data is hierarchical in nature. Spring Boot provides an alternative method of working with properties that lets strongly typed beans govern and validate the configuration of your application, as shown in the following example:

```
package com.example;

import java.net.InetAddress;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.springframework.boot.context.properties.ConfigurationProperties;

@ConfigurationProperties("acme")
```

```
public class AcmeProperties {

 private boolean enabled;

 private InetAddress remoteAddress;

 private final Security security = new Security();

 public boolean isEnabled() { ... }

 public void setEnabled(boolean enabled) { ... }

 public InetAddress getRemoteAddress() { ... }

 public void setRemoteAddress(InetAddress remoteAddress) { ... }

 public Security getSecurity() { ... }

 public static class Security {

  private String username;

  private String password;

  private List<String> roles = new ArrayList<>(Collections.singleton("USER"));

  public String getUsername() { ... }

  public void setUsername(String username) { ... }

  public String getPassword() { ... }

  public void setPassword(String password) { ... }

  public List<String> getRoles() { ... }

  public void setRoles(List<String> roles) { ... }

 }
}
```

The preceding POJO defines the following properties:

- `acme.enabled`, with a value of `false` by default.
- `acme.remote-address`, with a type that can be coerced from `String`.
- `acme.security.username`, with a nested "security" object whose name is determined by the name of the property. In particular, the return type is not used at all there and could have been `SecurityProperties`.
- `acme.security.password`.
- `acme.security.roles`, with a collection of `String`.

Note

Getters and setters are usually mandatory, since binding is through standard Java Beans property descriptors, just like in Spring MVC. A setter may be omitted in the following cases:

- Maps, as long as they are initialized, need a getter but not necessarily a setter, since they can be mutated by the binder.
- Collections and arrays can be accessed either through an index (typically with YAML) or by using a single comma-separated value (properties). In the latter case, a setter is mandatory.

We recommend to always add a setter for such types. If you initialize a collection, make sure it is not immutable (as in the preceding example).

- If nested POJO properties are initialized (like the `Security` field in the preceding example), a setter is not required. If you want the binder to create the instance on the fly by using its default constructor, you need a setter.

Some people use Project Lombok to add getters and setters automatically. Make sure that Lombok does not generate any particular constructor for such a type, as it is used automatically by the container to instantiate the object.

Tip

See also the differences between `@Value` and `@ConfigurationProperties`.

You also need to list the properties classes to register in the `@EnableConfigurationProperties` annotation, as shown in the following example:

```
@Configuration
@EnableConfigurationProperties(AcmeProperties.class)
public class MyConfiguration {
}
```

Note

When the `@ConfigurationProperties` bean is registered that way, the bean has a conventional name: `<prefix>-<fqn>`, where `<prefix>` is the environment key prefix specified in the `@ConfigurationProperties` annotation and `<fqn>` is the fully qualified name of the bean. If the annotation does not provide any prefix, only the fully qualified name of the bean is used.

The bean name in the example above is `acme-com.example.AcmeProperties`.

Even if the preceding configuration creates a regular bean for `AcmeProperties`, we recommend that `@ConfigurationProperties` only deal with the environment and, in particular, does not inject other beans from the context. Having said that, the `@EnableConfigurationProperties` annotation is *also* automatically applied to your project so that any *existing* bean annotated with `@ConfigurationProperties` is configured from the `Environment`. You could shortcut `MyConfiguration` by making sure `AcmeProperties` is already a bean, as shown in the following example:

```
@Component
@ConfigurationProperties(prefix="acme")
public class AcmeProperties {

 // ... see the preceding example

}
```

This style of configuration works particularly well with the `SpringApplication` external YAML configuration, as shown in the following example:

```
# application.yml

acme:
 remote-address: 192.168.1.1
 security:
  username: admin
  roles:
    - USER
    - ADMIN

# additional configuration as required
```

To work with `@ConfigurationProperties` beans, you can inject them in the same way as any other bean, as shown in the following example:

```
@Service
public class MyService {

 private final AcmeProperties properties;

 @Autowired
 public MyService(AcmeProperties properties) {
     this.properties = properties;
 }

  //...

 @PostConstruct
 public void openConnection() {
  Server server = new Server(this.properties.getRemoteAddress());
  // ...
 }

}
```

Tip

Using `@ConfigurationProperties` also lets you generate metadata files that can be used by IDEs to offer auto-completion for your own keys. See the Appendix B, *Configuration Metadata* appendix for details.

Third-party Configuration

As well as using `@ConfigurationProperties` to annotate a class, you can also use it on public `@Bean` methods. Doing so can be particularly useful when you want to bind properties to third-party components that are outside of your control.

To configure a bean from the `Environment` properties, add `@ConfigurationProperties` to its bean registration, as shown in the following example:

```
@ConfigurationProperties(prefix = "another")
@Bean
public AnotherComponent anotherComponent() {
 ...
}
```

Any property defined with the `another` prefix is mapped onto that `AnotherComponent` bean in manner similar to the preceding `AcmeProperties` example.

Relaxed Binding

Spring Boot uses some relaxed rules for binding `Environment` properties to `@ConfigurationProperties` beans, so there does not need to be an exact match between the `Environment` property name and the bean property name. Common examples where this is useful include dash-separated environment properties (for example, `context-path` binds to `contextPath`), and capitalized environment properties (for example, `PORT` binds to `port`).

For example, consider the following `@ConfigurationProperties` class:

```
@ConfigurationProperties(prefix="acme.my-project.person")
public class OwnerProperties {

 private String firstName;

 public String getFirstName() {
  return this.firstName;
 }

 public void setFirstName(String firstName) {
  this.firstName = firstName;
 }

}
```

In the preceding example, the following properties names can all be used:

Table 24.1. relaxed binding

Property	Note
`acme.my-project.person.firstName`	Standard camel case syntax.
`acme.my-project.person.first-name`	Kebab case, which is recommended for use in `.properties` and `.yml` files.
`acme.my-project.person.first_name`	Underscore notation, which is an alternative format for use in `.properties` and `.yml` files.
`ACME_MYPROJECT_PERSON_FIRSTNAME`	Upper case format, which is recommended when using system environment variables.

Note

The `prefix` value for the annotation *must* be in kebab case (lowercase and separated by `-`, such as `acme.my-project.person`).

Table 24.2. relaxed binding rules per property source

Property Source	Simple	List
Properties Files	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values
YAML Files	Camel case, kebab case, or underscore notation	Standard YAML list syntax or comma-separated values
Environment Variables	Upper case format with underscore as the delimiter. _ should not be used within a property name	Numeric values surrounded by underscores, such as `MY_ACME_1_OTHER = my.acme[1].other`

Property Source	Simple	List
System properties	Camel case, kebab case, or underscore notation	Standard list syntax using `[ ]` or comma-separated values

Tip

We recommend that, when possible, properties are stored in lower-case kebab format, such as `my.property-name=acme`.

Properties Conversion

Spring attempts to coerce the external application properties to the right type when it binds to the `@ConfigurationProperties` beans. If you need custom type conversion, you can provide a `ConversionService` bean (with a bean named `conversionService`) or custom property editors (through a `CustomEditorConfigurer` bean) or custom `Converters` (with bean definitions annotated as `@ConfigurationPropertiesBinding`).

Note

As this bean is requested very early during the application lifecycle, make sure to limit the dependencies that your `ConversionService` is using. Typically, any dependency that you require may not be fully initialized at creation time. You may want to rename your custom `ConversionService` if it is not required for configuration keys coercion and only rely on custom converters qualified with `@ConfigurationPropertiesBinding`.

@ConfigurationProperties Validation

Spring Boot attempts to validate `@ConfigurationProperties` classes whenever they are annotated with Spring's `@Validated` annotation. You can use JSR-303 `javax.validation` constraint annotations directly on your configuration class. To do so, ensure that a compliant JSR-303 implementation is on your classpath and then add constraint annotations to your fields, as shown in the following example:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

 @NotNull
 private InetAddress remoteAddress;

 // ... getters and setters

}
```

In order to validate the values of nested properties, you must annotate the associated field as `@Valid` to trigger its validation. The following example builds on the preceding `AcmeProperties` example:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties {

 @NotNull
 private InetAddress remoteAddress;

 @Valid
```

```
    private final Security security = new Security();

    // ... getters and setters

    public static class Security {

        @NotEmpty
        public String username;

        // ... getters and setters

    }

}
```

You can also add a custom Spring `Validator` by creating a bean definition called `configurationPropertiesValidator`. The `@Bean` method should be declared `static`. The configuration properties validator is created very early in the application's lifecycle, and declaring the `@Bean` method as static lets the bean be created without having to instantiate the `@Configuration` class. Doing so avoids any problems that may be caused by early instantiation. There is a property validation sample that shows how to set things up.

> **Tip**
>
> The `spring-boot-actuator` module includes an endpoint that exposes all `@ConfigurationProperties` beans. Point your web browser to `/actuator/configprops` or use the equivalent JMX endpoint. See the "Production ready features" section for details.

@ConfigurationProperties vs. @Value

The `@Value` annotation is a core container feature, and it does not provide the same features as type-safe configuration properties. The following table summarizes the features that are supported by `@ConfigurationProperties` and `@Value`:

Feature	`@ConfigurationProperties`	`@Value`
Relaxed binding	Yes	No
Meta-data support	Yes	No
SpEL evaluation	No	Yes

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. You should also be aware that, since `@Value` does not support relaxed binding, it is not a good candidate if you need to provide the value by using environment variables.

Finally, while you can write a `SpEL` expression in `@Value`, such expressions are not processed from application property files.

25. Profiles

Spring Profiles provide a way to segregate parts of your application configuration and make it be available only in certain environments. Any `@Component` or `@Configuration` can be marked with `@Profile` to limit when it is loaded, as shown in the following example:

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

 // ...

}
```

You can use a `spring.profiles.active` `Environment` property to specify which profiles are active. You can specify the property in any of the ways described earlier in this chapter. For example, you could include it in your `application.properties`, as shown in the following example:

```
spring.profiles.active=dev,hsqldb
```

You could also specify it on the command line by using the following switch: `--spring.profiles.active=dev,hsqldb`.

25.1 Adding Active Profiles

The `spring.profiles.active` property follows the same ordering rules as other properties: The highest `PropertySource` wins. This means that you can specify active profiles in `application.properties` and then **replace** them by using the command line switch.

Sometimes, it is useful to have profile-specific properties that **add** to the active profiles rather than replace them. The `spring.profiles.include` property can be used to unconditionally add active profiles. The `SpringApplication` entry point also has a Java API for setting additional profiles (that is, on top of those activated by the `spring.profiles.active` property). See the `setAdditionalProfiles()` method in SpringApplication.

For example, when an application with the following properties is run by using the switch, `--spring.profiles.active=prod`, the `proddb` and `prodmq` profiles are also activated:

```
---
my.property: fromyamlfile
---
spring.profiles: prod
spring.profiles.include:
  - proddb
  - prodmq
```

Note

Remember that the `spring.profiles` property can be defined in a YAML document to determine when this particular document is included in the configuration. See Section 73.7, “Change Configuration Depending on the Environment” for more details.

25.2 Programmatically Setting Profiles

You can programmatically set active profiles by calling `SpringApplication.setAdditionalProfiles(…)` before your application runs. It is also possible to activate profiles by using Spring's `ConfigurableEnvironment` interface.

25.3 Profile-specific Configuration Files

Profile-specific variants of both `application.properties` (or `application.yml`) and files referenced through `@ConfigurationProperties` are considered as files and loaded. See "Section 24.4, “Profile-specific Properties”" for details.

26. Logging

Spring Boot uses Commons Logging for all internal logging but leaves the underlying log implementation open. Default configurations are provided for Java Util Logging, Log4J2, and Logback. In each case, loggers are pre-configured to use console output with optional file output also available.

By default, if you use the "Starters", Logback is used for logging. Appropriate Logback routing is also included to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.

Tip

There are a lot of logging frameworks available for Java. Do not worry if the above list seems confusing. Generally, you do not need to change your logging dependencies and the Spring Boot defaults work just fine.

26.1 Log Format

The default log output from Spring Boot resembles the following example:

```
2014-03-05 10:57:51.112  INFO 45469 --- [           main] org.apache.catalina.core.StandardEngine  :
 Starting Servlet Engine: Apache Tomcat/7.0.52
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       :
 Initializing Spring embedded WebApplicationContext
2014-03-05 10:57:51.253  INFO 45469 --- [ost-startStop-1] o.s.web.context.ContextLoader            :
 Root WebApplicationContext: initialization completed in 1358 ms
2014-03-05 10:57:51.698  INFO 45469 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean        :
 Mapping servlet: 'dispatcherServlet' to [/]
2014-03-05 10:57:51.702  INFO 45469 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean  :
 Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
```

The following items are output:

- Date and Time: Millisecond precision and easily sortable.
- Log Level: `ERROR`, `WARN`, `INFO`, `DEBUG`, or `TRACE`.
- Process ID.
- A `---` separator to distinguish the start of actual log messages.
- Thread name: Enclosed in square brackets (may be truncated for console output).
- Logger name: This is usually the source class name (often abbreviated).
- The log message.

Note

Logback does not have a `FATAL` level. It is mapped to `ERROR`.

26.2 Console Output

The default log configuration echoes messages to the console as they are written. By default, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged. You can also enable a "debug" mode by starting your application with a `--debug` flag.

```
$ java -jar myapp.jar --debug
```

> **Note**
>
> You can also specify `debug=true` in your `application.properties`.

When the debug mode is enabled, a selection of core loggers (embedded container, Hibernate, and Spring Boot) are configured to output more information. Enabling the debug mode does *not* configure your application to log all messages with `DEBUG` level.

Alternatively, you can enable a “trace” mode by starting your application with a `--trace` flag (or `trace=true` in your `application.properties`). Doing so enables trace logging for a selection of core loggers (embedded container, Hibernate schema generation, and the whole Spring portfolio).

Color-coded Output

If your terminal supports ANSI, color output is used to aid readability. You can set `spring.output.ansi.enabled` to a supported value to override the auto detection.

Color coding is configured by using the `%clr` conversion word. In its simplest form, the converter colors the output according to the log level, as shown in the following example:

```
%clr(%5p)
```

The following table describes the mapping of log levels to colors:

Level	Color
`FATAL`	Red
`ERROR`	Red
`WARN`	Yellow
`INFO`	Green
`DEBUG`	Green
`TRACE`	Green

Alternatively, you can specify the color or style that should be used by providing it as an option to the conversion. For example, to make the text yellow, use the following setting:

```
%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){yellow}
```

The following colors and styles are supported:

- `blue`
- `cyan`
- `faint`
- `green`

- `magenta`
- `red`
- `yellow`

26.3 File Output

By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, you need to set a `logging.file` or `logging.path` property (for example, in your `application.properties`).

The following table shows how the `logging.*` properties can be used together:

Table 26.1. Logging properties

`logging.file`	`logging.path`	Example	Description
(none)	*(none)*		Console only logging.
Specific file	*(none)*	`my.log`	Writes to the specified log file. Names can be an exact location or relative to the current directory.
(none)	Specific directory	`/var/log`	Writes `spring.log` to the specified directory. Names can be an exact location or relative to the current directory.

Log files rotate when they reach 10 MB and, as with console output, `ERROR`-level, `WARN`-level, and `INFO`-level messages are logged by default. Size limits can be changed using the `logging.file.max-size` property. Previously rotated files are archived indefinitely unless the `logging.file.max-history` property has been set.

Note

The logging system is initialized early in the application lifecycle. Consequently, logging properties are not found in property files loaded through `@PropertySource` annotations.

Tip

Logging properties are independent of the actual logging infrastructure. As a result, specific configuration keys (such as `logback.configurationFile` for Logback) are not managed by spring Boot.

26.4 Log Levels

All the supported logging systems can have the logger levels set in the Spring `Environment` (for example, in `application.properties`) by using `logging.level.*=LEVEL` where `LEVEL` is one of TRACE, DEBUG, INFO, WARN, ERROR, FATAL, or OFF. The `root` logger can be configured by using `logging.level.root`. The following example shows potential logging settings in `application.properties`:

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

26.5 Custom Log Configuration

The various logging systems can be activated by including the appropriate libraries on the classpath and can be further customized by providing a suitable configuration file in the root of the classpath or in a location specified by the following Spring `Environment` property: `logging.config`.

You can force Spring Boot to use a particular logging system by using the `org.springframework.boot.logging.LoggingSystem` system property. The value should be the fully qualified class name of a `LoggingSystem` implementation. You can also disable Spring Boot's logging configuration entirely by using a value of `none`.

Note

Since logging is initialized **before** the `ApplicationContext` is created, it is not possible to control logging from `@PropertySources` in Spring `@Configuration` files. System properties and the conventional Spring Boot external configuration files work fine.)

Depending on your logging system, the following files are loaded:

Logging System	Customization
Logback	`logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, or `logback.groovy`
Log4j2	`log4j2-spring.xml` or `log4j2.xml`
JDK (Java Util Logging)	`logging.properties`

Note

When possible, we recommend that you use the `-spring` variants for your logging configuration (for example, `logback-spring.xml` rather than `logback.xml`). If you use standard configuration locations, Spring cannot completely control log initialization.

Warning

There are known classloading issues with Java Util Logging that cause problems when running from an 'executable jar'. We recommend that you avoid it when running from an 'executable jar' if at all possible.

To help with the customization, some other properties are transferred from the Spring `Environment` to System properties, as described in the following table:

Spring Environment	System Property	Comments
`logging.exception-conversion-word`	`LOG_EXCEPTION_CONVERSION_WORD`	The conversion word used when logging exceptions.
`logging.file`	`LOG_FILE`	If defined, it is used in the default log configuration.

Spring Environment	System Property	Comments
`logging.file.max-size`	`LOG_FILE_MAX_SIZE`	Maximum log file size (if LOG_FILE enabled). (Only supported with the default Logback setup.)
`logging.file.max-history`	`LOG_FILE_MAX_HISTORY`	Maximum number of archive log files to keep (if LOG_FILE enabled). (Only supported with the default Logback setup.)
`logging.path`	`LOG_PATH`	If defined, it is used in the default log configuration.
`logging.pattern.console`	`CONSOLE_LOG_PATTERN`	The log pattern to use on the console (stdout). (Only supported with the default Logback setup.)
`logging.pattern.dateformat`	`LOG_DATEFORMAT_PATTERN`	Appender pattern for log date format. (Only supported with the default Logback setup.)
`logging.pattern.file`	`FILE_LOG_PATTERN`	The log pattern to use in a file (if `LOG_FILE` is enabled). (Only supported with the default Logback setup.)
`logging.pattern.level`	`LOG_LEVEL_PATTERN`	The format to use when rendering the log level (default `%5p`). (Only supported with the default Logback setup.)
`PID`	`PID`	The current process ID (discovered if possible and when not already defined as an OS environment variable).

All the supported logging systems can consult System properties when parsing their configuration files. See the default configurations in `spring-boot.jar` for examples:

- Logback
- Log4j 2
- Java Util logging

Tip

If you want to use a placeholder in a logging property, you should use Spring Boot's syntax and not the syntax of the underlying framework. Notably, if you use Logback, you should use `:` as the delimiter between a property name and its default value and not use `:-`.

Tip

You can add MDC and other ad-hoc content to log lines by overriding only the `LOG_LEVEL_PATTERN` (or `logging.pattern.level` with Logback). For example, if you use `logging.pattern.level=user:%X{user} %5p`, then the default log format contains an MDC entry for "user", if it exists, as shown in the following example.

```
2015-09-30 12:30:04.031 user:someone INFO 22174 --- [  nio-8080-exec-0] demo.Controller
Handling authenticated request
```

26.6 Logback Extensions

Spring Boot includes a number of extensions to Logback that can help with advanced configuration. You can use these extensions in your `logback-spring.xml` configuration file.

Note

Because the standard `logback.xml` configuration file is loaded too early, you cannot use extensions in it. You need to either use `logback-spring.xml` or define a `logging.config` property.

Warning

The extensions cannot be used with Logback's configuration scanning. If you attempt to do so, making changes to the configuration file results in an error similar to one of the following being logged:

```
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProperty],
 current ElementPath is [[configuration][springProperty]]
ERROR in ch.qos.logback.core.joran.spi.Interpreter@4:71 - no applicable action for [springProfile],
 current ElementPath is [[configuration][springProfile]]
```

Profile-specific Configuration

The `<springProfile>` tag lets you optionally include or exclude sections of configuration based on the active Spring profiles. Profile sections are supported anywhere within the `<configuration>` element. Use the `name` attribute to specify which profile accepts the configuration. Multiple profiles can be specified with a comma-separated list. The following listing shows three sample profiles:

```
<springProfile name="staging">
 <!-- configuration to be enabled when the "staging" profile is active -->
</springProfile>

<springProfile name="dev, staging">
 <!-- configuration to be enabled when the "dev" or "staging" profiles are active -->
</springProfile>

<springProfile name="!production">
 <!-- configuration to be enabled when the "production" profile is not active -->
</springProfile>
```

Environment Properties

The `<springProperty>` tag lets you expose properties from the Spring `Environment` for use within Logback. Doing so can be useful if you want to access values from your `application.properties` file in your Logback configuration. The tag works in a similar way to Logback's standard `<property>`

tag. However, rather than specifying a direct `value`, you specify the `source` of the property (from the `Environment`). If you need to store the property somewhere other than in `local` scope, you can use the `scope` attribute. If you need a fallback value (in case the property is not set in the `Environment`), you can use the `defaultValue` attribute. The following example shows how to expose properties for use within Logback:

```
<springProperty scope="context" name="fluentHost" source="myapp.fluentd.host"
  defaultValue="localhost"/>
<appender name="FLUENT" class="ch.qos.logback.more.appenders.DataFluentAppender">
 <remoteHost>${fluentHost}</remoteHost>
 ...
</appender>
```

Note

The `source` must be specified in kebab case (such as `my.property-name`). However, properties can be added to the `Environment` by using the relaxed rules.

27. Developing Web Applications

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the `spring-boot-starter-web` module to get up and running quickly. You can also choose to build reactive web applications by using the `spring-boot-starter-webflux` module.

If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the *Getting started* section.

27.1 The “Spring Web MVC Framework”

The Spring Web MVC framework (often referred to as simply “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

 @RequestMapping(value="/{user}", method=RequestMethod.GET)
 public User getUser(@PathVariable Long user) {
  // ...
 }

 @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
 List<Customer> getUserCustomers(@PathVariable Long user) {
  // ...
 }

 @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
 public User deleteUser(@PathVariable Long user) {
  // ...
 }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides.

Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring’s defaults:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document)).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).

- Static `index.html` support.
- Custom `Favicon` support (covered later in this document).
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep Spring Boot MVC features and you want to add additional MVC configuration (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

```
import org.springframework.boot.autoconfigure.web.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

 @Bean
 public HttpMessageConverters customConverters() {
  HttpMessageConverter<?> additional = ...
  HttpMessageConverter<?> another = ...
  return new HttpMessageConverters(additional, another);
 }

}
```

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

Custom JSON Serializers and Deserializers

If you use Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually registered with Jackson through a module, but Spring Boot provides an alternative `@JsonComponent` annotation that makes it easier to directly register Spring Beans.

You can use the `@JsonComponent` annotation directly on `JsonSerializer` or `JsonDeserializer` implementations. You can also use it on classes that contain serializers/deserializers as inner classes, as shown in the following example:

```
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

 public static class Serializer extends JsonSerializer<SomeObject> {
  // ...
 }

 public static class Deserializer extends JsonDeserializer<SomeObject> {
  // ...
 }

}
```

All `@JsonComponent` beans in the `ApplicationContext` are automatically registered with Jackson. Because `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes that provide useful alternatives to the standard Jackson versions when serializing objects. See `JsonObjectSerializer` and `JsonObjectDeserializer` in the Javadoc for details.

MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver.format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is also enabled and acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

```
spring.mvc.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using the `spring.resources.static-locations` property (replacing the default values with a list of directory locations). The root Servlet context path, `"/"`, is automatically added as a location as well.

In addition to the “standard” static resource locations mentioned earlier, a special case is made for Webjars content. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

Tip

Do not use the `src/main/webapp` directory if your application is packaged as a jar. Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `webjars-locator` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/dist/jquery.min.js"` results in `"/webjars/jquery/x.y.z/dist/jquery.min.js"`. where `x.y.z` is the Webjar version.

Note

If you use JBoss, you need to declare the `webjars-locator-jboss-vfs` dependency instead of the `webjars-locator`. Otherwise, all Webjars resolve as a `404`.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`, in URLs:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
```

Note

Links to resources are rewritten in templates at runtime, thanks to a `ResourceUrlEncodingFilter` that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the `ResourceUrlProvider`.

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/js/lib/
spring.resources.chain.strategy.fixed.version=v12
```

With this configuration, JavaScript modules located under `"/js/lib/"` use a fixed versioning strategy (`"/v12/js/lib/mymodule.js"`), while other resources still use the content one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

See `ResourceProperties` for more supported options.

Tip

This feature has been thoroughly described in a dedicated blog post and in Spring Framework's reference documentation.

Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

Custom Favicon

Spring Boot looks for a `favicon.ico` in the configured static content locations and the root of the classpath (in that order). If such a file is present, it is automatically used as the favicon of the application.

Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like `"GET /projects/spring-boot.json"` won't be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a best practice for Spring MVC applications. This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that don't consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like `"GET /projects/spring-boot?format=json"` will be mapped to `@GetMapping("/projects/spring-boot")`:

```
spring.mvc.content-negotiation.favor-parameter=true

# We can change the parameter name, which is "format" by default:
# spring.mvc.content-negotiation.parameter-name=myparam

# We can also register additional file extensions/media types with:
spring.mvc.content-negotiation.media-types.markdown=text/markdown
```

If you understand the caveats and would still like your application to use suffix pattern matching, the following configuration is required:

```
spring.mvc.content-negotiation.favor-path-extension=true

# You can also restrict that feature to known extensions only
# spring.mvc.path-match.use-registered-suffix-pattern=true

# We can also register additional file extensions/media types with:
# spring.mvc.content-negotiation.media-types.adoc=text/asciidoc
```

ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer` `@Bean`, Spring Boot automatically configures Spring MVC to use it.

Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

> **Tip**
>
> If possible, JSPs should be avoided. There are several known limitations when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

> **Tip**
>
> Depending on how you run your application, IntelliJ IDEA orders the classpath differently. Running your application in the IDE from its main method results in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the templates on the classpath. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first. Alternatively, you can configure the template prefix to search every `templates` directory on the classpath, as follows: `classpath*:/templates/`.

Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a "global" error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a "whitelabel" error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`). To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

> **Tip**
>
> The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example:

```
@ControllerAdvice(basePackageClasses = AcmeController.class)
public class AcmeControllerAdvice extends ResponseEntityExceptionHandler {

 @ExceptionHandler(YourException.class)
 @ResponseBody
 ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {
  HttpStatus status = getStatus(request);
  return new ResponseEntity<>(new CustomErrorType(status.value(), ex.getMessage()), status);
 }

 private HttpStatus getStatus(HttpServletRequest request) {
  Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status_code");
  if (statusCode == null) {
   return HttpStatus.INTERNAL_SERVER_ERROR;
  }
  return HttpStatus.valueOf(statusCode);
 }

}
```

In the preceding example, if `YourException` is thrown by a controller defined in the same package as `AcmeController`, a JSON representation of the `CustomErrorType` POJO is used instead of the `ErrorAttributes` representation.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftl
             +- <other templates>
```

For more complex mappings, you can also add beans that implement the `ErrorViewResolver` interface, as shown in the following example:

```
public class MyErrorViewResolver implements ErrorViewResolver {

 @Override
 public ModelAndView resolveErrorView(HttpServletRequest request,
   HttpStatus status, Map<String, Object> model) {
```

```
  // Use the request or status to optionally return a ModelAndView
  return ...
 }

}
```

You can also use regular Spring MVC features such as `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` then picks up any unhandled exceptions.

Mapping Error Pages outside of Spring MVC

For applications that do not use Spring MVC, you can use the `ErrorPageRegistrar` interface to directly register `ErrorPages`. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

```
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
 return new MyErrorPageRegistrar();
}

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar {

 @Override
 public void registerErrorPages(ErrorPageRegistry registry) {
  registry.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
 }

}
```

> **Note**
>
> If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

```
@Bean
public FilterRegistrationBean myFilter() {
 FilterRegistrationBean registration = new FilterRegistrationBean();
 registration.setFilter(new MyFilter());
 ...
 registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.class));
 return registration;
}
```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

CAUTION:When deployed to a servlet container, Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

Spring HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications, including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that doing so disables the `ObjectMapper` customization described earlier.

CORS Support

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that lets you specify in a flexible way what kind of cross-domain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAME or JSONP.

As of version 4.2, Spring MVC supports CORS. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. Global CORS configuration can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

```
@Configuration
public class MyConfiguration {

 @Bean
 public WebMvcConfigurer corsConfigurer() {
  return new WebMvcConfigurer() {
   @Override
   public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**");
   }
  };
 }
}
```

27.2 The “Spring WebFlux Framework”

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the Servlet API, is fully asynchronous and non-blocking, and implements the Reactive Streams specification through the Reactor project.

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

```
@RestController
@RequestMapping("/users")
public class MyRestController {

 @GetMapping("/{user}")
 public Mono<User> getUser(@PathVariable Long user) {
  // ...
 }

 @GetMapping("/{user}/customers")
 public Flux<Customer> getUserCustomers(@PathVariable Long user) {
  // ...
 }

 @DeleteMapping("/{user}")
 public Mono<User> deleteUser(@PathVariable Long user) {
  // ...
 }

}
```

“WebFlux.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

```
@Configuration
public class RoutingConfiguration {

 @Bean
 public RouterFunction<ServerResponse> monoRouterFunction(UserHandler userHandler) {
  return route(GET("/{user}").and(accept(APPLICATION_JSON)), userHandler::getUser)
    .andRoute(GET("/{user}/customers").and(accept(APPLICATION_JSON)), userHandler::getUserCustomers)
    .andRoute(DELETE("/{user}").and(accept(APPLICATION_JSON)), userHandler::deleteUser);
 }

}

@Component
public class UserHandler {

 public Mono<ServerResponse> getUser(ServerRequest request) {
  // ...
 }

 public Mono<ServerResponse> getUserCustomers(ServerRequest request) {
  // ...
 }

 public Mono<ServerResponse> deleteUser(ServerRequest request) {
  // ...
 }
}
```

WebFlux is part of the Spring Framework and detailed information is available in its reference documentation.

Tip

You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

To get started, add the `spring-boot-starter-webflux` module to your application.

Note

Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type to `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

Spring WebFlux Auto-configuration

Spring Boot provides auto-configuration for Spring WebFlux that works well with most applications.

The auto-configuration adds the following features on top of Spring’s defaults:

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described later in this document).
- Support for serving static resources, including support for WebJars (described later in this document).

If you want to keep Spring Boot WebFlux features and you want to add additional WebFlux configuration, you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

HTTP Codecs with HttpMessageReaders and HttpMessageWriters

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries available in your classpath.

Spring Boot applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

```
import org.springframework.boot.web.codec.CodecCustomizer;

@Configuration
public class MyConfiguration {

 @Bean
 public CodecCustomizer myCodecCustomizer() {
  return codecConfigurer -> {
   // ...
  }
 }

}
```

You can also leverage Boot's custom JSON serializers and deserializers.

Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

By default, resources are mapped on `/**`, but you can tune that by setting the `spring.webflux.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

```
spring.webflux.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using `spring.resources.static-locations`. Doing so replaces the default values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the “standard” static resource locations listed earlier, a special case is made for Webjars content. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

> **Tip**
>
> Spring WebFlux applications do not strictly depend on the Servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Thymeleaf
- Mustache

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a “whitelabel” error handler that renders the same data in HTML format. You can also provide your own HTML templates to display errors (see the next section).

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because a `WebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

```java
public class CustomErrorWebExceptionHandler extends AbstractErrorWebExceptionHandler {

 // Define constructor here

 @Override
 protected RouterFunction<ServerResponse> getRoutingFunction(ErrorAttributes errorAttributes) {

  return RouterFunctions
    .route(aPredicate, aHandler)
    .andRoute(anotherPredicate, anotherHandler);
 }

}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or built with templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a Mustache template, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.mustache
             +- <other templates>
```

Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implemented `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter	Order
`MetricsWebFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`WebFilterChainProxy` (Spring Security)	`-100`
`HttpTraceWebFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

27.3 JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. Jersey 1.x and Apache CXF work quite well out of the box if you register their `Servlet` or `Filter` as a `@Bean` in your application context. Jersey 2.x has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey 2.x, include the `spring-boot-starter-jersey` as a dependency and then you need one `@Bean` of type `ResourceConfig` in which you register all the endpoints, as shown in the following example:

```
@Component
public class JerseyConfig extends ResourceConfig {

 public JerseyConfig() {
  register(Endpoint.class);
 }

}
```

Warning

Jersey's support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in `WEB-INF/classes` when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```
@Component
@Path("/hello")
public class Endpoint {

 @GET
 public String message() {
  return "Hello";
 }

}
```

Since the `Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a Servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

There is a Jersey sample so that you can see how to set things up. There is also a Jersey 1.x sample. Note that, in the Jersey 1.x sample, the spring-boot maven plugin has been configured to unpack some Jersey jars so that they can be scanned by the JAX-RS implementation (because the sample asks for them to be scanned in its `Filter` registration). If any of your JAX-RS resources are packaged as nested jars, you may need to do the same.

27.4 Embedded Servlet Container Support

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port `8080`.

> **Warning**
>
> If you choose to use Tomcat on CentOS, be aware that, by default, a temporary directory is used to store compiled JSPs, file uploads, and so on. This directory may be deleted by `tmpwatch` while your application is running, leading to failures. To avoid this behavior, you may want to customize your `tmpwatch` configuration such that `tomcat.*` directories are not deleted or configure `server.tomcat.basedir` such that embedded Tomcat uses a different location.

Servlets, Filters, and listeners

When using an embedded servlet container, you can register servlets, filters, and all the listeners (such as `HttpSessionListener`) from the Servlet spec, either by using Spring beans or by scanning for Servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any `Servlet`, `Filter`, or servlet `*Listener` instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet, it is mapped to `/`. In the case of multiple servlet beans, the bean name is used as a path prefix. Filters map to `/*`.

If convention-based mapping is not flexible enough, you can use the `ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean` classes for complete control.

Spring Boot ships with many auto-configurations that may define Filter beans. Here are a few examples of Filters and their respective order (lower order value means higher precedence):

Servlet Filter	Order
`OrderedCharacterEncodingFilter`	`Ordered.HIGHEST_PRECEDENCE`
`WebMvcMetricsFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`ErrorPageFilter`	`Ordered.HIGHEST_PRECEDENCE + 1`
`HttpTraceFilter`	`Ordered.LOWEST_PRECEDENCE - 10`

It is usually safe to leave Filter beans unordered.

If a specific order is required, you should avoid configuring a Filter that reads the request body at `Ordered.HIGHEST_PRECEDENCE`, since it might go against the character encoding configuration of your application. If a Servlet filter wraps the request, it should be configured with an order that is less than or equal to `FilterRegistrationBean.REQUEST_WRAPPER_FILTER_MAX_ORDER`.

Servlet Context Initialization

Embedded servlet containers do not directly execute the Servlet 3.0+ `javax.servlet.ServletContainerInitializer` interface or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.web.servlet.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

Tip

`@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

The ServletWebServerApplicationContext

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support. The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.

Note

You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistence`), session timeout (`server.servlet.session.timeout`), location of session

data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).

- Error management: Location of the error page (`server.error.path`) and so on.
- SSL
- HTTP compression

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, access logs can be configured with specific features of the embedded servlet container.

Tip

See the `ServerProperties` class for a complete list.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. Dedicated variants exist for Tomcat, Jetty, and Undertow. The following example shows programmatically setting the port:

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.boot.web.servlet.server.ConfigurableServletWebServerFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements
 WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

 @Override
 public void customize(ConfigurableServletWebServerFactory server) {
  server.setPort(9000);
 }

}
```

Customizing ConfigurableServletWebServerFactory Directly

If the preceding customization techniques are too limited, you can register the `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` bean yourself.

```
@Bean
public ConfigurableServletWebServerFactory webServerFactory() {
 TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();
 factory.setPort(9000);
 factory.setSessionTimeout(10, TimeUnit.MINUTES);
 factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/notfound.html"));
 return factory;
}
```

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the source code documentation for details.

JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Tomcat, it should work if you use war packaging. That is, an executable war works and is also deployable to a standard container (not limited to, but including Tomcat). An executable jar does not work because of a hard-coded file pattern in Tomcat.
- With Jetty, it should work if you use war packaging. That is, an executable war works, and is also deployable to any standard container.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for error handling. Custom error pages should be used instead.

There is a JSP sample so that you can see how to set things up.

28. Security

If Spring Security is on the classpath, then web applications are secure by default. Spring Boot relies on Spring Security's content-negotiation strategy to determine whether to use `httpBasic` or `formLogin`. To add method-level security to a web application, you can also add `@EnableGlobalMethodSecurity` with your desired settings. Additional information can be found in the Spring Security Reference Guide.

The default `AuthenticationManager` has a single user. The user name is `user`, and the password is random and is printed at INFO level when the application starts, as shown in the following example:

```
Using generated security password: 78fa095d-3f4c-48b1-ad50-e24c31d5cf35
```

Note

If you fine-tune your logging configuration, ensure that the `org.springframework.boot.autoconfigure.security` category is set to log `INFO`-level messages. Otherwise, the default password is not printed.

You can change the username and password by providing a `spring.security.user.name` and `spring.security.user.password`.

The basic features you get by default in a web application are:

- A `UserDetailsService` (or `ReactiveUserDetailsService` in case of a WebFlux application) bean with in-memory store and a single user with a generated password (see `SecurityProperties.User` for the properties of the user).
- Form-based login or HTTP Basic security (depending on Content-Type) for the entire application (including actuator endpoints if actuator is on the classpath).

28.1 MVC Security

The default security configuration is implemented in `SecurityAutoConfiguration` and in the classes imported from there (`SpringBootWebSecurityConfiguration` for web security and `AuthenticationManagerConfiguration` for authentication configuration, which is also relevant in non-web applications). To switch off the default web application security configuration completely, you can add a bean of type `WebSecurityConfigurerAdapter` (doing so does not disable the authentication manager configuration or Actuator's security).

To also switch off the authentication manager configuration, you can add a bean of type `UserDetailsService`, `AuthenticationProvider`, or `AuthenticationManager`. There are several secure applications in the Spring Boot samples to get you started with common use cases.

Access rules can be overridden by adding a custom `WebSecurityConfigurerAdapter`. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `RequestMatcher` that is based on the `management.endpoints.web.base-path` property. `PathRequest` can be used to create a `RequestMatcher` for resources in commonly used locations.

28.2 WebFlux Security

The default security configuration is implemented in `ReactiveSecurityAutoConfiguration` and in the classes imported from there (`WebFluxSecurityConfiguration` for web security and `ReactiveAuthenticationManagerConfiguration` for authentication configuration, which is also relevant in non-web applications). To switch off the default web application security configuration completely, you can add a bean of type `WebFilterChainProxy` (doing so does not disable the authentication manager configuration or Actuator's security).

To also switch off the authentication manager configuration, you can add a bean of type `ReactiveUserDetailsService` or `ReactiveAuthenticationManager`.

Access rules can be configured by adding a custom `SecurityWebFilterChain`. Spring Boot provides convenience methods that can be used to override access rules for actuator endpoints and static resources. `EndpointRequest` can be used to create a `ServerWebExchangeMatcher` that is based on the `management.endpoints.web.base-path` property.

`PathRequest` can be used to create a `ServerWebExchangeMatcher` for resources in commonly used locations.

28.3 OAuth2

OAuth2 is a widely used authorization framework that is supported by Spring.

Client

If you have `spring-security-oauth2-client` on your classpath, you can take advantage of some auto-configuration to make it easy to set up an OAuth2 Client. This configuration makes use of the properties under `OAuth2ClientProperties`.

You can register multiple OAuth2 clients and providers under the `spring.security.oauth2.client` prefix, as shown in the following example:

```
spring.security.oauth2.client.registration.my-client-1.client-id=abcd
spring.security.oauth2.client.registration.my-client-1.client-secret=password
spring.security.oauth2.client.registration.my-client-1.client-name=Client for user scope
spring.security.oauth2.client.registration.my-client-1.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-1.scope=user
spring.security.oauth2.client.registration.my-client-1.redirect-uri-template=http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-1.client-authentication-method=basic
spring.security.oauth2.client.registration.my-client-1.authorization-grant-type=authorization_code

spring.security.oauth2.client.registration.my-client-2.client-id=abcd
spring.security.oauth2.client.registration.my-client-2.client-secret=password
spring.security.oauth2.client.registration.my-client-2.client-name=Client for email scope
spring.security.oauth2.client.registration.my-client-2.provider=my-oauth-provider
spring.security.oauth2.client.registration.my-client-2.scope=email
spring.security.oauth2.client.registration.my-client-2.redirect-uri-template=http://my-redirect-uri.com
spring.security.oauth2.client.registration.my-client-2.client-authentication-method=basic
spring.security.oauth2.client.registration.my-client-2.authorization-grant-type=authorization_code

spring.security.oauth2.client.provider.my-oauth-provider.authorization-uri=http://my-auth-server/oauth/
authorize
spring.security.oauth2.client.provider.my-oauth-provider.token-uri=http://my-auth-server/oauth/token
spring.security.oauth2.client.provider.my-oauth-provider.user-info-uri=http://my-auth-server/userinfo
spring.security.oauth2.client.provider.my-oauth-provider.jwk-set-uri=http://my-auth-server/token_keys
spring.security.oauth2.client.provider.my-oauth-provider.user-name-attribute=name
```

By default, Spring Security's `OAuth2LoginAuthenticationFilter` only processes URLs matching `/login/oauth2/code/*`. If you want to customize the `redirect-uri-template` to use a different pattern, you need to provide configuration to process that custom pattern. For example, you can add your own `WebSecurityConfigurerAdapter` that resembles the following:

```
public class OAuth2LoginSecurityConfig extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http
   .authorizeRequests()
    .anyRequest().authenticated()
    .and()
   .oauth2Login()
    .redirectionEndpoint()
     .baseUri("/custom-callback");
 }
}
```

For common OAuth2 and OpenID providers, including Google, Github, Facebook, and Okta, we provide a set of provider defaults (`google`, `github`, `facebook`, and `okta`, respectively).

If you do not need to customize these providers, you can set the `provider` attribute to the one for which you need to infer defaults. Also, if the ID of your client matches the default supported provider, Spring Boot infers that as well.

In other words, the two configurations in the following example use the Google provider:

```
spring.security.oauth2.client.registration.my-client.client-id=abcd
spring.security.oauth2.client.registration.my-client.client-secret=password
spring.security.oauth2.client.registration.my-client.provider=google

spring.security.oauth2.client.registration.google.client-id=abcd
spring.security.oauth2.client.registration.google.client-secret=password
```

28.4 Actuator Security

For security purposes, all actuators other than `/health` and `/info` are disabled by default. The `management.endpoints.web.expose` flag can be used to enable the actuators. If Spring Security is on the classpath and no other WebSecurityConfigurerAdapter is present, the actuators are secured by Spring Boot auto-config. If you define a custom `WebSecurityConfigurerAdapter`, Spring Boot auto-config will back off and you will be in full control of actuator access rules.

Note

Before setting the `management.endpoints.web.expose`, ensure that the exposed actuators do not contain sensitive information and/or are secured by placing them behind a firewall or by something like Spring Security.

29. Working with SQL Databases

The Spring Framework provides extensive support for working with SQL databases, from direct JDBC access using `JdbcTemplate` to complete "object relational mapping" technologies such as Hibernate. Spring Data provides an additional level of functionality: creating `Repository` implementations directly from interfaces and using conventions to generate queries from your method names.

29.1 Configure a DataSource

Java's `javax.sql.DataSource` interface provides a standard method of working with database connections. Traditionally, a 'DataSource' uses a `URL` along with some credentials to establish a database connection.

> **Tip**
>
> See the "How-to" section for more advanced examples, typically to take full control over the configuration of the DataSource.

Embedded Database Support

It is often convenient to develop applications by using an in-memory embedded database. Obviously, in-memory databases do not provide persistent storage. You need to populate your database when your application starts and be prepared to throw away data when your application ends.

> **Tip**
>
> The "How-to" section includes a section on how to initialize a database.

Spring Boot can auto-configure embedded H2, HSQL, and Derby databases. You need not provide any connection URLs. You need only include a build dependency to the embedded database that you want to use.

> **Note**
>
> If you are using this feature in your tests, you may notice that the same database is reused by your whole test suite regardless of the number of application contexts that you use. If you want to make sure that each context has a separate embedded database, you should set `spring.datasource.generate-unique-name` to `true`.

For example, the typical POM dependencies would be as follows:

```xml
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
 <groupId>org.hsqldb</groupId>
 <artifactId>hsqldb</artifactId>
 <scope>runtime</scope>
</dependency>
```

> **Note**
>
> You need a dependency on `spring-jdbc` for an embedded database to be auto-configured. In this example, it is pulled in transitively through `spring-boot-starter-data-jpa`.

> **Tip**
>
> If, for whatever reason, you do configure the connection URL for an embedded database, take care to ensure that the database's automatic shutdown is disabled. If you use H2, you should use `DB_CLOSE_ON_EXIT=FALSE` to do so. If you use HSQLDB, you should ensure that `shutdown=true` is not used. Disabling the database's automatic shutdown lets Spring Boot control when the database is closed, thereby ensuring that it happens once access to the database is no longer needed.

Connection to a Production Database

Production database connections can also be auto-configured by using a pooling `DataSource`. Spring Boot uses the following algorithm for choosing a specific implementation:

1. We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it.
2. Otherwise, if the Tomcat pooling `DataSource` is available, we use it.
3. If neither HikariCP nor the Tomcat pooling datasource are available and if Commons DBCP2 is available, we use it.

If you use the `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` "starters", you automatically get a dependency to `HikariCP`.

> **Note**
>
> You can bypass that algorithm completely and specify the connection pool to use by setting the `spring.datasource.type` property. This is especially important if you run your application in a Tomcat container, as `tomcat-jdbc` is provided by default.

> **Tip**
>
> Additional connection pools can always be configured manually. If you define your own `DataSource` bean, auto-configuration does not occur.

DataSource configuration is controlled by external configuration properties in `spring.datasource.*`. For example, you might declare the following section in `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

> **Note**
>
> You should at least specify the URL by setting the `spring.datasource.url` property. Otherwise, Spring Boot tries to auto-configure an embedded database.

Tip

You often do not need to specify the `driver-class-name`, since Spring Boot can deduce it for most databases from the `url`.

Note

For a pooling `DataSource` to be created, we need to be able to verify that a valid `Driver` class is available, so we check for that before doing anything. In other words, if you set `spring.datasource.driver-class-name=com.mysql.jdbc.Driver`, then that class has to be loadable.

See `DataSourceProperties` for more of the supported options. These are the standard options that work regardless of the actual implementation. It is also possible to fine-tune implementation-specific settings by using their respective prefix (`spring.datasource.hikari.*`, `spring.datasource.tomcat.*`, and `spring.datasource.dbcp2.*`). Refer to the documentation of the connection pool implementation you are using for more details.

For instance, if you use the Tomcat connection pool, you could customize many additional settings, as shown in the following example:

```
# Number of ms to wait before throwing an exception if no connection is available.
spring.datasource.tomcat.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at the same time.
spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.
spring.datasource.tomcat.test-on-borrow=true
```

Connection to a JNDI DataSource

If you deploy your Spring Boot application to an Application Server, you might want to configure and manage your DataSource by using your Application Server’s built-in features and access it by using JNDI.

The `spring.datasource.jndi-name` property can be used as an alternative to the `spring.datasource.url`, `spring.datasource.username`, and `spring.datasource.password` properties to access the `DataSource` from a specific JNDI location. For example, the following section in `application.properties` shows how you can access a JBoss AS defined `DataSource`:

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

29.2 Using JdbcTemplate

Spring’s `JdbcTemplate` and `NamedParameterJdbcTemplate` classes are auto-configured, and you can `@Autowire` them directly into your own beans, as shown in the following example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final JdbcTemplate jdbcTemplate;

 @Autowired
 public MyBean(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
 }

 // ...

}
```

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, as shown in the following example:

```
spring.jdbc.template.max-rows=500
```

Note

The `NamedParameterJdbcTemplate` reuses the same `JdbcTemplate` instance behind the scenes. If more than one `JdbcTemplate` is defined and no primary candidate exists, the `NamedParameterJdbcTemplate` is not auto-configured.

29.3 JPA and “Spring Data”

The Java Persistence API is a standard technology that lets you “map” objects to relational databases. The `spring-boot-starter-data-jpa` POM provides a quick way to get started. It provides the following key dependencies:

- Hibernate: One of the most popular JPA implementations.
- Spring Data JPA: Makes it easy to implement JPA-based repositories.
- Spring ORMs: Core ORM support from the Spring Framework.

Tip

We do not go into too many details of JPA or Spring Data here. You can follow the “Accessing Data with JPA” guide from spring.io and read the Spring Data JPA and Hibernate reference documentation.

Entity Classes

Traditionally, JPA “Entity” classes are specified in a `persistence.xml` file. With Spring Boot, this file is not necessary and “Entity Scanning” is used instead. By default, all packages below your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) are searched.

Any classes annotated with `@Entity`, `@Embeddable`, or `@MappedSuperclass` are considered. A typical entity class resembles the following example:

```
package com.example.myapp.domain;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class City implements Serializable {

 @Id
 @GeneratedValue
 private Long id;

 @Column(nullable = false)
 private String name;

 @Column(nullable = false)
 private String state;

 // ... additional members, often include @OneToMany mappings

 protected City() {
  // no-args constructor required by JPA spec
  // this one is protected since it shouldn't be used directly
 }

 public City(String name, String state) {
  this.name = name;
  this.country = country;
 }

 public String getName() {
  return this.name;
 }

 public String getState() {
  return this.state;
 }

 // ... etc

}
```

Tip

You can customize entity scanning locations by using the `@EntityScan` annotation. See the "Section 78.4, "Separate @Entity Definitions from Spring Configuration"" how-to.

Spring Data JPA Repositories

{http://projects.spring.io/spring-data-jpa/}[Spring Data JPA] repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a `CityRepository` interface might declare a `findAllByState(String state)` method to find all the cities in a given state.

For more complex queries, you can annotate your method with Spring Data's `Query` annotation.

Spring Data repositories usually extend from the `Repository` or `CrudRepository` interfaces. If you use auto-configuration, repositories are searched from the package containing your main configuration class (the one annotated with `@EnableAutoConfiguration` or `@SpringBootApplication`) down.

The following example shows a typical Spring Data repository interface definition:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

Tip

We have barely scratched the surface of Spring Data JPA. For complete details, see the Spring Data JPA reference documentation.

Creating and Dropping JPA Databases

By default, JPA databases are automatically created **only** if you use an embedded database (H2, HSQL, or Derby). You can explicitly configure JPA settings by using `spring.jpa.*` properties. For example, to create and drop tables you can add the following line to your `application.properties`:

```
spring.jpa.hibernate.ddl-auto=create-drop
```

Note

Hibernate's own internal property name for this (if you happen to remember it better) is `hibernate.hbm2ddl.auto`. You can set it, along with other Hibernate native properties, by using `spring.jpa.properties.*` (the prefix is stripped before adding them to the entity manager). The following line shows an example of setting JPA properties for Hibernate:

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```

The line in the preceding example passes a value of `true` for the `hibernate.globally_quoted_identifiers` property to the Hibernate entity manager.

By default, the DDL execution (or validation) is deferred until the `ApplicationContext` has started. There is also a `spring.jpa.generate-ddl` flag, but it is not used if Hibernate auto-configuration is active, because the `ddl-auto` settings are more fine-grained.

Open EntityManager in View

If you are running a web application, Spring Boot by default registers `OpenEntityManagerInViewInterceptor` to apply the "Open EntityManager in View" pattern, to allow for lazy loading in web views. If you do not want this behavior, you should set `spring.jpa.open-in-view` to `false` in your `application.properties`.

29.4 Using H2's Web Console

The H2 database provides a browser-based console that Spring Boot can auto-configure for you. The console is auto-configured when the following conditions are met:

- You are developing a web application.
- `com.h2database:h2` is on the classpath.
- You are using Spring Boot's developer tools.

Tip

If you are not using Spring Boot's developer tools but would still like to make use of H2's console, you can configure the `spring.h2.console.enabled` property with a value of `true`.

Note

The H2 console is only intended for use during development, so you should take care to ensure that `spring.h2.console.enabled` is not set to `true` in production.

Changing the H2 Console’s Path

By default, the console is available at `/h2-console`. You can customize the console’s path by using the `spring.h2.console.path` property.

29.5 Using jOOQ

Java Object Oriented Querying (jOOQ) is a popular product from Data Geekery which generates Java code from your database and lets you build type-safe SQL queries through its fluent API. Both the commercial and open source editions can be used with Spring Boot.

Code Generation

In order to use jOOQ type-safe queries, you need to generate Java classes from your database schema. You can follow the instructions in the jOOQ user manual. If you use the `jooq-codegen-maven` plugin and you also use the `spring-boot-starter-parent` “parent POM”, you can safely omit the plugin’s `<version>` tag. You can also use Spring Boot-defined version variables (such as `h2.version`) to declare the plugin’s database dependency. The following listing shows an example:

```
<plugin>
 <groupId>org.jooq</groupId>
 <artifactId>jooq-codegen-maven</artifactId>
 <executions>
  ...
 </executions>
 <dependencies>
  <dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <version>${h2.version}</version>
  </dependency>
 </dependencies>
 <configuration>
  <jdbc>
   <driver>org.h2.Driver</driver>
   <url>jdbc:h2:~/yourdatabase</url>
  </jdbc>
  <generator>
   ...
  </generator>
 </configuration>
</plugin>
```

Using DSLContext

The fluent API offered by jOOQ is initiated through the `org.jooq.DSLContext` interface. Spring Boot auto-configures a `DSLContext` as a Spring Bean and connects it to your application `DataSource`. To use the `DSLContext`, you can `@Autowire` it, as shown in the following example:

```
@Component
public class JooqExample implements CommandLineRunner {

 private final DSLContext create;

 @Autowired
 public JooqExample(DSLContext dslContext) {
  this.create = dslContext;
 }

}
```

Tip

The jOOQ manual tends to use a variable named `create` to hold the `DSLContext`.

You can then use the `DSLContext` to construct your queries, as shown in the following example:

```
public List<GregorianCalendar> authorsBornAfter1980() {
 return this.create.selectFrom(AUTHOR)
  .where(AUTHOR.DATE_OF_BIRTH.greaterThan(new GregorianCalendar(1980, 0, 1)))
  .fetch(AUTHOR.DATE_OF_BIRTH);
}
```

jOOQ SQL Dialect

Unless the `spring.jooq.sql-dialect` property has been configured, Spring Boot determines the SQL dialect to use for your datasource. If Spring Boot could not detect the dialect, it uses `DEFAULT`.

Note

Spring Boot can only auto-configure dialects supported by the open source version of jOOQ.

Customizing jOOQ

More advanced customizations can be achieved by defining your own `@Bean` definitions, which is used when the jOOQ `Configuration` is created. You can define beans for the following jOOQ Types:

- `ConnectionProvider`
- `TransactionProvider`
- `RecordMapperProvider`
- `RecordListenerProvider`
- `ExecuteListenerProvider`
- `VisitListenerProvider`

You can also create your own `org.jooq.Configuration` `@Bean` if you want to take complete control of the jOOQ configuration.

30. Working with NoSQL Technologies

Spring Data provides additional projects that help you access a variety of NoSQL technologies, including: MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Cassandra, Couchbase and LDAP. Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr Cassandra, Couchbase, and LDAP. You can make use of the other projects, but you must configure them yourself. Refer to the appropriate reference documentation at projects.spring.io/spring-data.

30.1 Redis

Redis is a cache, message broker, and richly-featured key-value store. Spring Boot offers basic auto-configuration for the Lettuce and Jedis client libraries and the abstractions on top of them provided by Spring Data Redis.

There is a `spring-boot-starter-data-redis` "Starter" for collecting the dependencies in a convenient way. By default, it uses Lettuce. That starter handles both traditional and reactive applications.

> **Tip**
>
> we also provide a `spring-boot-starter-data-redis-reactive` "Starter" for consistency with the other stores with reactive support.

Connecting to Redis

You can inject an auto-configured `RedisConnectionFactory`, `StringRedisTemplate`, or vanilla `RedisTemplate` instance as you would any other Spring Bean. By default, the instance tries to connect to a Redis server at `localhost:6379`. The following listing shows an example of such a bean:

```
@Component
public class MyBean {

 private StringRedisTemplate template;

 @Autowired
 public MyBean(StringRedisTemplate template) {
  this.template = template;
 }

 // ...

}
```

> **Tip**
>
> You can also register an arbitrary number of beans that implement `LettuceClientConfigurationBuilderCustomizer` for more advanced customizations. If you use Jedis, `JedisClientConfigurationBuilderCustomizer` is also available.

If you add your own `@Bean` of any of the auto-configured types, it replaces the default (except in the case of `RedisTemplate`, when the exclusion is based on the bean name, `redisTemplate`, not its type). By default, if `commons-pool2` is on the classpath, you get a pooled connection factory.

30.2 MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the `spring-boot-starter-data-mongodb` and `spring-boot-starter-data-mongodb-reactive` “Starters”.

Connecting to a MongoDB Database

To access Mongo databases, you can inject an auto-configured `org.springframework.data.mongodb.MongoDbFactory`. By default, the instance tries to connect to a MongoDB server at `mongodb://localhost/test` The following example shows how to connect to a MongoDB database:

```
import org.springframework.data.mongodb.MongoDbFactory;
import com.mongodb.DB;

@Component
public class MyBean {

 private final MongoDbFactory mongo;

 @Autowired
 public MyBean(MongoDbFactory mongo) {
  this.mongo = mongo;
 }

 // ...

 public void example() {
  DB db = mongo.getDb();
  // ...
 }

}
```

You can set the `spring.data.mongodb.uri` property to change the URL and configure additional settings such as the *replica set*, as shown in the following example:

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345,mongo2.example.com:23456/test
```

Alternatively, as long as you use Mongo 2.x, you can specify a `host`/`port`. For example, you might declare the following settings in your `application.properties`:

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

Note

If you use the Mongo 3.0 Java driver, `spring.data.mongodb.host` and `spring.data.mongodb.port` are not supported. In such cases, `spring.data.mongodb.uri` should be used to provide all of the configuration.

Tip

If `spring.data.mongodb.port` is not specified, the default of `27017` is used. You could delete this line from the example shown earlier.

Tip

If you do not use Spring Data Mongo, you can inject `com.mongodb.MongoClient` beans instead of using `MongoDbFactory`. If you want to take complete control of establishing the MongoDB connection, you can also declare your own `MongoDbFactory` or `MongoClient` bean.

Note

If you are using the reactive driver, Netty is required for SSL. The auto-configuration configures this factory automatically if Netty is available and the factory to use hasn't been customized already.

MongoTemplate

Spring Data MongoDB provides a `MongoTemplate` class that is very similar in its design to Spring's `JdbcTemplate`. As with `JdbcTemplate`, Spring Boot auto-configures a bean for you to inject the template, as follows:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final MongoTemplate mongoTemplate;

 @Autowired
 public MyBean(MongoTemplate mongoTemplate) {
  this.mongoTemplate = mongoTemplate;
 }

 // ...

}
```

See the `MongoOperations` Javadoc for complete details.

Spring Data MongoDB Repositories

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed automatically, based on method names.

In fact, both Spring Data JPA and Spring Data MongoDB share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a Mongo data class rather than a JPA `@Entity`, it works in the same way, as shown in the following example:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends Repository<City, Long> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountryAllIgnoringCase(String name, String country);

}
```

Tip

You can customize document scanning locations by using the `@EntityScan` annotation.

Tip

For complete details of Spring Data MongoDB, including its rich object mapping technologies, refer to its reference documentation.

Embedded Mongo

Spring Boot offers auto-configuration for Embedded Mongo. To use it in your Spring Boot application, add a dependency on `de.flapdoodle.embed:de.flapdoodle.embed.mongo`.

The port that Mongo listens on can be configured by setting the `spring.data.mongodb.port` property. To use a randomly allocated free port, use a value of 0. The `MongoClient` created by `MongoAutoConfiguration` is automatically configured to use the randomly allocated port.

Note

If you do not configure a custom port, the embedded support uses a random port (rather than 27017) by default.

If you have SLF4J on the classpath, the output produced by Mongo is automatically routed to a logger named `org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongo`.

You can declare your own `IMongodConfig` and `IRuntimeConfig` beans to take control of the Mongo instance's configuration and logging routing.

30.3 Neo4j

Neo4j is an open-source NoSQL graph database that uses a rich data model of nodes related by first class relationships, which is better suited for connected big data than traditional rdbms approaches. Spring Boot offers several conveniences for working with Neo4j, including the `spring-boot-starter-data-neo4j` “Starter”.

Connecting to a Neo4j Database

You can inject an auto-configured `Neo4jSession`, `Session`, or `Neo4jOperations` instance as you would any other Spring Bean. By default, the instance tries to connect to a Neo4j server at `localhost:7474`. The following example shows how to inject a Neo4j bean:

```
@Component
public class MyBean {

 private final Neo4jTemplate neo4jTemplate;

 @Autowired
 public MyBean(Neo4jTemplate neo4jTemplate) {
  this.neo4jTemplate = neo4jTemplate;
 }

 // ...

}
```

You can take full control of the configuration by adding a `org.neo4j.ogm.config.Configuration` `@Bean` of your own. Also, adding a `@Bean` of type `Neo4jOperations` disables the auto-configuration.

You can configure the user and credentials to use by setting the `spring.data.neo4j.*` properties, as shown in the following example:

```
spring.data.neo4j.uri=http://my-server:7474
spring.data.neo4j.username=neo4j
spring.data.neo4j.password=secret
```

Using the Embedded Mode

If you add `org.neo4j:neo4j-ogm-embedded-driver` to the dependencies of your application, Spring Boot automatically configures an in-process embedded instance of Neo4j that does not persist any data when your application shuts down. You can explicitly disable that mode by setting `spring.data.neo4j.embedded.enabled=false`. You can also enable persistence for the embedded mode by providing a path to a database file, as shown in the following example:

```
spring.data.neo4j.uri=file://var/tmp/graph.db
```

Note

The Neo4j OGM embedded driver does not provide the Neo4j kernel. Users are expected to provide this dependency manually. See the documentation for more details.

Neo4jSession

By default, if you are running a web application, the session is bound to the thread for the entire processing of the request (that is, it uses the "Open Session in View" pattern). If you do not want this behavior, add the following line to your `application.properties` file:

```
spring.data.neo4j.open-in-view=false
```

Spring Data Neo4j Repositories

Spring Data includes repository support for Neo4j.

In fact, both Spring Data JPA and Spring Data Neo4j share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a Neo4j OGM `@NodeEntity` rather than a JPA `@Entity`, it works in the same way.

Tip

You can customize entity scanning locations by using the `@EntityScan` annotation.

To enable repository support (and optionally support for `@Transactional`), add the following two annotations to your Spring configuration:

```
@EnableNeo4jRepositories(basePackages = "com.example.myapp.repository")
@EnableTransactionManagement
```

Repository Example

The following example shows an interface definition for a Neo4j repository:

```
package com.example.myapp.domain;

import org.springframework.data.domain.*;
import org.springframework.data.repository.*;

public interface CityRepository extends GraphRepository<City> {

 Page<City> findAll(Pageable pageable);

 City findByNameAndCountry(String name, String country);

}
```

Tip

For complete details of Spring Data Neo4j, including its rich object mapping technologies, refer to the reference documentation.

30.4 Gemfire

Spring Data Gemfire provides convenient Spring-friendly tools for accessing the Pivotal Gemfire data management platform. There is a `spring-boot-starter-data-gemfire` “Starter” for collecting the dependencies in a convenient way. There is currently no auto-configuration support for Gemfire, but you can enable Spring Data Repositories with a single annotation: `@EnableGemfireRepositories`.

30.5 Solr

Apache Solr is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 client library and the abstractions on top of it provided by Spring Data Solr. There is a `spring-boot-starter-data-solr` “Starter” for collecting the dependencies in a convenient way.

Connecting to Solr

You can inject an auto-configured `SolrClient` instance as you would any other Spring bean. By default, the instance tries to connect to a server at `localhost:8983/solr`. The following example shows how to inject a Solr bean:

```
@Component
public class MyBean {

 private SolrClient solr;

 @Autowired
 public MyBean(SolrClient solr) {
  this.solr = solr;
 }

 // ...

}
```

If you add your own `@Bean` of type `SolrClient`, it replaces the default.

Spring Data Solr Repositories

Spring Data includes repository support for Apache Solr. As with the JPA repositories discussed earlier, the basic principle is that queries are automatically constructed for \ you based on method names.

In fact, both Spring Data JPA and Spring Data Solr share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now a `@SolrDocument` class rather than a JPA `@Entity`, it works in the same way.

> **Tip**
>
> For complete details of Spring Data Solr, refer to the reference documentation.

30.6 Elasticsearch

Elasticsearch is an open source, distributed, real-time search and analytics engine. Spring Boot offers basic auto-configuration for Elasticsearch and the abstractions on top of it provided by Spring Data Elasticsearch. There is a `spring-boot-starter-data-elasticsearch` “Starter” for collecting the dependencies in a convenient way. Spring Boot also supports Jest.

Connecting to Elasticsearch by Using Jest

If you have `Jest` on the classpath, you can inject an auto-configured `JestClient` that by default targets `localhost:9200`. You can further tune how the client is configured, as shown in the following example:

```
spring.elasticsearch.jest.uris=http://search.example.com:9200
spring.elasticsearch.jest.read-timeout=10000
spring.elasticsearch.jest.username=user
spring.elasticsearch.jest.password=secret
```

You can also register an arbitrary number of beans that implement `HttpClientConfigBuilderCustomizer` for more advanced customizations. The following example tunes additional HTTP settings:

```
static class HttpSettingsCustomizer implements HttpClientConfigBuilderCustomizer {

 @Override
 public void customize(HttpClientConfig.Builder builder) {
  builder.maxTotalConnection(100).defaultMaxTotalConnectionPerRoute(5);
 }

}
```

To take full control over the registration, define a `JestClient` bean.

Connecting to Elasticsearch by Using Spring Data

To connect to Elasticsearch, you must provide the address of one or more cluster nodes. The address can be specified by setting the `spring.data.elasticsearch.cluster-nodes` property to a comma-separated `host:port` list. With this configuration in place, an `ElasticsearchTemplate` or `TransportClient` can be injected like any other Spring bean, as shown in the following example:

```
spring.data.elasticsearch.cluster-nodes=localhost:9300
```

```
@Component
public class MyBean {

 private final ElasticsearchTemplate template;

 public MyBean(ElasticsearchTemplate template) {
  this.template = template;
 }
 // ...

}
```

If you add your own `ElasticsearchTemplate` or `TransportClient` `@Bean`, it replaces the default.

Spring Data Elasticsearch Repositories

Spring Data includes repository support for Elasticsearch. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

In fact, both Spring Data JPA and Spring Data Elasticsearch share the same common infrastructure. You could take the JPA example from earlier and, assuming that `City` is now an Elasticsearch `@Document` class rather than a JPA `@Entity`, it works in the same way.

Tip

For complete details of Spring Data Elasticsearch, refer to the reference documentation.

30.7 Cassandra

Cassandra is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and the abstractions on top of it provided by Spring Data Cassandra. There is a `spring-boot-starter-data-cassandra` "Starter" for collecting the dependencies in a convenient way.

Connecting to Cassandra

You can inject an auto-configured `CassandraTemplate` or a Cassandra `Session` instance as you would with any other Spring Bean. The `spring.data.cassandra.*` properties can be used to customize the connection. Generally, you provide `keyspace-name` and `contact-points` properties, as shown in the following example:

```
spring.data.cassandra.keyspace-name=mykeyspace
spring.data.cassandra.contact-points=cassandrahost1,cassandrahost2
```

The following code listing shows how to inject a Cassandra bean:

```
@Component
public class MyBean {

 private CassandraTemplate template;

 @Autowired
 public MyBean(CassandraTemplate template) {
  this.template = template;
 }

 // ...

}
```

If you add your own `@Bean` of type `CassandraTemplate`, it replaces the default.

Spring Data Cassandra Repositories

Spring Data includes basic repository support for Cassandra. Currently, this is more limited than the JPA repositories discussed earlier and needs to annotate finder methods with `@Query`.

> **Tip**
>
> For complete details of Spring Data Cassandra, refer to the reference documentation.

30.8 Couchbase

Couchbase is an open-source, distributed, multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and the abstractions on top of it provided by Spring Data Couchbase. There are `spring-boot-starter-data-couchbase` and `spring-boot-starter-data-couchbase-reactive` “Starters” for collecting the dependencies in a convenient way.

Connecting to Couchbase

You can get a `Bucket` and `Cluster` by adding the Couchbase SDK and some configuration. The `spring.couchbase.*` properties can be used to customize the connection. Generally, you provide the bootstrap hosts, bucket name, and password, as shown in the following example:

```
spring.couchbase.bootstrap-hosts=my-host-1,192.168.1.123
spring.couchbase.bucket.name=my-bucket
spring.couchbase.bucket.password=secret
```

> **Tip**
>
> You need to provide *at least* the bootstrap host(s), in which case the bucket name is `default` and the password is an empty String. Alternatively, you can define your own `org.springframework.data.couchbase.config.CouchbaseConfigurer` `@Bean` to take control over the whole configuration.

It is also possible to customize some of the `CouchbaseEnvironment` settings. For instance, the following configuration changes the timeout to use to open a new `Bucket` and enables SSL support:

```
spring.couchbase.env.timeouts.connect=3000
spring.couchbase.env.ssl.key-store=/location/of/keystore.jks
spring.couchbase.env.ssl.key-store-password=secret
```

Check the `spring.couchbase.env.*` properties for more details.

Spring Data Couchbase Repositories

Spring Data includes repository support for Couchbase. For complete details of Spring Data Couchbase, refer to the reference documentation.

You can inject an auto-configured `CouchbaseTemplate` instance as you would with any other Spring Bean, provided a *default* `CouchbaseConfigurer` is available (which happens when you enable Couchbase support, as explained earlier).

The following examples shows how to inject a Couchbase bean:

```
@Component
public class MyBean {

 private final CouchbaseTemplate template;

 @Autowired
 public MyBean(CouchbaseTemplate template) {
  this.template = template;
 }

 // ...

}
```

There are a few beans that you can define in your own configuration to override those provided by the auto-configuration:

- A `CouchbaseTemplate` `@Bean` with a name of `couchbaseTemplate`.
- An `IndexManager` `@Bean` with a name of `couchbaseIndexManager`.
- A `CustomConversions` `@Bean` with a name of `couchbaseCustomConversions`.

To avoid hard-coding those names in your own config, you can reuse `BeanNames` provided by Spring Data Couchbase. For instance, you can customize the converters to use, as follows:

```
@Configuration
public class SomeConfiguration {

 @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
 public CustomConversions myCustomConversions() {
  return new CustomConversions(...);
 }

 // ...

}
```

Tip

If you want to fully bypass the auto-configuration for Spring Data Couchbase, provide your own implementation of `org.springframework.data.couchbase.config.AbstractCouchbaseDataConfiguration`.

30.9 LDAP

LDAP (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. Spring Boot offers auto-configuration for any compliant LDAP server as well as support for the embedded in-memory LDAP server from UnboundID.

LDAP abstractions are provided by Spring Data LDAP. There is a `spring-boot-starter-data-ldap` “Starter” for collecting the dependencies in a convenient way.

Connecting to an LDAP Server

To connect to an LDAP server, make sure you declare a dependency on the `spring-boot-starter-data-ldap` “Starter” or `spring-ldap-core` and then declare the URLs of your server in your application.properties, as shown in the following example:

```
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

If you need to customize connection settings, you can use the `spring.ldap.base` and `spring.ldap.base-environment` properties.

Spring Data LDAP Repositories

Spring Data includes repository support for LDAP. For complete details of Spring Data LDAP, refer to the reference documentation.

You can also inject an auto-configured `LdapTemplate` instance as you would with any other Spring Bean, as shown in the following example:

```
@Component
public class MyBean {

 private final LdapTemplate template;

 @Autowired
 public MyBean(LdapTemplate template) {
  this.template = template;
 }

 // ...

}
```

Embedded In-memory LDAP Server

For testing purposes, Spring Boot supports auto-configuration of an in-memory LDAP server from UnboundID. To configure the server, add a dependency to `com.unboundid:unboundid-ldapsdk` and declare a `base-dn` property, as follows:

```
spring.ldap.embedded.base-dn=dc=spring,dc=io
```

By default, the server starts on a random port and triggers the regular LDAP support. There is no need to specify a `spring.ldap.urls` property.

If there is a `schema.ldif` file on your classpath, it is used to initialize the server. If you want to load the initialization script from a different resource, you can also use the `spring.ldap.embedded.ldif` property.

By default, a standard schema is used to validate `LDIF` files. You can turn off validation altogether by setting the `spring.ldap.embedded.validation.enabled` property. If you have custom attributes, you can use `spring.ldap.embedded.validation.schema` to define your custom attribute types or object classes.

30.10 InfluxDB

InfluxDB is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet-of-Things sensor data, and real-time analytics.

Connecting to InfluxDB

Spring Boot auto-configures an `InfluxDB` instance, provided the `influxdb-java` client is on the classpath and the URL of the database is set, as shown in the following example:

```
spring.influx.url=http://172.0.0.1:8086
```

If the connection to InfluxDB requires a user and password, you can set the `spring.influx.user` and `spring.influx.password` properties accordingly.

InfluxDB relies on OkHttp. If you need to tune the http client `InfluxDB` uses behind the scenes, you can register an `OkHttpClient.Builder` bean.

31. Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, thus reducing the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache infrastructure as long as caching support is enabled via the `@EnableCaching` annotation.

Note

Check the relevant section of the Spring Framework reference for more details.

In a nutshell, adding caching to an operation of your service is as easy as adding the relevant annotation to its method, as shown in the following example:

```
   import org.springframework.cache.annotation.Cacheable
import org.springframework.stereotype.Component;

@Component
public class MathService {

 @Cacheable("piDecimals")
 public int computePiDecimal(int i) {
  // ...
 }

}
```

This example demonstrates the use of caching on a potentially costly operation. Before invoking `computePiDecimal`, the abstraction looks for an entry in the `piDecimals` cache that matches the `i` argument. If an entry is found, the content in the cache is immediately returned to the caller, and the method is not invoked. Otherwise, the method is invoked, and the cache is updated before returning the value.

Caution

You can also use the standard JSR-107 (JCache) annotations (such as `@CacheResult`) transparently. However, we strongly advise you to not mix and match the Spring Cache and JCache annotations.

If you do not add any specific cache library, Spring Boot auto-configures a simple provider that uses concurrent maps in memory. When a cache is required (such as `piDecimals` in the preceding example), this provider creates it for you. The simple provider is not really recommended for production usage, but it is great for getting started and making sure that you understand the features. When you have made up your mind about the cache provider to use, please make sure to read its documentation to figure out how to configure the caches that your application uses. Nearly all providers require you to explicitly configure every cache that you use in the application. Some offer a way to customize the default caches defined by the `spring.cache.cache-names` property.

Tip

It is also possible to transparently update or evict data from the cache.

> **Note**
>
> If you use the cache infrastructure with beans that are not interface-based, make sure to enable the `proxyTargetClass` attribute of `@EnableCaching`.

31.1 Supported Cache Providers

The cache abstraction does not provide an actual store and relies on abstraction materialized by the `org.springframework.cache.Cache` and `org.springframework.cache.CacheManager` interfaces.

If you have not defined a bean of type `CacheManager` or a `CacheResolver` named `cacheResolver` (see `CachingConfigurer`), Spring Boot tries to detect the following providers (in the indicated order):

1. Generic
2. JCache (JSR-107) (EhCache 3, Hazelcast, Infinispan, and others)
3. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Couchbase
7. Redis
8. Caffeine
9. Simple

> **Tip**
>
> It is also possible to *force* a particular cache provider by setting the `spring.cache.type` property. Use this property if you need to disable caching altogether in certain environment (such as tests).

> **Tip**
>
> Use the `spring-boot-starter-cache` “Starter” to quickly add basic caching dependencies. The starter brings in `spring-context-support`. If you add dependencies manually, you must include `spring-context-support` in order to use the JCache, EhCache 2.x, or Guava support.

If the `CacheManager` is auto-configured by Spring Boot, you can further tune its configuration before it is fully initialized by exposing a bean that implements the `CacheManagerCustomizer` interface. The following example sets a flag to say that null values should be passed down to the underlying map:

```
@Bean
public CacheManagerCustomizer<ConcurrentMapCacheManager> cacheManagerCustomizer() {
 return new CacheManagerCustomizer<ConcurrentMapCacheManager>() {
  @Override
  public void customize(ConcurrentMapCacheManager cacheManager) {
   cacheManager.setAllowNullValues(false);
  }
 };
}
```

Note

In the preceding example, an auto-configured `ConcurrentMapCacheManager` is expected. If that is not the case (either you provided your own config or a different cache provider was auto-configured), the customizer is not invoked at all. You can have as many customizers as you want, and you can also order them by using `@Order` or `Ordered`.

Generic

Generic caching is used if the context defines *at least* one `org.springframework.cache.Cache` bean. A `CacheManager` wrapping all beans of that type is created.

JCache (JSR-107)

JCache is bootstrapped through the presence of a `javax.cache.spi.CachingProvider` on the classpath (that is, a JSR-107 compliant caching library exists on the classpath), and the `JCacheCacheManager` is provided by the `spring-boot-starter-cache` “Starter”. Various compliant libraries are available, and Spring Boot provides dependency management for Ehcache 3, Hazelcast, and Infinispan. Any other compliant library can be added as well.

It might happen that more than one provider is present, in which case the provider must be explicitly specified. Even if the JSR-107 standard does not enforce a standardized way to define the location of the configuration file, Spring Boot does its best to accommodate setting a cache with implementation details, as shown in the following example:

```
   # Only necessary if more than one provider is present
spring.cache.jcache.provider=com.acme.MyCachingProvider
spring.cache.jcache.config=classpath:acme.xml
```

Note

When a cache library offers both a native implementation and JSR-107 support, Spring Boot prefers the JSR-107 support, so that the same features are available if you switch to a different JSR-107 implementation.

Tip

Spring Boot has general support for Hazelcast. If a single `HazelcastInstance` is available, it is automatically reused for the `CacheManager` as well, unless the `spring.cache.jcache.config` property is specified.

There are two ways to customize the underlying `javax.cache.cacheManager`:

- Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `javax.cache.configuration.Configuration` bean is defined, it is used to customize them.
- `org.springframework.boot.autoconfigure.cache.JCacheManagerCustomizer` beans are invoked with the reference of the `CacheManager` for full customization.

Tip

If a standard `javax.cache.CacheManager` bean is defined, it is wrapped automatically in an `org.springframework.cache.CacheManager` implementation that the abstraction expects. No further customization is applied to it.

EhCache 2.x

EhCache 2.x is used if a file named `ehcache.xml` can be found at the root of the classpath. If EhCache 2.x is found, the `EhCacheCacheManager` provided by the `spring-boot-starter-cache` “Starter” is used to bootstrap the cache manager. An alternate configuration file can be provided as well, as shown in the following example:

```
spring.cache.ehcache.config=classpath:config/another-config.xml
```

Hazelcast

Spring Boot has general support for Hazelcast. If a `HazelcastInstance` has been auto-configured, it is automatically wrapped in a `CacheManager`.

Infinispan

Infinispan has no default configuration file location, so it must be specified explicitly. Otherwise, the default bootstrap is used.

```
spring.cache.infinispan.config=infinispan.xml
```

Caches can be created on startup by setting the `spring.cache.cache-names` property. If a custom `ConfigurationBuilder` bean is defined, it is used to customize the caches.

Note

The support of Infinispan in Spring Boot is restricted to the embedded mode and is quite basic. If you want more options, you should use the official Infinispan Spring Boot starter instead. See Infinispan’s documentation for more details.

Couchbase

If the Couchbase Java client and the `couchbase-spring-cache` implementation are available and Couchbase is configured, a `CouchbaseCacheManager` is auto-configured. It is also possible to create additional caches on startup by setting the `spring.cache.cache-names` property. These caches operate on the `Bucket` that was auto-configured. You can *also* create additional caches on another `Bucket` by using the customizer. Assume you need two caches (`cache1` and `cache2`) on the "main" `Bucket` and one (`cache3`) cache with a custom time to live of 2 seconds on the “another” `Bucket`. You can create the first two caches through configuration, as follows:

```
spring.cache.cache-names=cache1,cache2
```

Then you can define a `@Configuration` class to configure the extra `Bucket` and the `cache3` cache, as follows:

```
@Configuration
public class CouchbaseCacheConfiguration {

 private final Cluster cluster;

 public CouchbaseCacheConfiguration(Cluster cluster) {
  this.cluster = cluster;
 }

 @Bean
 public Bucket anotherBucket() {
  return this.cluster.openBucket("another", "secret");
 }

 @Bean
 public CacheManagerCustomizer<CouchbaseCacheManager> cacheManagerCustomizer() {
  return c -> {
   c.prepareCache("cache3", CacheBuilder.newInstance(anotherBucket())
     .withExpiration(2));
  };
 }

}
```

This sample configuration reuses the `Cluster` that was created through auto-configuration.

Redis

If Redis is available and configured, a `RedisCacheManager` is auto-configured. It is possible to create additional caches on startup by setting the `spring.cache.cache-names` property and cache defaults can be configured by using `spring.cache.redis.*` properties. For instance, the following configuration creates `cache1` and `cache2` caches with a *time to live* of 10 minutes:

```
spring.cache.cache-names=cache1,cache2
spring.cache.redis.time-to-live=600000
```

Note

By default, a key prefix is added so that, if two separate caches use the same key, Redis does not have overlapping keys and cannot return invalid values. We strongly recommend keeping this setting enabled if you create your own `RedisCacheManager`.

Tip

You can take full control of the configuration by adding a `RedisCacheConfiguration` `@Bean` of your own. This can be useful if you're looking for customizing the serialization strategy.

Caffeine

Caffeine is a Java 8 rewrite of Guava's cache that supersedes support for Guava. If Caffeine is present, a `CaffeineCacheManager` (provided by the `spring-boot-starter-cache` "Starter") is auto-configured. Caches can be created on startup by setting the `spring.cache.cache-names` property and can be customized by one of the following (in the indicated order):

1. A cache spec defined by `spring.cache.caffeine.spec`
2. A `com.github.benmanes.caffeine.cache.CaffeineSpec` bean is defined
3. A `com.github.benmanes.caffeine.cache.Caffeine` bean is defined

For instance, the following configuration creates `cache1` and `cache2` caches with a maximum size of 500 and a *time to live* of 10 minutes

```
spring.cache.cache-names=cache1,cache2
spring.cache.caffeine.spec=maximumSize=500,expireAfterAccess=600s
```

If a `com.github.benmanes.caffeine.cache.CacheLoader` bean is defined, it is automatically associated to the `CaffeineCacheManager`. Since the `CacheLoader` is going to be associated with *all* caches managed by the cache manager, it must be defined as `CacheLoader<Object, Object>`. The auto-configuration ignores any other generic type.

Simple

If none of the other providers can be found, a simple implementation using a `ConcurrentHashMap` as the cache store is configured. This is the default if no caching library is present in your application. By default, caches are created as needed, but you can restrict the list of available caches by setting the `cache-names` property. For instance, if you want only `cache1` and `cache2` caches, set the `cache-names` property as follows:

```
spring.cache.cache-names=cache1,cache2
```

If you do so and your application uses a cache not listed, then it fails at runtime when the cache is needed, but not on startup. This is similar to the way the "real" cache providers behave if you use an undeclared cache.

None

When `@EnableCaching` is present in your configuration, a suitable cache configuration is expected as well. If you need to disable caching altogether in certain environments, force the cache type to `none` to use a no-op implementation, as shown in the following example:

```
spring.cache.type=none
```

32. Messaging

The Spring Framework provides extensive support for integrating with messaging systems, from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the Advanced Message Queuing Protocol. Spring Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. Spring WebSocket natively includes support for STOMP messaging, and Spring Boot has support for that through starters and a small amount of auto-configuration. Spring Boot also has support for Apache Kafka.

32.1 JMS

The `javax.jms.ConnectionFactory` interface provides a standard method of creating a `javax.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally need not use it directly yourself and can instead rely on higher level messaging abstractions. (See the relevant section of the Spring Framework reference documentation for details.) Spring Boot also auto-configures the necessary infrastructure to send and receive messages.

ActiveMQ Support

When ActiveMQ is available on the classpath, Spring Boot can also configure a `ConnectionFactory`. If the broker is present, an embedded broker is automatically started and configured (provided no broker URL is specified through configuration).

Note

If you use `spring-boot-starter-activemq`, the necessary dependencies to connect or embed an ActiveMQ instance are provided, as is the Spring infrastructure to integrate with JMS.

ActiveMQ configuration is controlled by external configuration properties in `spring.activemq.*`. For example, you might declare the following section in `application.properties`:

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

You can also pool JMS resources by adding a dependency to `org.apache.activemq:activemq-pool` and configuring the `PooledConnectionFactory` accordingly, as shown in the following example:

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

Tip

See `ActiveMQProperties` for more of the supported options. You can also register an arbitrary number of beans that implement `ActiveMQConnectionFactoryCustomizer` for more advanced customizations.

By default, ActiveMQ creates a destination if it does not yet exist so that destinations are resolved against their provided names.

Artemis Support

Spring Boot can auto-configure a `ConnectionFactory` when it detects that Artemis is available on the classpath. If the broker is present, an embedded broker is automatically started and configured (unless the mode property has been explicitly set). The supported modes are `embedded` (to make explicit that an embedded broker is required and that an error should occur if the broker is not available on the classpath) and `native` (to connect to a broker using the `netty` transport protocol). When the latter is configured, Spring Boot configures a `ConnectionFactory` that connects to a broker running on the local machine with the default settings.

> **Note**
>
> If you use `spring-boot-starter-artemis`, the necessary dependencies to connect to an existing Artemis instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding `org.apache.activemq:artemis-jms-server` to your application lets you use embedded mode.

Artemis configuration is controlled by external configuration properties in `spring.artemis.*`. For example, you might declare the following section in `application.properties`:

```
spring.artemis.mode=native
spring.artemis.host=192.168.1.210
spring.artemis.port=9876
spring.artemis.user=admin
spring.artemis.password=secret
```

When embedding the broker, you can choose if you want to enable persistence and list the destinations that should be made available. These can be specified as a comma-separated list to create them with the default options, or you can define bean(s) of type `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` or `org.apache.activemq.artemis.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations, respectively.

See `ArtemisProperties` for more supported options.

No JNDI lookup is involved, and destinations are resolved against their names, using either the `name` attribute in the Artemis configuration or the names provided through configuration.

Using a JNDI ConnectionFactory

If you are running your application in an application server, Spring Boot tries to locate a JMS `ConnectionFactory` by using JNDI. By default, the `java:/JmsXA` and `java:/XAConnectionFactory` location are checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location, as shown in the following example:

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

Sending a Message

Spring's `JmsTemplate` is auto-configured, and you can autowire it directly into your own beans, as shown in the following example:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final JmsTemplate jmsTemplate;

 @Autowired
 public MyBean(JmsTemplate jmsTemplate) {
  this.jmsTemplate = jmsTemplate;
 }

 // ...

}
```

Note

`JmsMessagingTemplate` can be injected in a similar manner. If a `DestinationResolver` or a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `JmsTemplate`.

Receiving a Message

When the JMS infrastructure is present, any bean can be annotated with `@JmsListener` to create a listener endpoint. If no `JmsListenerContainerFactory` has been defined, a default one is configured automatically. If a `DestinationResolver` or a `MessageConverter` beans is defined, it is associated automatically to the default factory.

By default, the default factory is transactional. If you run in an infrastructure where a `JtaTransactionManager` is present, it is associated to the listener container by default. If not, the `sessionTransacted` flag is enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding `@Transactional` on your listener method (or a delegate thereof). This ensures that the incoming message is acknowledged, once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

The following component creates a listener endpoint on the `someQueue` destination:

```
@Component
public class MyBean {

 @JmsListener(destination = "someQueue")
 public void processMessage(String content) {
  // ...
 }

}
```

Tip

See the Javadoc of `@EnableJms` for more details.

If you need to create more `JmsListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `DefaultJmsListenerContainerFactoryConfigurer` that you can use to initialize a `DefaultJmsListenerContainerFactory` with the same settings as the one that is auto-configured.

For instance, the following example exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class JmsConfiguration {

 @Bean
 public DefaultJmsListenerContainerFactory myFactory(
   DefaultJmsListenerContainerFactoryConfigurer configurer) {
  DefaultJmsListenerContainerFactory factory =
    new DefaultJmsListenerContainerFactory();
  configurer.configure(factory, connectionFactory());
  factory.setMessageConverter(myMessageConverter());
  return factory;
 }

}
```

Then you can use the factory in any `@JmsListener`-annotated method as follows:

```
@Component
public class MyBean {

 @JmsListener(destination = "someQueue", containerFactory="myFactory")
 public void processMessage(String content) {
  // ...
 }

}
```

32.2 AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. Spring Boot offers several conveniences for working with AMQP through RabbitMQ, including the `spring-boot-starter-amqp` “Starter”.

RabbitMQ support

RabbitMQ is a lightweight, reliable, scalable, and portable message broker based on the AMQP protocol. Spring uses `RabbitMQ` to communicate through the AMQP protocol.

RabbitMQ configuration is controlled by external configuration properties in `spring.rabbitmq.*`. For example, you might declare the following section in `application.properties`:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

See `RabbitProperties` for more of the supported options.

Tip

See Understanding AMQP, the protocol used by RabbitMQ for more details.

Sending a Message

Spring’s `AmqpTemplate` and `AmqpAdmin` are auto-configured, and you can autowire them directly into your own beans, as shown in the following example:

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

 private final AmqpAdmin amqpAdmin;
 private final AmqpTemplate amqpTemplate;

 @Autowired
 public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
  this.amqpAdmin = amqpAdmin;
  this.amqpTemplate = amqpTemplate;
 }

 // ...

}
```

> **Note**
>
> `RabbitMessagingTemplate` can be injected in a similar manner. If a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `AmqpTemplate`.

If necessary, any `org.springframework.amqp.core.Queue` that is defined as a bean is automatically used to declare a corresponding queue on the RabbitMQ instance.

To retry operations, you can enable retries on the `AmqpTemplate` (for example, in the event that the broker connection is lost). Retries are disabled by default.

Receiving a Message

When the Rabbit infrastructure is present, any bean can be annotated with `@RabbitListener` to create a listener endpoint. If no `RabbitListenerContainerFactory` has been defined, a default `SimpleRabbitListenerContainerFactory` is automatically configured and you can switch to a direct container using the `spring.rabbitmq.listener.type` property. If a `MessageConverter` or a `MessageRecoverer` bean is defined, it is automatically associated with the default factory.

The following sample component creates a listener endpoint on the `someQueue` queue:

```
@Component
public class MyBean {

 @RabbitListener(queues = "someQueue")
 public void processMessage(String content) {
  // ...
 }

}
```

> **Tip**
>
> See the Javadoc of `@EnableRabbit` for more details.

If you need to create more `RabbitListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `SimpleRabbitListenerContainerFactoryConfigurer` and a `DirectRabbitListenerContainerFactoryConfigurer` that you can use to initialize a `SimpleRabbitListenerContainerFactory` and a

`DirectRabbitListenerContainerFactory` with the same settings as the factories used by the auto-configuration.

> **Tip**
>
> It does not matter which container type you chose. Those two beans are exposed by the auto-configuration.

For instance, the following configuration class exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class RabbitConfiguration {

 @Bean
 public SimpleRabbitListenerContainerFactory myFactory(
   SimpleRabbitListenerContainerFactoryConfigurer configurer) {
  SimpleRabbitListenerContainerFactory factory =
    new SimpleRabbitListenerContainerFactory();
  configurer.configure(factory, connectionFactory);
  factory.setMessageConverter(myMessageConverter());
  return factory;
 }

}
```

Then you can use the factory in any `@RabbitListener`-annotated method, as follows:

```
@Component
public class MyBean {

 @RabbitListener(queues = "someQueue", containerFactory="myFactory")
 public void processMessage(String content) {
  // ...
 }

}
```

You can enable retries to handle situations where your listener throws an exception. By default, `RejectAndDontRequeueRecoverer` is used, but you can define a `MessageRecoverer` of your own. When retries are exhausted, the message is rejected and either dropped or routed to a dead-letter exchange if the broker is configured to do so. By default, retries are disabled.

> **Important**
>
> By default, if retries are disabled and the listener throws an exception, the delivery is retried indefinitely. You can modify this behavior in two ways: Set the `defaultRequeueRejected` property to `false` so that zero re-deliveries are attempted or throw an `AmqpRejectAndDontRequeueException` to signal the message should be rejected. The latter is the mechanism used when retries are enabled and the maximum number of delivery attempts is reached.

32.3 Apache Kafka Support

Apache Kafka is supported by providing auto-configuration of the `spring-kafka` project.

Kafka configuration is controlled by external configuration properties in `spring.kafka.*`. For example, you might declare the following section in `application.properties`:

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

> **Tip**
>
> To create a topic on startup, add a bean of type `NewTopic`. If the topic already exists, the bean is ignored.

See [`KafkaProperties`]() for more supported options.

Sending a Message

Spring's `KafkaTemplate` is auto-configured, and you can autowire it directly in your own beans, as shown in the following example:

```
@Component
public class MyBean {

 private final KafkaTemplate kafkaTemplate;

 @Autowired
 public MyBean(KafkaTemplate kafkaTemplate) {
  this.kafkaTemplate = kafkaTemplate;
 }

 // ...

}
```

> **Note**
>
> If a `RecordMessageConverter` bean is defined, it is automatically associated to the auto-configured `KafkaTemplate`.

Receiving a Message

When the Apache Kafka infrastructure is present, any bean can be annotated with `@KafkaListener` to create a listener endpoint. If no `KafkaListenerContainerFactory` has been defined, a default one is automatically configured with keys defined in `spring.kafka.listener.*`. Also, if a `RecordMessageConverter` bean is defined, it is automatically associated to the default factory.

The following component creates a listener endpoint on the `someTopic` topic:

```
@Component
public class MyBean {

 @KafkaListener(topics = "someTopic")
 public void processMessage(String content) {
  // ...
 }

}
```

Additional Kafka Properties

The properties supported by auto configuration are shown in [Appendix A, *Common application properties*](). Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties. Refer to the Apache Kafka documentation for details.

The first few of these properties apply to both producers and consumers but can be specified at the producer or consumer level if you wish to use different values for each. Apache Kafka designates properties with an importance of HIGH, MEDIUM, or LOW. Spring Boot auto-configuration supports all HIGH importance properties, some selected MEDIUM and LOW properties, and any properties that do not have a default value.

Only a subset of the properties supported by Kafka are available through the `KafkaProperties` class. If you wish to configure the producer or consumer with additional properties that are not directly supported, use the following properties:

```
spring.kafka.properties.prop.one=first
spring.kafka.admin.properties.prop.two=second
spring.kafka.consumer.properties.prop.three=third
spring,kafka.producer.properties.prop.four=fourth
```

This sets the common `prop.one` Kafka property to `first` (applies to producers, consumers and admins), the `prop.two` admin property to `second`, the `prop.three` consumer property to `third` and the `prop.four` producer property to `fourth`.

You can also configure the Spring Kafka `JsonDeserializer` as follows:

```
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.value.default.type=org.foo.Invoice
spring.kafka.consumer.properties.spring.json.trusted.packages=org.foo,org.bar
```

Similarly, you can disable the `JsonSerializer` default behavior of sending type information in headers:

```
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties.spring.json.add.type.headers=false
```

Important

Properties set in this way override any configuration item that Spring Boot explicitly supports.

33. Calling REST Services with `RestTemplate`

If you need to call remote REST services from your application, you can use the Spring Framework's `RestTemplate` class. Since `RestTemplate` instances often need to be customized before being used, Spring Boot does not provide any single auto-configured `RestTemplate` bean. It does, however, auto-configure a `RestTemplateBuilder`, which can be used to create `RestTemplate` instances when needed. The auto-configured `RestTemplateBuilder` ensures that sensible `HttpMessageConverters` are applied to `RestTemplate` instances.

The following code shows a typical example:

```
@Service
public class MyService {

 private final RestTemplate restTemplate;

 public MyBean(RestTemplateBuilder restTemplateBuilder) {
  this.restTemplate = restTemplateBuilder.build();
 }

 public Details someRestCall(String name) {
  return this.restTemplate.getForObject("/{name}/details", Details.class, name);
 }

}
```

Tip

`RestTemplateBuilder` includes a number of useful methods that can be used to quickly configure a `RestTemplate`. For example, to add BASIC auth support, you can use `builder.basicAuthorization("user", "password").build()`.

33.1 RestTemplate Customization

There are three main approaches to `RestTemplate` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `RestTemplateBuilder` and then call its methods as required. Each method call returns a new `RestTemplateBuilder` instance, so the customizations only affect this use of the builder.

To make an application-wide, additive customization, use a `RestTemplateCustomizer` bean. All such beans are automatically registered with the auto-configured `RestTemplateBuilder` and are applied to any templates that are built with it.

The following example shows a customizer that configures the use of a proxy for all hosts except `192.168.0.5`:

```
static class ProxyCustomizer implements RestTemplateCustomizer {

 @Override
 public void customize(RestTemplate restTemplate) {
  HttpHost proxy = new HttpHost("proxy.example.com");
  HttpClient httpClient = HttpClientBuilder.create()
    .setRoutePlanner(new DefaultProxyRoutePlanner(proxy) {

     @Override
     public HttpHost determineProxy(HttpHost target,
       HttpRequest request, HttpContext context)
         throws HttpException {
      if (target.getHostName().equals("192.168.0.5")) {
       return null;
      }
      return super.determineProxy(target, request, context);
     }

    }).build();
  restTemplate.setRequestFactory(
    new HttpComponentsClientHttpRequestFactory(httpClient));
 }

}
```

Finally, the most extreme (and rarely used) option is to create your own `RestTemplateBuilder` bean. Doing so switches off the auto-configuration of a `RestTemplateBuilder` and prevents any `RestTemplateCustomizer` beans from being used.

34. Calling REST Services with `WebClient`

If you have Spring WebFlux on your classpath, you can also choose to use `WebClient` to call remote REST services. Compared to `RestTemplate`, this client has a more functional feel and is fully reactive. You can create your own client instance with the builder, `WebClient.create()`. See the relevant section on WebClient.

Spring Boot creates and pre-configures such a builder for you. For example, client HTTP codecs are configured in the same fashion as the server ones (see WebFlux HTTP codecs auto-configuration).

The following code shows a typical example:

```
@Service
public class MyService {

 private final WebClient webClient;

 public MyBean(WebClient.Builder webClientBuilder) {
  this.webClient = webClientBuilder.baseUrl("http://example.org").build();
 }

 public Mono<Details> someRestCall(String name) {
  return this.webClient.get().url("/{name}/details", name)
      .retrieve().bodyToMono(Details.class);
 }

}
```

34.1 WebClient Customization

There are three main approaches to `WebClient` customization, depending on how broadly you want the customizations to apply.

To make the scope of any customizations as narrow as possible, inject the auto-configured `WebClient.Builder` and then call its methods as required. `WebClient.Builder` instances are stateful: Any change on the builder is reflected in all clients subsequently created with it. If you want to create several clients with the same builder, you can also consider cloning the builder with `WebClient.Builder other = builder.clone();`.

To make an application-wide, additive customization to all `WebClient.Builder` instances, you can declare `WebClientCustomizer` beans and change the `WebClient.Builder` locally at the point of injection.

Finally, you can fall back to the original API and use `WebClient.create()`. In that case, no auto-configuration or `WebClientCustomizer` is applied.

35. Validation

The method validation feature supported by Bean Validation 1.1 is automatically enabled as long as a JSR-303 implementation (such as Hibernate validator) is on the classpath. This lets bean methods be annotated with `javax.validation` constraints on their parameters and/or on their return value. Target classes with such annotated methods need to be annotated with the `@Validated` annotation at the type level for their methods to be searched for inline constraint annotations.

For instance, the following service triggers the validation of the first argument, making sure its size is between 8 and 10:

```
@Service
@Validated
public class MyBean {

 public Archive findByCodeAndAuthor(@Size(min = 8, max = 10) String code,
   Author author) {
  ...
 }

}
```

36. Sending Email

The Spring Framework provides an easy abstraction for sending email by using the `JavaMailSender` interface, and Spring Boot provides auto-configuration for it as well as a starter module.

> **Tip**
>
> See the reference documentation for a detailed explanation of how you can use `JavaMailSender`.

If `spring.mail.host` and the relevant libraries (as defined by `spring-boot-starter-mail`) are available, a default `JavaMailSender` is created if none exists. The sender can be further customized by configuration items from the `spring.mail` namespace. See `MailProperties` for more details.

In particular, certain default timeout values are infinite, and you may want to change that to avoid having a thread blocked by an unresponsive mail server, as shown in the following example:

```
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=3000
spring.mail.properties.mail.smtp.writetimeout=5000
```

37. Distributed Transactions with JTA

Spring Boot supports distributed JTA transactions across multiple XA resources by using either an Atomikos or Bitronix embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

When a JTA environment is detected, Spring's `JtaTransactionManager` is used to manage transactions. Auto-configured JMS, DataSource, and JPA beans are upgraded to support XA transactions. You can use standard Spring idioms, such as `@Transactional`, to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions, you can set the `spring.jta.enabled` property to `false` to disable the JTA auto-configuration.

37.1 Using an Atomikos Transaction Manager

Atomikos is a popular open source transaction manager which can be embedded into your Spring Boot application. You can use the `spring-boot-starter-jta-atomikos` Starter to pull in the appropriate Atomikos libraries. Spring Boot auto-configures Atomikos and ensures that appropriate `depends-on` settings are applied to your Spring beans for correct startup and shutdown ordering.

By default, Atomikos transaction logs are written to a `transaction-logs` directory in your application's home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a `spring.jta.log-dir` property in your `application.properties` file. Properties starting with `spring.jta.atomikos.properties` can also be used to customize the Atomikos `UserTransactionServiceImp`. See the `AtomikosProperties` Javadoc for complete details.

Note

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Atomikos instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Atomikos is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

37.2 Using a Bitronix Transaction Manager

Bitronix is a popular open-source JTA transaction manager implementation. You can use the `spring-boot-starter-jta-bitronix` starter to add the appropriate Bitronix dependencies to your project. As with Atomikos, Spring Boot automatically configures Bitronix and post-processes your beans to ensure that startup and shutdown ordering is correct.

By default, Bitronix transaction log files (`part1.btm` and `part2.btm`) are written to a `transaction-logs` directory in your application home directory. You can customize the location of this directory by setting the `spring.jta.log-dir` property. Properties starting with `spring.jta.bitronix.properties` are also bound to the `bitronix.tm.Configuration` bean, allowing for complete customization. See the Bitronix documentation for details.

Note

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Bitronix instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Bitronix is running. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

37.3 Using a Narayana Transaction Manager

Narayana is a popular open source JTA transaction manager implementation supported by JBoss. You can use the `spring-boot-starter-jta-narayana` starter to add the appropriate Narayana dependencies to your project. As with Atomikos and Bitronix, Spring Boot automatically configures Narayana and post-processes your beans to ensure that startup and shutdown ordering is correct.

By default, Narayana transaction logs are written to a `transaction-logs` directory in your application home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a `spring.jta.log-dir` property in your `application.properties` file. Properties starting with `spring.jta.narayana.properties` can also be used to customize the Narayana configuration. See the `NarayanaProperties` Javadoc for complete details.

Note

To ensure that multiple transaction managers can safely coordinate the same resource managers, each Narayana instance must be configured with a unique ID. By default, this ID is set to `1`. To ensure uniqueness in production, you should configure the `spring.jta.transaction-manager-id` property with a different value for each instance of your application.

37.4 Using a Java EE Managed Transaction Manager

If you package your Spring Boot application as a `war` or `ear` file and deploy it to a Java EE application server, you can use your application server's built-in transaction manager. Spring Boot tries to auto-configure a transaction manager by looking at common JNDI locations (`java:comp/UserTransaction`, `java:comp/TransactionManager`, and so on). If you use a transaction service provided by your application server, you generally also want to ensure that all resources are managed by the server and exposed over JNDI. Spring Boot tries to auto-configure JMS by looking for a `ConnectionFactory` at the JNDI path (`java:/JmsXA` or `java:/XAConnectionFactory`), and you can use the `spring.datasource.jndi-name` property to configure your `DataSource`.

37.5 Mixing XA and Non-XA JMS Connections

When using JTA, the primary JMS `ConnectionFactory` bean is XA-aware and participates in distributed transactions. In some situations, you might want to process certain JMS messages by using a non-XA `ConnectionFactory`. For example, your JMS processing logic might take longer than the XA timeout.

If you want to use a non-XA `ConnectionFactory`, you can inject the `nonXaJmsConnectionFactory` bean rather than the `@Primary jmsConnectionFactory` bean. For consistency, the `jmsConnectionFactory` bean is also provided by using the bean alias `xaJmsConnectionFactory`.

The following example shows how to inject `ConnectionFactory` instances:

```java
// Inject the primary (XA aware) ConnectionFactory
@Autowired
private ConnectionFactory defaultConnectionFactory;
```

```
// Inject the XA aware ConnectionFactory (uses the alias and injects the same as above)
@Autowired
@Qualifier("xaJmsConnectionFactory")
private ConnectionFactory xaConnectionFactory;

// Inject the non-XA aware ConnectionFactory
@Autowired
@Qualifier("nonXaJmsConnectionFactory")
private ConnectionFactory nonXaConnectionFactory;
```

37.6 Supporting an Alternative Embedded Transaction Manager

The `XAConnectionFactoryWrapper` and `XADataSourceWrapper` interfaces can be used to support alternative embedded transaction managers. The interfaces are responsible for wrapping `XAConnectionFactory` and `XADataSource` beans and exposing them as regular `ConnectionFactory` and `DataSource` beans, which transparently enroll in the distributed transaction. DataSource and JMS auto-configuration use JTA variants, provided you have a `JtaTransactionManager` bean and appropriate XA wrapper beans registered within your `ApplicationContext`.

The BitronixXAConnectionFactoryWrapper and BitronixXADataSourceWrapper provide good examples of how to write XA wrappers.

38. Hazelcast

If Hazelcast is on the classpath and a suitable configuration is found, Spring Boot auto-configures a `HazelcastInstance` that you can inject in your application.

If you define a `com.hazelcast.config.Config` bean, Spring Boot uses that. If your configuration defines an instance name, Spring Boot tries to locate an existing instance rather than creating a new one.

You could also specify the `hazelcast.xml` configuration file to use through configuration, as shown in the following example:

```
spring.hazelcast.config=classpath:config/my-hazelcast.xml
```

Otherwise, Spring Boot tries to find the Hazelcast configuration from the default locations: `hazelcast.xml` in the working directory or at the root of the classpath. We also check if the `hazelcast.config` system property is set. See the Hazelcast documentation for more details.

If `hazelcast-client` is present on the classpath, Spring Boot first attempts to create a client by checking the following configuration options:

- The presence of a `com.hazelcast.client.config.ClientConfig` bean.
- A configuration file defined by the `spring.hazelcast.config` property.
- The presence of the `hazelcast.client.config` system property.
- A `hazelcast-client.xml` in the working directory or at the root of the classpath.

Note

Spring Boot also has explicit caching support for Hazelcast. If caching is enabled, the `HazelcastInstance` is automatically wrapped in a `CacheManager` implementation.

39. Quartz Scheduler

Spring Boot offers several conveniences for working with the Quartz scheduler, including the `spring-boot-starter-quartz` “Starter”. If Quartz is available, a `Scheduler` is auto-configured (through the `SchedulerFactoryBean` abstraction).

Beans of the following types are automatically picked up and associated with the `Scheduler`:

- `JobDetail`: defines a particular Job. `JobDetail` instances can be built with the `JobBuilder` API.
- `Calendar`.
- `Trigger`: defines when a particular job is triggered.

By default, an in-memory `JobStore` is used. However, it is possible to configure a JDBC-based store if a `DataSource` bean is available in your application and if the `spring.quartz.job-store-type` property is configured accordingly, as shown in the following example:

```
spring.quartz.job-store-type=jdbc
```

When the JDBC store is used, the schema can be initialized on startup, as shown in the following example:

```
spring.quartz.jdbc.initialize-schema=always
```

Note

By default, the database is detected and initialized by using the standard scripts provided with the Quartz library. It is also possible to provide a custom script by setting the `spring.quartz.jdbc.schema` property.

Quartz Scheduler configuration can be customized by using Quartz configuration properties ()`spring.quartz.properties.*`) and `SchedulerFactoryBeanCustomizer` beans, which allow programmatic `SchedulerFactoryBean` customization.

Jobs can define setters to inject data map properties. Regular beans can also be injected in a similar manner, as shown in the following example:

```
public class SampleJob extends QuartzJobBean {

 private MyService myService;
 private String name;

 // Inject "MyService" bean
 public void setMyService(MyService myService) { ... }

 // Inject the "name" job data property
 public void setName(String name) { ... }

 @Override
 protected void executeInternal(JobExecutionContext context)
   throws JobExecutionException {
  ...
 }

}
```

40. Spring Integration

Spring Boot offers several conveniences for working with Spring Integration, including the `spring-boot-starter-integration` “Starter”. Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP, and others. If Spring Integration is available on your classpath, it is initialized through the `@EnableIntegration` annotation.

Spring Boot also configures some features that are triggered by the presence of additional Spring Integration modules. If `spring-integration-jmx` is also on the classpath, message processing statistics are published over JMX . If `spring-integration-jdbc` is available, the default database schema can be created on startup, as shown in the following line:

```
spring.integration.jdbc.initialize-schema=always
```

See the `IntegrationAutoConfiguration` and `IntegrationProperties` classes for more details.

41. Spring Session

Spring Boot provides Spring Session auto-configuration for a wide range of data stores. When building a Servlet web application, the following stores can be auto-configured:

- JDBC
- Redis
- Hazelcast
- MongoDB

When building a reactive web application, the following stores can be auto-configured:

- Redis
- MongoDB

If Spring Session is available, you must choose the `StoreType` that you wish to use to store the sessions. For instance, to use JDBC as the back-end store, you can configure your application as follows:

```
spring.session.store-type=jdbc
```

Tip

You can disable Spring Session by setting the `store-type` to `none`.

Each store has specific additional settings. For instance, it is possible to customize the name of the table for the JDBC store, as shown in the following example:

```
spring.session.jdbc.table-name=SESSIONS
```

42. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default, Spring Boot creates an `MBeanServer` bean with an ID of `mbeanServer` and exposes any of your beans that are annotated with Spring JMX annotations (`@ManagedResource`, `@ManagedAttribute`, or `@ManagedOperation`).

See the `JmxAutoConfiguration` class for more details.

43. Testing

Spring Boot provides a number of utilities and annotations to help when testing your application. Test support is provided by two modules: `spring-boot-test` contains core items, and `spring-boot-test-autoconfigure` supports auto-configuration for tests.

Most developers use the `spring-boot-starter-test` “Starter”, which imports both Spring Boot test modules as well as JUnit, AssertJ, Hamcrest, and a number of other useful libraries.

43.1 Test Scope Dependencies

The `spring-boot-starter-test` “Starter” (in the `test scope`)contains the following provided libraries:

- JUnit: The de-facto standard for unit testing Java applications.
- Spring Test & Spring Boot Test: Utilities and integration test support for Spring Boot applications.
- AssertJ: A fluent assertion library.
- Hamcrest: A library of matcher objects (also known as constraints or predicates).
- Mockito: A Java mocking framework.
- JSONassert: An assertion library for JSON.
- JsonPath: XPath for JSON.

We generally find these common libraries to be useful when writing tests. If these libraries do not suit your needs, you can add additional test dependencies of your own.

43.2 Testing Spring Applications

One of the major advantages of dependency injection is that it should make your code easier to unit test. You can instantiate objects by using the `new` operator without even involving Spring. You can also use *mock objects* instead of real dependencies.

Often, you need to move beyond unit testing and start integration testing (with a Spring `ApplicationContext`). It is useful to be able to perform integration testing without requiring deployment of your application or needing to connect to other infrastructure.

The Spring Framework includes a dedicated test module for such integration testing. You can declare a dependency directly to `org.springframework:spring-test` or use the `spring-boot-starter-test` “Starter” to pull it in transitively.

If you have not used the `spring-test` module before, you should start by reading the relevant section of the Spring Framework reference documentation.

43.3 Testing Spring Boot Applications

A Spring Boot application is a Spring `ApplicationContext`, so nothing very special has to be done to test it beyond what you would normally do with a vanilla Spring context.

> **Note**
>
> External properties, logging, and other features of Spring Boot are installed in the context by default only if you use `SpringApplication` to create it.

Spring Boot provides a `@SpringBootTest` annotation, which can be used as an alternative to the standard `spring-test` `@ContextConfiguration` annotation when you need Spring Boot features. The annotation works by creating the `ApplicationContext` used in your tests through `SpringApplication`.

You can use the `webEnvironment` attribute of `@SpringBootTest` to further refine how your tests run:

- `MOCK`: Loads a `WebApplicationContext` and provides a mock servlet environment. Embedded servlet containers are not started when using this annotation. If servlet APIs are not on your classpath, this mode transparently falls back to creating a regular non-web `ApplicationContext`. It can be used in conjunction with `@AutoConfigureMockMvc` for `MockMvc`-based testing of your application.
- `RANDOM_PORT`: Loads an `ServletWebServerApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listen on a random port.
- `DEFINED_PORT`: Loads a `ServletWebServerApplicationContext` and provides a real servlet environment. Embedded servlet containers are started and listen on a defined port (from your `application.properties` or on the default port of `8080`).
- `NONE`: Loads an `ApplicationContext` by using `SpringApplication` but does not provide *any* servlet environment (mock or otherwise).

> **Note**
>
> If your test is `@Transactional`, it rolls back the transaction at the end of each test method by default. However, as using this arrangement with either `RANDOM_PORT` or `DEFINED_PORT` implicitly provides a real servlet environment, the HTTP client and server run in separate threads and, thus, in separate transactions. Any transaction initiated on the server does not roll back in this case.

> **Note**
>
> In addition to `@SpringBootTest`, a number of other annotations are also provided for testing more specific slices of an application. You can find more detail throughout this chapter.

> **Tip**
>
> Do not forget to add `@RunWith(SpringRunner.class)` to your test. Otherwise, the annotations are ignored.

Detecting Test Configuration

If you are familiar with the Spring Test Framework, you may be used to using `@ContextConfiguration(classes=…)` in order to specify which Spring `@Configuration` to load. Alternatively, you might have often used nested `@Configuration` classes within your test.

When testing Spring Boot applications, this is often not required. Spring Boot's `@*Test` annotations search for your primary configuration automatically whenever you do not explicitly define one.

The search algorithm works up from the package that contains the test until it finds a class annotated with `@SpringBootApplication` or `@SpringBootConfiguration`. As long as you structured your code in a sensible way, your main configuration is usually found.

Note

If you use a test annotation to test a more specific slice of your application, you should avoid adding configuration settings that are specific to a particular area on the main method's application class.

If you want to customize the primary configuration, you can use a nested `@TestConfiguration` class. Unlike a nested `@Configuration` class, which would be used instead of your application's primary configuration, a nested `@TestConfiguration` class is used in addition to your application's primary configuration.

Note

Spring's test framework caches application contexts between tests. Therefore, as long as your tests share the same configuration (no matter how it is discovered), the potentially time-consuming process of loading the context happens only once.

Excluding Test Configuration

If your application uses component scanning (for example, if you use `@SpringBootApplication` or `@ComponentScan`), you may find top-level configuration classes that you created only for specific tests accidentally get picked up everywhere.

As we have seen earlier, `@TestConfiguration` can be used on an inner class of a test to customize the primary configuration. When placed on a top-level class, `@TestConfiguration` indicates that classes in `src/test/java` should not be picked up by scanning. You can then import that class explicitly where it is required, as shown in the following example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Import(MyTestsConfiguration.class)
public class MyTests {

 @Test
 public void exampleTest() {
  ...
 }

}
```

Note

If you directly use `@ComponentScan` (that is, not through `@SpringBootApplication`) you need to register the `TypeExcludeFilter` with it. See the Javadoc for details.

Working with Random Ports

If you need to start a full running server for tests, we recommend that you use random ports. If you use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)`, an available port is picked at random each time your test runs.

The `@LocalServerPort` annotation can be used to inject the actual port used into your test. For convenience, tests that need to make REST calls to the started server can additionally `@Autowire` a `TestRestTemplate`, which resolves relative links to the running server, as shown in the following example:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.test.context.junit4.SpringRunner;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortTestRestTemplateExampleTests {

 @Autowired
 private TestRestTemplate restTemplate;

 @Test
 public void exampleTest() {
  String body = this.restTemplate.getForObject("/", String.class);
  assertThat(body).isEqualTo("Hello World");
 }

}
```

If you prefer to use a `WebTestClient`, you can use that as well:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class RandomPortWebTestClientExampleTests {

 @Autowired
 private WebTestClient webClient;

 @Test
 public void exampleTest() {
  this.webClient.get().uri("/").exchange().expectStatus().isOk()
    .expectBody(String.class).isEqualTo("Hello World");
 }

}
```

Mocking and Spying Beans

When running tests, it is sometimes necessary to mock certain components within your application context. For example, you may have a facade over some remote service that is unavailable during development. Mocking can also be useful when you want to simulate failures that might be hard to trigger in a real environment.

Spring Boot includes a `@MockBean` annotation that can be used to define a Mockito mock for a bean inside your `ApplicationContext`. You can use the annotation to add new beans or replace a single existing bean definition. The annotation can be used directly on test classes, on fields within your test, or on `@Configuration` classes and fields. When used on a field, the instance of the created mock is also injected. Mock beans are automatically reset after each test method.

> **Note**
>
> If your test uses one of Spring Boot's test annotations (such as `@SpringBootTest`), this feature is automatically enabled. To use this feature with a different arrangement, a listener must be explicitly added, as shown in the following example:
>
> ```
> @TestExecutionListeners(MockitoTestExecutionListener.class)
> ```

The following example replaces an existing `RemoteService` bean with a mock implementation:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.mock.mockito.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

 @MockBean
 private RemoteService remoteService;

 @Autowired
 private Reverser reverser;

 @Test
 public void exampleTest() {
  // RemoteService has been injected into the reverser bean
  given(this.remoteService.someCall()).willReturn("mock");
  String reverse = reverser.reverseSomeCall();
  assertThat(reverse).isEqualTo("kcom");
 }

}
```

Additionally, you can use `@SpyBean` to wrap any existing bean with a Mockito `spy`. See the Javadoc for full details.

Auto-configured Tests

Spring Boot's auto-configuration system works well for applications but can sometimes be a little too much for tests. It often helps to load only the parts of the configuration that are required to test a "slice" of your application. For example, you might want to test that Spring MVC controllers are mapping URLs correctly, and you do not want to involve database calls in those tests, or you might want to test JPA entities, and you are not interested in the web layer when those tests run.

The `spring-boot-test-autoconfigure` module includes a number of annotations that can be used to automatically configure such "slices". Each of them works in a similar way, providing a `@…Test` annotation that loads the `ApplicationContext` and one or more `@AutoConfigure…` annotations that can be used to customize auto-configuration settings.

Note

Each slice loads a very restricted set of auto-configuration classes. If you need to exclude one of them, most `@…Test` annotations provide an `excludeAutoConfiguration` attribute. Alternatively, you can use `@ImportAutoConfiguration#exclude`.

Tip

It is also possible to use the `@AutoConfigure…` annotations with the standard `@SpringBootTest` annotation. You can use this combination if you are not interested in “slicing” your application but you want some of the auto-configured test beans.

Auto-configured JSON Tests

To test that object JSON serialization and deserialization is working as expected, you can use the `@JsonTest` annotation. `@JsonTest` auto-configures the available supported JSON mapper, which can be one of the following libraries:

- Jackson `ObjectMapper`, any `@JsonComponent` beans and any Jackson `Modules`
- `Gson`
- `Jsonb`

If you need to configure elements of the auto-configuration, you can use the `@AutoConfigureJsonTesters` annotation.

Spring Boot includes AssertJ-based helpers that work with the JSONassert and JsonPath libraries to check that JSON appears as expected. The `JacksonTester`, `GsonTester`, `JsonbTester`, and `BasicJsonTester` classes can be used for Jackson, Gson, Jsonb, and Strings respectively. Any helper fields on the test class can be `@Autowired` when using `@JsonTest`. The following example shows a test class for Jackson:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.json.*;
import org.springframework.boot.test.context.*;
import org.springframework.boot.test.json.*;
import org.springframework.test.context.junit4.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

 @Autowired
 private JacksonTester<VehicleDetails> json;

 @Test
 public void testSerialize() throws Exception {
  VehicleDetails details = new VehicleDetails("Honda", "Civic");
  // Assert against a `.json` file in the same package as the test
  assertThat(this.json.write(details)).isEqualToJson("expected.json");
  // Or use JSON path based assertions
```

```
    assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
    assertThat(this.json.write(details)).extractingJsonPathStringValue("@.make")
      .isEqualTo("Honda");
  }

  @Test
  public void testDeserialize() throws Exception {
    String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
    assertThat(this.json.parse(content))
      .isEqualTo(new VehicleDetails("Ford", "Focus"));
    assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
  }

}
```

Note

JSON helper classes can also be used directly in standard unit tests. To do so, call the `initFields` method of the helper in your `@Before` method if you do not use `@JsonTest`.

A list of the auto-configuration that is enabled by `@JsonTest` can be found in the appendix.

Auto-configured Spring MVC Tests

To test whether Spring MVC controllers are working as expected, use the `@WebMvcTest` annotation. `@WebMvcTest` auto-configures the Spring MVC infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `WebMvcConfigurer`, and `HandlerMethodArgumentResolver`. Regular `@Component` beans are not scanned when using this annotation.

Tip

If you need to register extra components, such as the Jackson `Module`, you can import additional configuration classes by using `@Import` on your test.

Often, `@WebMvcTest` is limited to a single controller and is used in combination with `@MockBean` to provide mock implementations for required collaborators.

`@WebMvcTest` also auto-configures `MockMvc`. Mock MVC offers a powerful way to quickly test MVC controllers without needing to start a full HTTP server.

Tip

You can also auto-configure `MockMvc` in a non-`@WebMvcTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureMockMvc`. The following example uses `MockMvc`:

```
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
```

```
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

 @Autowired
 private MockMvc mvc;

 @MockBean
 private UserVehicleService userVehicleService;

 @Test
 public void testExample() throws Exception {
  given(this.userVehicleService.getVehicleDetails("sboot"))
    .willReturn(new VehicleDetails("Honda", "Civic"));
  this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
    .andExpect(status().isOk()).andExpect(content().string("Honda Civic"));
 }

}
```

Tip

If you need to configure elements of the auto-configuration (for example, when servlet filters should be applied) you can use attributes in the `@AutoConfigureMockMvc` annotation.

If you use HtmlUnit or Selenium, auto-configuration also provides an HTMLUnit `WebClient` bean and/or a `WebDriver` bean. The following example uses HtmlUnit:

```
import com.gargoylesoftware.htmlunit.*;
import org.junit.*;
import org.junit.runner.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.test.autoconfigure.web.servlet.*;
import org.springframework.boot.test.mock.mockito.*;

import static org.assertj.core.api.Assertions.*;
import static org.mockito.BDDMockito.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

 @Autowired
 private WebClient webClient;

 @MockBean
 private UserVehicleService userVehicleService;

 @Test
 public void testExample() throws Exception {
  given(this.userVehicleService.getVehicleDetails("sboot"))
    .willReturn(new VehicleDetails("Honda", "Civic"));
  HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
  assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
 }

}
```

Note

By default, Spring Boot puts `WebDriver` beans in a special “scope” to ensure that the driver exits after each test and that a new instance is injected. If you do not want this behavior, you can add `@Scope("singleton")` to your `WebDriver` `@Bean` definition.

A list of the auto-configuration settings that are enabled by `@WebMvcTest` can be found in the appendix.

Auto-configured Spring WebFlux Tests

To test that Spring WebFlux controllers are working as expected, you can use the `@WebFluxTest` annotation. `@WebFluxTest` auto-configures the Spring WebFlux infrastructure and limits scanned beans to `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, and `WebFluxConfigurer`. Regular `@Component` beans are not scanned when the `@WebFluxTest` annotation is used.

Tip

If you need to register extra components, such as Jackson `Module`, you can import additional configuration classes using `@Import` on your test.

Often, `@WebFluxTest` is limited to a single controller and used in combination with the `@MockBean` annotation to provide mock implementations for required collaborators.

`@WebFluxTest` also auto-configures `WebTestClient`, which offers a powerful way to quickly test WebFlux controllers without needing to start a full HTTP server.

Tip

You can also auto-configure `WebTestClient` in a non-`@WebFluxTest` (such as `@SpringBootTest`) by annotating it with `@AutoConfigureWebTestClient`. The following example shows a class that uses both `@WebFluxTest` and a `WebTestClient`:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.reactive.WebFluxTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.reactive.server.WebTestClient;

@RunWith(SpringRunner.class)
@WebFluxTest(UserVehicleController.class)
public class MyControllerTests {

 @Autowired
 private WebTestClient webClient;

 @MockBean
 private UserVehicleService userVehicleService;

 @Test
 public void testExample() throws Exception {
  given(this.userVehicleService.getVehicleDetails("sboot"))
    .willReturn(new VehicleDetails("Honda", "Civic"));
  this.webClient.get().uri("/sboot/vehicle").accept(MediaType.TEXT_PLAIN)
    .exchange()
    .expectStatus().isOk()
    .expectBody(String.class).isEqualTo("Honda Civic");
 }

}
```

A list of the auto-configuration that is enabled by `@WebFluxTest` can be found in the appendix.

Auto-configured Data JPA Tests

You can use the `@DataJpaTest` annotation to test JPA applications. By default, it configures an in-memory embedded database, scans for `@Entity` classes, and configures Spring Data JPA repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`.

By default, data JPA tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class as follows:

```java
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataJpaTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

Data JPA tests may also inject a `TestEntityManager` bean, which provides an alternative to the standard JPA `EntityManager` that is specifically designed for tests. If you want to use `TestEntityManager` outside of `@DataJpaTest` instances, you can also use the `@AutoConfigureTestEntityManager` annotation. A `JdbcTemplate` is also available if you need that. The following example shows the `@DataJpaTest` annotation in use:

```java
import org.junit.*;
import org.junit.runner.*;
import org.springframework.boot.test.autoconfigure.orm.jpa.*;

import static org.assertj.core.api.Assertions.*;

@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

 @Autowired
 private TestEntityManager entityManager;

 @Autowired
 private UserRepository repository;

 @Test
 public void testExample() throws Exception {
  this.entityManager.persist(new User("sboot", "1234"));
  User user = this.repository.findByUsername("sboot");
  assertThat(user.getUsername()).isEqualTo("sboot");
  assertThat(user.getVin()).isEqualTo("1234");
 }

}
```

In-memory embedded databases generally work well for tests, since they are fast and do not require any installation. If, however, you prefer to run tests against a real database you can use the `@AutoConfigureTestDatabase` annotation, as shown in the following example:

```java
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class ExampleRepositoryTests {
 // ...

}
```

A list of the auto-configuration settings that are enabled by `@DataJpaTest` can be found in the appendix.

Auto-configured JDBC Tests

`@JdbcTest` is similar to `@DataJpaTest` but is for pure JDBC-related tests. By default, it also configures an in-memory embedded database and a `JdbcTemplate`. Regular `@Component` beans are not loaded into the `ApplicationContext`.

By default, JDBC tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jdbc.JdbcTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@JdbcTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

If you prefer your test to run against a real database, you can use the `@AutoConfigureTestDatabase` annotation in the same way as for `DataJpaTest`. (See "the section called “Auto-configured Data JPA Tests”".)

A list of the auto-configuration that is enabled by `@JdbcTest` can be found in the appendix.

Auto-configured jOOQ Tests

You can use `@JooqTest` in a similar fashion as `@JdbcTest` but for jOOQ-related tests. As jOOQ relies heavily on a Java-based schema that corresponds with the database schema, the existing `DataSource` is used. If you want to replace it with an in-memory database, you can use `@AutoconfigureTestDatabase` to override those settings. (For more about using jOOQ with Spring Boot, see "Section 29.5, “Using jOOQ”", earlier in this chapter.)

`@JooqTest` configures a `DSLContext`. Regular `@Component` beans are not loaded into the `ApplicationContext`. The following example shows the `@JooqTest` annotation in use:

```
import org.jooq.DSLContext;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.jooq.JooqTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@JooqTest
public class ExampleJooqTests {

 @Autowired
 private DSLContext dslContext;
}
```

JOOQ tests are transactional and roll back at the end of each test by default. If that is not what you want, you can disable transaction management for a test or for the whole test class as shown in the JDBC example.

A list of the auto-configuration that is enabled by `@JooqTest` can be found in the appendix.

Auto-configured Data MongoDB Tests

You can use `@DataMongoTest` to test MongoDB applications. By default, it configures an in-memory embedded MongoDB (if available), configures a `MongoTemplate`, scans for `@Document` classes, and configures Spring Data MongoDB repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using MongoDB with Spring Boot, see "Section 30.2, “MongoDB”", earlier in this chapter.)

The following class shows the `@DataMongoTest` annotation in use:

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest
public class ExampleDataMongoTests {

 @Autowired
 private MongoTemplate mongoTemplate;

 //
}
```

In-memory embedded MongoDB generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run tests against a real MongoDB server, you should exclude the embedded MongoDB auto-configuration, as shown in the following example:

```
import org.junit.runner.RunWith;
 import org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class)
public class ExampleDataMongoNonEmbeddedTests {

}
```

A list of the auto-configuration settings that are enabled by `@DataMongoTest` can be found in the appendix.

Auto-configured Data Neo4j Tests

You can use `@DataNeo4jTest` to test Neo4j applications. By default, it uses an in-memory embedded Neo4j (if the embedded driver is available), scans for `@NodeEntity` classes, and configures Spring Data Neo4j repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using Neo4J with Spring Boot, see "Section 30.3, “Neo4j”", earlier in this chapter.)

The following example shows a typical setup for using Neo4J tests in Spring Boot:

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataNeo4jTest
public class ExampleDataNeo4jTests {

 @Autowired
 private YourRepository repository;

 //
}
```

By default, Data Neo4j tests are transactional and roll back at the end of each test. See the relevant section in the Spring Framework Reference Documentation for more details. If that is not what you want, you can disable transaction management for a test or for the whole class, as follows:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.autoconfigure.data.neo4j.DataNeo4jTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringRunner.class)
@DataNeo4jTest
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class ExampleNonTransactionalTests {

}
```

A list of the auto-configuration settings that are enabled by `@DataNeo4jTest` can be found in the appendix.

Auto-configured Data Redis Tests

You can use `@DataRedisTest` to test Redis applications. By default, it scans for `@RedisHash` classes and configures Spring Data Redis repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using Redis with Spring Boot, see "Section 30.1, “Redis”", earlier in this chapter.)

The following example shows the `@DataRedisTest` annotation in use:

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.redis.DataRedisTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataRedisTest
public class ExampleDataRedisTests {

 @Autowired
 private YourRepository repository;

 //
}
```

A list of the auto-configuration settings that are enabled by `@DataRedisTest` can be found in the appendix.

Auto-configured Data LDAP Tests

You can use `@DataLdapTest` to test LDAP applications. By default, it configures an in-memory embedded LDAP (if available), configures an `LdapTemplate`, scans for `@Entry` classes, and configures Spring Data LDAP repositories. Regular `@Component` beans are not loaded into the `ApplicationContext`. (For more about using LDAP with Spring Boot, see "Section 30.9, “LDAP”", earlier in this chapter.)

The following example shows the `@DataLdapTest` annotation in use:

```
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.ldap.core.LdapTemplate;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest
public class ExampleDataLdapTests {

 @Autowired
 private LdapTemplate ldapTemplate;

 //
}
```

In-memory embedded LDAP generally works well for tests, since it is fast and does not require any developer installation. If, however, you prefer to run tests against a real LDAP server, you should exclude the embedded LDAP auto-configuration, as shown in the following example:

```
import org.junit.runner.RunWith;
import org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration;
import org.springframework.boot.test.autoconfigure.data.ldap.DataLdapTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@DataLdapTest(excludeAutoConfiguration = EmbeddedLdapAutoConfiguration.class)
public class ExampleDataLdapNonEmbeddedTests {

}
```

A list of the auto-configuration settings that are enabled by `@DataLdapTest` can be found in the appendix.

Auto-configured REST Clients

You can use the `@RestClientTest` annotation to test REST clients. By default, it auto-configures Jackson, GSON, and Jsonb support, configures a `RestTemplateBuilder`, and adds support for `MockRestServiceServer`. The specific beans that you want to test should be specified by using the `value` or `components` attribute of `@RestClientTest`, as shown in the following example:

```
@RunWith(SpringRunner.class)
@RestClientTest(RemoteVehicleDetailsService.class)
public class ExampleRestClientTest {

 @Autowired
 private RemoteVehicleDetailsService service;

 @Autowired
 private MockRestServiceServer server;

 @Test
 public void getVehicleDetailsWhenResultIsSuccessShouldReturnDetails()
   throws Exception {
  this.server.expect(requestTo("/greet/details"))
    .andRespond(withSuccess("hello", MediaType.TEXT_PLAIN));
  String greeting = this.service.callRestService();
  assertThat(greeting).isEqualTo("hello");
 }

}
```

A list of the auto-configuration settings that are enabled by `@RestClientTest` can be found in the appendix.

Auto-configured Spring REST Docs Tests

You can use the `@AutoConfigureRestDocs` annotation to use Spring REST Docs in your tests with Mock MVC or REST Assured. It removes the need for the JUnit rule in Spring REST Docs.

`@AutoConfigureRestDocs` can be used to override the default output directory (`target/generated-snippets` if you are using Maven or `build/generated-snippets` if you are using Gradle). It can also be used to configure the host, scheme, and port that appears in any documented URIs.

Auto-configured Spring REST Docs Tests with Mock MVC

`@AutoConfigureRestDocs` customizes the `MockMvc` bean to use Spring REST Docs. You can inject it by using `@Autowired` and use it in your tests as you normally would when using Mock MVC and Spring REST Docs, as shown in the following example:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;

import static org.springframework.restdocs.mockmvc.MockMvcRestDocumentation.document;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@RunWith(SpringRunner.class)
@WebMvcTest(UserController.class)
@AutoConfigureRestDocs
public class UserDocumentationTests {

 @Autowired
 private MockMvc mvc;

 @Test
 public void listUsers() throws Exception {
  this.mvc.perform(get("/users").accept(MediaType.TEXT_PLAIN))
    .andExpect(status().isOk())
    .andDo(document("list-users"));
 }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, you can use a `RestDocsMockMvcConfigurationCustomizer` bean, as shown in the following example:

```
@TestConfiguration
static class CustomizationConfiguration
  implements RestDocsMockMvcConfigurationCustomizer {

 @Override
 public void customize(MockMvcRestDocumentationConfigurer configurer) {
  configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
 }

}
```

If you want to make use of Spring REST Docs support for a parameterized output directory, you can create a `RestDocumentationResultHandler` bean. The auto-configuration calls `alwaysDo` with this result handler, thereby causing each `MockMvc` call to automatically generate the default snippets. The following example shows a `RestDocumentationResultHandler` being defined:

```
@TestConfiguration
static class ResultHandlerConfiguration {

 @Bean
 public RestDocumentationResultHandler restDocumentation() {
  return MockMvcRestDocumentation.document("{method-name}");
 }

}
```

Auto-configured Spring REST Docs Tests with REST Assured

`@AutoConfigureRestDocs` makes a `RequestSpecification` bean, preconfigured to use Spring REST Docs, available to your tests. You can inject it by using `@Autowired` and use it in your tests as you normally would when using REST Assured and Spring REST Docs, as shown in the following example:

```
import io.restassured.specification.RequestSpecification;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.restdocs.AutoConfigureRestDocs;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.web.server.LocalServerPort;
import org.springframework.test.context.junit4.SpringRunner;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;
import static org.springframework.restdocs.restassured3.RestAssuredRestDocumentation.document;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureRestDocs
public class UserDocumentationTests {

 @LocalServerPort
 private int port;

 @Autowired
 private RequestSpecification documentationSpec;

 @Test
 public void listUsers() {
  given(this.documentationSpec).filter(document("list-users")).when()
    .port(this.port).get("/").then().assertThat().statusCode(is(200));
 }

}
```

If you require more control over Spring REST Docs configuration than offered by the attributes of `@AutoConfigureRestDocs`, a `RestDocsRestAssuredConfigurationCustomizer` bean can be used, as shown in the following example:

```
@TestConfiguration
public static class CustomizationConfiguration
  implements RestDocsRestAssuredConfigurationCustomizer {

 @Override
 public void customize(RestAssuredRestDocumentationConfigurer configurer) {
  configurer.snippets().withTemplateFormat(TemplateFormats.markdown());
 }

}
```

User Configuration and Slicing

If you structure your code in a sensible way, your `@SpringBootApplication` class is used by default as the configuration of your tests.

It then becomes important not to litter the application's main class with configuration settings that are specific to a particular area of its functionality.

Assume that you are using Spring Batch and you rely on the auto-configuration for it. You could define your `@SpringBootApplication` as follows:

```
@SpringBootApplication
@EnableBatchProcessing
public class SampleApplication { ... }
```

Because this class is the source configuration for the test, any slice test actually tries to start Spring Batch, which is definitely not what you want to do. A recommended approach is to move that area-specific configuration to a separate `@Configuration` class at the same level as your application, as shown in the following example:

```
@Configuration
@EnableBatchProcessing
public class BatchConfiguration { ... }
```

Note

Depending on the complexity of your application, you may either have a single `@Configuration` class for your customizations or one class per domain area. The latter approach lets you enable it in one of your tests, if necessary, with the `@Import` annotation.

Another source of confusion is classpath scanning. Assume that, while you structured your code in a sensible way, you need to scan an additional package. Your application may resemble the following code:

```
@SpringBootApplication
@ComponentScan({ "com.example.app", "org.acme.another" })
public class SampleApplication { ... }
```

Doing so effectively overrides the default component scan directive with the side effect of scanning those two packages regardless of the slice that you chose. For instance, a `@DataJpaTest` seems to suddenly scan components and user configurations of your application. Again, moving the custom directive to a separate class is a good way to fix this issue.

Tip

If this is not an option for you, you can create a `@SpringBootConfiguration` somewhere in the hierarchy of your test so that it is used instead. Alternatively, you can specify a source for your test, which disables the behavior of finding a default one.

Using Spock to Test Spring Boot Applications

If you wish to use Spock to test a Spring Boot application, you should add a dependency on Spock's `spock-spring` module to your application's build. `spock-spring` integrates Spring's test framework into Spock. It is recommended that you use Spock 1.1 or later to benefit from a number of improvements to Spock's Spring Framework and Spring Boot integration. See the documentation for Spock's Spring module for further details.

43.4 Test Utilities

A few test utility classes that are generally useful when testing your application are packaged as part of `spring-boot`.

ConfigFileApplicationContextInitializer

`ConfigFileApplicationContextInitializer` is an `ApplicationContextInitializer` that you can apply to your tests to load Spring Boot `application.properties` files. You can use it when you do not need the full set of features provided by `@SpringBootTest`, as shown in the following example:

```
@ContextConfiguration(classes = Config.class,
 initializers = ConfigFileApplicationContextInitializer.class)
```

Note

Using `ConfigFileApplicationContextInitializer` alone does not provide support for `@Value("${…}")` injection. Its only job is to ensure that `application.properties` files are loaded into Spring's `Environment`. For `@Value` support, you need to either additionally configure a `PropertySourcesPlaceholderConfigurer` or use `@SpringBootTest`, which auto-configures one for you.

EnvironmentTestUtils

`EnvironmentTestUtils` lets you quickly add properties to a `ConfigurableEnvironment` or `ConfigurableApplicationContext`. You can call it with `key=value` strings, as follows:

```
EnvironmentTestUtils.addEnvironment(env, "org=Spring", "name=Boot");
```

OutputCapture

`OutputCapture` is a JUnit `Rule` that you can use to capture `System.out` and `System.err` output. You can declare the capture as a `@Rule` and then use `toString()` for assertions, as follows:

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.rule.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

 @Rule
 public OutputCapture capture = new OutputCapture();

 @Test
 public void testName() throws Exception {
  System.out.println("Hello World!");
  assertThat(capture.toString(), containsString("World"));
 }

}
```

TestRestTemplate

`TestRestTemplate` is a convenience alternative to Spring's `RestTemplate` that is useful in integration tests. You can get a vanilla template or one that sends Basic HTTP authentication (with a username and password). In either case, the template behaves in a test-friendly way by not throwing exceptions on server-side errors. It is recommended, but not mandatory, to use the Apache HTTP Client (version 4.3.2 or better). If you have that on your classpath, the `TestRestTemplate` responds by configuring the client appropriately. If you do use Apache's HTTP client, some additional test-friendly features are enabled:

- Redirects are not followed (so you can assert the response location).
- Cookies are ignored (so the template is stateless).

`TestRestTemplate` can be instantiated directly in your integration tests, as shown in the following example:

```
public class MyTest {

 private TestRestTemplate template = new TestRestTemplate();

 @Test
 public void testRequest() throws Exception {
  HttpHeaders headers = template.getForEntity("http://myhost.com/example", String.class).getHeaders();
  assertThat(headers.getLocation().toString(), containsString("myotherhost"));
 }

}
```

Alternatively, if you use the `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT` or `WebEnvironment.DEFINED_PORT`, you can inject a fully configured `TestRestTemplate` and start using it. If necessary, additional customizations can be applied through the `RestTemplateBuilder` bean. Any URLs that do not specify a host and port automatically connect to the embedded server, as shown in the following example:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTest {

 @Autowired
 private TestRestTemplate template;

 @Test
 public void testRequest() throws Exception {
  HttpHeaders headers = template.getForEntity("/example", String.class).getHeaders();
  assertThat(headers.getLocation().toString(), containsString("myotherhost"));
 }
```

```
  @TestConfiguration
  static class Config {

   @Bean
   public RestTemplateBuilder restTemplateBuilder() {
    return new RestTemplateBuilder()
     .additionalMessageConverters(...)
     .customizers(...);
   }

  }

}
```

44. WebSockets

Spring Boot provides WebSockets auto-configuration for embedded Tomcat 8.5, Jetty 9, and Undertow. If you deploy a war file to a standalone container, Spring Boot assumes that the container is responsible for the configuration of its WebSocket support.

Spring Framework provides rich WebSocket support that can be easily accessed through the `spring-boot-starter-websocket` module.

45. Web Services

Spring Boot provides Web Services auto-configuration so that all you must do is define your `Endpoints`.

The Spring Web Services features can be easily accessed with the `spring-boot-starter-webservices` module.

`SimpleWsdl11Definition` and `SimpleXsdSchema` beans can be automatically created for your WSDLs and XSDs respectively. To do so, configure their location, as shown in the following example:

```
spring.webservices.wsdl-locations=classpath:/wsdl
```

46. Creating Your Own Auto-configuration

If you work in a company that develops shared libraries, or if you work on an open-source or commercial library, you might want to develop your own auto-configuration. Auto-configuration classes can be bundled in external jars and still be picked-up by Spring Boot.

Auto-configuration can be associated to a “starter” that provides the auto-configuration code as well as the typical libraries that you would use with it. We first cover what you need to know to build your own auto-configuration and then we move on to the typical steps required to create a custom starter.

Tip

A demo project is available to showcase how you can create a starter step-by-step.

46.1 Understanding Auto-configured Beans

Under the hood, auto-configuration is implemented with standard `@Configuration` classes. Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply. Usually, auto-configuration classes use `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations. This ensures that auto-configuration applies only when relevant classes are found and when you have not declared your own `@Configuration`.

You can browse the source code of `spring-boot-autoconfigure` to see the `@Configuration` classes that Spring provides (see the `META-INF/spring.factories` file).

46.2 Locating Auto-configuration Candidates

Spring Boot checks for the presence of a `META-INF/spring.factories` file within your published jar. The file should list your configuration classes under the `EnableAutoConfiguration` key, as shown in the following example:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

You can use the `@AutoConfigureAfter` or `@AutoConfigureBefore` annotations if your configuration needs to be applied in a specific order. For example, if you provide web-specific configuration, your class may need to be applied after `WebMvcAutoConfiguration`.

If you want to order certain auto-configurations that should not have any direct knowledge of each other, you can also use `@AutoConfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

Note

Auto-configurations must be loaded that way *only*. Make sure that they are defined in a specific package space and that, in particular, they are never the target of component scanning.

46.3 Condition Annotations

You almost always want to include one or more `@Conditional` annotations on your auto-configuration class. The `@ConditionalOnMissingBean` annotation is one common example that is used to allow developers to override auto-configuration if they are not happy with your defaults.

Spring Boot includes a number of `@Conditional` annotations that you can reuse in your own code by annotating `@Configuration` classes or individual `@Bean` methods. These annotations include:

- the section called “Class Conditions”
- the section called “Bean Conditions”
- the section called “Property Conditions”
- the section called “Resource Conditions”
- the section called “Web Application Conditions”
- the section called “SpEL Expression Conditions”

Class Conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations let configuration be included based on the presence or absence of specific classes. Due to the fact that annotation metadata is parsed by using ASM, you can use the `value` attribute to refer to the real class, even though that class might not actually appear on the running application classpath. You can also use the `name` attribute if you prefer to specify the class name by using a `String` value.

> **Tip**
>
> If you use `@ConditionalOnClass` or `@ConditionalOnMissingClass` as a part of a meta-annotation to compose your own composed annotations, you must use `name` as referring to the class in such a case is not handled.

Bean Conditions

The `@ConditionalOnBean` and `@ConditionalOnMissingBean` annotations let a bean be included based on the presence or absence of specific beans. You can use the `value` attribute to specify beans by type or `name` to specify beans by name. The `search` attribute lets you limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

When placed on a `@Bean` method, the target type defaults to the return type of the method, as shown in the following example:

```
@Configuration
public class MyAutoConfiguration {

 @Bean
 @ConditionalOnMissingBean
 public MyService myService() { ... }

}
```

In the preceding example, the `myService` bean is going to be created if no bean of type `MyService` is already contained in the `ApplicationContext`.

> **Tip**
>
> You need to be very careful about the order in which bean definitions are added, as these conditions are evaluated based on what has been processed so far. For this reason, we recommend using only `@ConditionalOnBean` and `@ConditionalOnMissingBean`

annotations on auto-configuration classes (since these are guaranteed to load after any user-defined bean definitions have been added).

Note

@ConditionalOnBean and @ConditionalOnMissingBean do not prevent @Configuration classes from being created. Using these conditions at the class level is equivalent to marking each contained @Bean method with the annotation.

Property Conditions

The @ConditionalOnProperty annotation lets configuration be included based on a Spring Environment property. Use the prefix and name attributes to specify the property that should be checked. By default, any property that exists and is not equal to false is matched. You can also create more advanced checks by using the havingValue and matchIfMissing attributes.

Resource Conditions

The @ConditionalOnResource annotation lets configuration be included only when a specific resource is present. Resources can be specified by using the usual Spring conventions, as shown in the following example: file:/home/user/test.dat.

Web Application Conditions

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations let configuration be included depending on whether the application is a “web application”. A web application is any application that uses a Spring WebApplicationContext, defines a session scope, or has a StandardServletEnvironment.

SpEL Expression Conditions

The @ConditionalOnExpression annotation lets configuration be included based on the result of a SpEL expression.

46.4 Testing your Auto-configuration

An auto-configuration can be affected by many factors: user configuration (@Bean definition and Environment customization), condition evaluation (presence of a particular library), and others. Concretely, each test should create a well defined ApplicationContext that represents a combination of those customizations. ApplicationContextRunner provides a great way to achieve that.

ApplicationContextRunner is usually defined as a field of the test class to gather the base, common configuration. The following example makes sure that UserServiceAutoConfiguration is always invoked:

```
private final ApplicationContextRunner contextRunner = new ApplicationContextRunner()
  .withConfiguration(AutoConfigurations.of(UserServiceAutoConfiguration.class));
```

Tip

If multiple auto-configurations have to be defined, there is no need to order their declarations as they are invoked in the exact same order as when running the application.

Each test can use the runner to represent a particular use case. For instance, the sample below invokes a user configuration (`UserConfiguration`) and checks that the auto-configuration backs off properly. Invoking `run` provides a callback context that can be used with `Assert4J`.

```
@Test
public void defaultServiceBacksOff() {
 this.contextRunner.withUserConfiguration(UserConfiguration.class)
   .run((context) -> {
    assertThat(context).hasSingleBean(UserService.class);
    assertThat(context.getBean(UserService.class)).isSameAs(
      context.getBean(UserConfiguration.class).myUserService());
   });
}

@Configuration
static class UserConfiguration {

 @Bean
 public UserService myUserService() {
  return new UserService("mine");
 }

}
```

It is also possible to easily customize the `Environment`, as shown in the following example:

```
@Test
public void serviceNameCanBeConfigured() {
 this.contextRunner.withPropertyValues("user.name=test123").run((context) -> {
  assertThat(context).hasSingleBean(UserService.class);
  assertThat(context.getBean(UserService.class).getName()).isEqualTo("test123");
 });
}
```

Simulating a Web Context

If you need to test an auto-configuration that only operates in a Servlet or Reactive web application context, use the `WebApplicationContextRunner` or `ReactiveWebApplicationContextRunner` respectively.

Overriding the Classpath

It is also possible to test what happens when a particular class and/or package is not present at runtime. Spring Boot ships with a `FilteredClassLoader` that can easily be used by the runner. In the following example, we assert that if `UserService` is not present, the auto-configuration is properly disabled:

```
@Test
public void serviceIsIgnoredIfLibraryIsNotPresent() {
 this.contextRunner.withClassLoader(new FilteredClassLoader(UserService.class))
   .run((context) -> assertThat(context).doesNotHaveBean("userService"));
}
```

46.5 Creating Your Own Starter

A full Spring Boot starter for a library may contain the following components:

- The `autoconfigure` module that contains the auto-configuration code.
- The `starter` module that provides a dependency to the `autoconfigure` module as well as the library and any additional dependencies that are typically useful. In a nutshell, adding the starter should provide everything needed to start using that library.

> **Tip**
>
> You may combine the auto-configuration code and the dependency management in a single module if you do not need to separate those two concerns.

Naming

You should make sure to provide a proper namespace for your starter. Do not start your module names with `spring-boot`, even if you use a different Maven `groupId`. We may offer official support for the thing you auto-configure in the future.

As a rule of thumb, you should name a combined module after the starter. For example, assume that you are creating a starter for "acme" and that you name the auto-configure module `acme-spring-boot-autoconfigure` and the starter `acme-spring-boot-starter`. If you only have one module that combines the two, name it `acme-spring-boot-starter`.

Also, if your starter provides configuration keys, use a unique namespace for them. In particular, do not include your keys in the namespaces that Spring Boot uses (such as `server`, `management`, `spring`, and so on). If you use the same namespace, we may modify these namespaces in the future in ways that break your modules.

Make sure to trigger meta-data generation so that IDE assistance is available for your keys as well. You may want to review the generated meta-data (`META-INF/spring-configuration-metadata.json`) to make sure your keys are properly documented.

`autoconfigure` Module

The `autoconfigure` module contains everything that is necessary to get started with the library. It may also contain configuration key definitions (such as `@ConfigurationProperties`) and any callback interface that can be used to further customize how the components are initialized.

> **Tip**
>
> You should mark the dependencies to the library as optional so that you can include the `autoconfigure` module in your projects more easily. If you do it that way, the library is not provided and, by default, Spring Boot backs off.

Starter Module

The starter is really an empty jar. Its only purpose is to provide the necessary dependencies to work with the library. You can think of it as an opinionated view of what is required to get started.

Do not make assumptions about the project in which your starter is added. If the library you are auto-configuring typically requires other starters, mention them as well. Providing a proper set of *default* dependencies may be hard if the number of optional dependencies is high, as you should avoid including dependencies that are unnecessary for a typical usage of the library. In other words, you should not include optional dependencies.

> **Note**
>
> Either way, your starter must reference the core Spring Boot starter (`spring-boot-starter`) directly or indirectly (i.e. no need to add it if your starter relies on another starter). If a project

> is created with only your custom starter, Spring Boot's core features will be honoured by the presence of the core starter.

47. What to Read Next

If you want to learn more about any of the classes discussed in this section, you can check out the Spring Boot API documentation or you can browse the source code directly. If you have specific questions, take a look at the how-to section.

If you are comfortable with Spring Boot’s core features, you can continue on and read about production-ready features.

Part V. Spring Boot Actuator: Production-ready features

Spring Boot includes a number of additional features to help you monitor and manage your application when you push it to production. You can choose to manage and monitor your application by using HTTP endpoints or with JMX. Auditing, health, and metrics gathering can also be automatically applied to your application.

48. Enabling Production-ready Features

The `spring-boot-actuator` module provides all of Spring Boot's production-ready features. The simplest way to enable the features is to add a dependency to the `spring-boot-starter-actuator` 'Starter'.

Definition of Actuator

An actuator is a manufacturing term that refers to a mechanical device for moving or controlling something. Actuators can generate a large amount of motion from a small change.

To add the actuator to a Maven based project, add the following 'Starter' dependency:

```
<dependencies>
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
 </dependency>
</dependencies>
```

For Gradle, use the following declaration:

```
dependencies {
 compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

49. Endpoints

Actuator endpoints let you monitor and interact with your application. Spring Boot includes a number of built-in endpoints and lets you add your own. For example, the `health` endpoint provides basic application health information.

Each individual endpoint can be enabled or disabled. This controls whether or not the endpoint is created and its bean exists in the application context. To be remotely accessible an endpoint also has to be exposed via JMX or HTTP. Most applications choose HTTP, where the ID of the endpoint along with a prefix of `/actuator` is mapped to a URL. For example, by default, the `health` endpoint is mapped to `/actuator/health`.

The following technology-agnostic endpoints are available:

ID	Description	Enabled by default
`auditevents`	Exposes audit events information for the current application.	Yes
`beans`	Displays a complete list of all the Spring beans in your application.	Yes
`conditions`	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.	Yes
`configprops`	Displays a collated list of all `@ConfigurationProperties`.	Yes
`env`	Exposes properties from Spring’s `ConfigurableEnvironment`.	Yes
`flyway`	Shows any Flyway database migrations that have been applied.	Yes
`health`	Shows application health information.	Yes
`httptrace`	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges).	Yes
`info`	Displays arbitrary application info.	Yes
`loggers`	Shows and modifies the configuration of loggers in the application.	Yes
`liquibase`	Shows any Liquibase database migrations that have been applied.	Yes
`metrics`	Shows ‘metrics’ information for the current application.	Yes
`mappings`	Displays a collated list of all `@RequestMapping` paths.	Yes
`scheduledtasks`	Displays the scheduled tasks in your application.	Yes
`sessions`	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Not available	Yes

ID	Description	Enabled by default
	when using Spring Session's support for reactive web applications.	
`shutdown`	Lets the application be gracefully shutdown.	No
`threaddump`	Performs a thread dump.	Yes

If your application is a web application (Spring MVC, Spring WebFlux, or Jersey), you can use the following additional endpoints:

ID	Description	Enabled by default
`heapdump`	Returns a GZip compressed `hprof` heap dump file.	Yes
`jolokia`	Exposes JMX beans over HTTP (when Jolokia is on the classpath, not available for WebFlux).	Yes
`logfile`	Returns the contents of the logfile (if `logging.file` or `logging.path` properties have been set). Supports the use of the HTTP `Range` header to retrieve part of the log file's content.	Yes
`prometheus`	Exposes metrics in a format that can be scraped by a Prometheus server.	Yes

To learn more about the Actuator's endpoints and their request and response formats, please refer to the separate API documentation (HTML or PDF).

49.1 Enabling Endpoints

By default, all endpoints except for `shutdown` are enabled. To configure the enablement of an endpoint, use its `management.endpoints.<id>.enabled` property. The following example enables the `shutdown` endpoint:

```
management.endpoint.shutdown.enabled=true
```

If you prefer endpoint enablement to be opt-in rather than opt-out, set the `management.endpoints.enabled-by-default` property to `false` and use individual endpoint `enabled` properties to opt back in. The following example enables the `info` endpoint and disables all other endpoints:

```
management.endpoints.enabled-by-default=false
management.endpoint.info.enabled=true
```

Note

Disabled endpoints are removed entirely from the application context. If you want to change only the technologies over which an endpoint is exposed, use the `expose` and `exclude` properties instead.

49.2 Exposing Endpoints

Since Endpoints may contain sensitive information, careful consideration should be given about when to expose them. The following table shows the default exposure for the built-in endpoints:

ID	JMX	Web
`auditevents`	Yes	No
`beans`	Yes	No
`conditions`	Yes	No
`configprops`	Yes	No
`env`	Yes	No
`flyway`	Yes	No
`health`	Yes	Yes
`heapdump`	N/A	No
`httptrace`	Yes	No
`info`	Yes	Yes
`jolokia`	N/A	No
`logfile`	N/A	No
`loggers`	Yes	No
`liquibase`	Yes	No
`metrics`	Yes	No
`mappings`	Yes	No
`prometheus`	N/A	No
`scheduledtasks`	Yes	No
`sessions`	Yes	No
`shutdown`	Yes	No
`threaddump`	Yes	No

To change which endpoints are exposed, use the following technology-specific `expose` and `exclude` properties:

Property	Default
`management.endpoints.jmx.exclude`	
`management.endpoints.jmx.expose`	`*`
`management.endpoints.web.exclude`	
`management.endpoints.web.expose`	`info, health`

The `expose` property lists the IDs of the endpoints that are exposed. The `exclude` property lists the IDs of the endpoints that should not be exposed. The `exclude` property takes precedence over the `expose` property.

For example, to stop exposing all endpoints over JMX and only expose the `health` endpoint, use the following property:

```
management.endpoints.jmx.expose=health
```

`*` can be used to select all endpoints. For example, to expose everything over HTTP except the `env` endpoint, use the following properties:

```
management.endpoints.web.expose=*
management.endpoints.web.exclude=env
```

Note

If your application is exposed publicly, we strongly recommend that you also secure your endpoints.

Tip

If you want to implement your own strategy for when endpoints are exposed, you can register an `EndpointFilter` bean.

49.3 Securing HTTP Endpoints

You should take care to secure HTTP endpoints in the same way that you would any other sensitive URL. If Spring Security is present, endpoints are secured by default using Spring Security's content-negotiation strategy. If you wish to configure custom security for HTTP endpoints, for example, only allow users with a certain role to access them, Spring Boot provides some convenient `RequestMatcher` objects that can be used in combination with Spring Security.

A typical Spring Security configuration might look something like the following example:

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests()
    .anyRequest().hasRole("ENDPOINT_ADMIN")
    .and()
   .httpBasic();
 }

}
```

The preceding example uses `EndpointRequest.toAnyEndpoint()` to match a request to any endpoint and then ensures that all have the `ENDPOINT_ADMIN` role. Several other matcher methods are also available on `EndpointRequest`. See the API documentation (HTML or PDF) for details.

If you deploy applications behind a firewall, you may prefer that all your actuator endpoints can be accessed without requiring authentication. You can do so by changing the `management.endpoints.web.expose` property, as follows:

application.properties.

```
management.endpoints.web.expose=*
```

Additionally, if Spring Security is present, you would need to add custom security configuration that allows unauthenticated access to the endpoints as shown in the following example:

```
@Configuration
public class ActuatorSecurity extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  http.requestMatcher(EndpointRequest.toAnyEndpoint()).authorizeRequests()
   .anyRequest().permitAll()
 }

}
```

49.4 Configuring Endpoints

Endpoints automatically cache responses to read operations that do not take any parameters. To configure the amount of time for which an endpoint will cache a response, use its `cache.time-to-live` property. The following example sets the time-to-live of the `beans` endpoint's cache to 10 seconds:

application.properties.

```
management.endpoint.beans.cache.time-to-live=10s
```

> **Note**
>
> The prefix `management.endpoint.<name>` is used to uniquely identify the endpoint that is being configured.

49.5 Hypermedia for Actuator Web Endpoints

A “discovery page” is added with links to all the endpoints. The “discovery page” is available on `/actuator` by default.

When a custom management context path is configured, the “discovery page” automatically moves from `/actuator` to the root of the management context. For example, if the management context path is `/management`, then the discovery page is available from `/management`. When the management context path is set to `/`, the discovery page is disabled to prevent the possibility of a clash with other mappings.

49.6 Actuator Web Endpoint Paths

By default, endpoints are exposed over HTTP under the `/actuator` path by using the ID of the endpoint. For example, the `beans` endpoint is exposed under `/actuator/beans`. If you want to map endpoints to a different path, you can use the `management.endpoints.web.path-mapping` property. Also, if you want change the base path, you can use `management.endpoints.web.base-path`.

The following example remaps `/actuator/health` to `/healthcheck`:

application.properties.

```
management.endpoints.web.base-path=/
management.endpoints.web.path-mapping.health=healthcheck
```

49.7 CORS Support

Cross-origin resource sharing (CORS) is a W3C specification that lets you specify in a flexible way what kind of cross-domain requests are authorized. If you use Spring MVC or Spring WebFlux, Actuator's web endpoints can be configured to support such scenarios.

CORS support is disabled by default and is only enabled once the `management.endpoints.web.cors.allowed-origins` property has been set. The following configuration permits `GET` and `POST` calls from the `example.com` domain:

```
management.endpoints.web.cors.allowed-origins=http://example.com
management.endpoints.web.cors.allowed-methods=GET,POST
```

> **Tip**
>
> See CorsEndpointProperties for a complete list of options.

49.8 Adding Custom Endpoints

If you add a `@Bean` annotated with `@Endpoint`, any methods annotated with `@ReadOperation`, `@WriteOperation`, or `@DeleteOperation` are automatically exposed over JMX and, in a web application, over HTTP as well.

You can also write technology-specific endpoints by using `@JmxEndpoint` or `@WebEndpoint`. These endpoints are filtered to their respective technologies. For example, `@WebEndpoint` is exposed only over HTTP and not over JMX.

Finally, you can write technology-specific extensions by using `@EndpointWebExtension` and `@EndpointJmxExtension`. These annotations let you provide technology-specific operations to augment an existing endpoint.

> **Tip**
>
> If you add endpoints as a library feature, consider adding a configuration class annotated with `@ManagementContextConfiguration` to `/META-INF/spring.factories` under the following key: `org.springframework.boot.actuate.autoconfigure.web.ManagementContextConfiguration`. If you do so and if your users ask for a separate management port or address, the endpoint moves to a child context with all the other web endpoints.

49.9 Health Information

You can use health information to check the status of your running application. It is often used by monitoring software to alert someone when a production system goes down. The information exposed by the `health` endpoint depends on the `management.endpoint.health.show-details` property. By default, the property's value is `false` and a simple "status" message is returned. When the property's value is set to `true`, additional details from the individual health indicators are also displayed.

Health information is collected from all `HealthIndicator` beans defined in your `ApplicationContext`. Spring Boot includes a number of auto-configured `HealthIndicators`, and you can also write your own. By default, the final system state is derived by the `HealthAggregator`, which sorts the statuses from each `HealthIndicator` based on an ordered list of statuses. The first status in the sorted list is used as the overall health status. If no `HealthIndicator` returns a status that is known to the `HealthAggregator`, an `UNKNOWN` status is used.

Auto-configured HealthIndicators

The following `HealthIndicators` are auto-configured by Spring Boot when appropriate:

Name	Description
`CassandraHealthIndicator`	Checks that a Cassandra database is up.
`DiskSpaceHealthIndicator`	Checks for low disk space.
`DataSourceHealthIndicator`	Checks that a connection to `DataSource` can be obtained.
`ElasticsearchHealthIndicator`	Checks that an Elasticsearch cluster is up.
`InfluxDbHealthIndicator`	Checks that an InfluxDB server is up.
`JmsHealthIndicator`	Checks that a JMS broker is up.
`MailHealthIndicator`	Checks that a mail server is up.
`MongoHealthIndicator`	Checks that a Mongo database is up.
`Neo4jHealthIndicator`	Checks that a Neo4j server is up.
`RabbitHealthIndicator`	Checks that a Rabbit server is up.
`RedisHealthIndicator`	Checks that a Redis server is up.
`SolrHealthIndicator`	Checks that a Solr server is up.

Tip

You can disable them all by setting the `management.health.defaults.enabled` property.

Writing Custom HealthIndicators

To provide custom health information, you can register Spring beans that implement the `HealthIndicator` interface. You need to provide an implementation of the `health()` method and return a `Health` response. The `Health` response should include a status and can optionally include additional details to be displayed. The following code shows a sample `HealthIndicator` implementation:

```
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class MyHealthIndicator implements HealthIndicator {

 @Override
 public Health health() {
  int errorCode = check(); // perform some specific health check
  if (errorCode != 0) {
   return Health.down().withDetail("Error Code", errorCode).build();
  }
  return Health.up().build();
 }

}
```

Note

The identifier for a given `HealthIndicator` is the name of the bean without the `HealthIndicator` suffix, if it exists. In the preceding example, the health information is available in an entry named `my`.

In addition to Spring Boot's predefined `Status` types, it is also possible for `Health` to return a custom `Status` that represents a new system state. In such cases, a custom implementation of the `HealthAggregator` interface also needs to be provided, or the default implementation has to be configured by using the `management.health.status.order` configuration property.

For example, assume a new `Status` with code `FATAL` is being used in one of your `HealthIndicator` implementations. To configure the severity order, add the following property to your application properties:

```
management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

The HTTP status code in the response reflects the overall health status (for example, `UP` maps to 200, while `OUT_OF_SERVICE` and `DOWN` map to 503). You might also want to register custom status mappings if you access the health endpoint over HTTP. For example, the following property maps FATAL to 503 (service unavailable):

```
management.health.status.http-mapping.FATAL=503
```

Tip

If you need more control, you can define your own `HealthStatusHttpMapper` bean.

The following table shows the default status mappings for the built-in statuses:

Status	Mapping
DOWN	SERVICE_UNAVAILABLE (503)
OUT_OF_SERVICE	SERVICE_UNAVAILABLE (503)
UP	No mapping by default, so http status is 200
UNKNOWN	No mapping by default, so http status is 200

Reactive Health Indicators

For reactive applications, such as those using Spring WebFlux, `ReactiveHealthIndicator` provides a non-blocking contract for getting application health. Similar to a traditional `HealthIndicator`, health information is collected from all `ReactiveHealthIndicator` beans defined in your `ApplicationContext`. Regular `HealthIndicator` beans that do not check against a reactive API are included and executed on the elastic scheduler.

To provide custom health information from a reactive API, you can register Spring beans that implement the `ReactiveHealthIndicator` interface. The following code shows a sample `ReactiveHealthIndicator` implementation:

```
@Component
public class MyReactiveHealthIndicator implements ReactiveHealthIndicator {

 @Override
 public Mono<Health> health() {
  return doHealthCheck() //perform some specific health check that returns a Mono<Health>
   .onErrorResume(ex -> Mono.just(new Health.Builder().down(ex).build())));
 }

}
```

Tip

To handle the error automatically, consider extending from `AbstractReactiveHealthIndicator`.

Auto-configured ReactiveHealthIndicators

The following `ReactiveHealthIndicators` are auto-configured by Spring Boot when appropriate:

Name	Description
`RedisReactiveHealthIndicator`	Checks that a Redis server is up.

Tip

If necessary, reactive indicators replace the regular ones. Also, any `HealthIndicator` that is not handled explicitly is wrapped automatically.

49.10 Application Information

Application information exposes various information collected from all `InfoContributor` beans defined in your `ApplicationContext`. Spring Boot includes a number of auto-configured `InfoContributor` beans, and you can write your own.

Auto-configured InfoContributors

The following `InfoContributor` beans are auto-configured by Spring Boot, when appropriate:

Name	Description
`EnvironmentInfoContributor`	Exposes any key from the `Environment` under the `info` key.
`GitInfoContributor`	Exposes git information if a `git.properties` file is available.
`BuildInfoContributor`	Exposes build information if a `META-INF/build-info.properties` file is available.

Tip

It is possible to disable them all by setting the `management.info.defaults.enabled` property.

Custom Application Information

You can customize the data exposed by the `info` endpoint by setting `info.*` Spring properties. All `Environment` properties under the `info` key are automatically exposed. For example, you could add the following settings to your `application.properties` file:

```
info.app.encoding=UTF-8
info.app.java.source=1.8
info.app.java.target=1.8
```

> **Tip**
>
> Rather than hardcoding those values, you could also expand info properties at build time.
>
> Assuming you use Maven, you could rewrite the preceding example as follows:
>
> ```
> info.app.encoding=@project.build.sourceEncoding@
> info.app.java.source=@java.version@
> info.app.java.target=@java.version@
> ```

Git Commit Information

Another useful feature of the `info` endpoint is its ability to publish information about the state of your `git` source code repository when the project was built. If a `GitProperties` bean is available, the `git.branch`, `git.commit.id`, and `git.commit.time` properties are exposed.

> **Tip**
>
> A `GitProperties` bean is auto-configured if a `git.properties` file is available at the root of the classpath. See "Generate git information" for more details.

If you want to display the full git information (that is, the full content of `git.properties`), use the `management.info.git.mode` property, as follows:

```
management.info.git.mode=full
```

Build Information

If a `BuildProperties` bean is available, the `info` endpoint can also publish information about your build. This happens if a `META-INF/build-info.properties` file is available in the classpath.

> **Tip**
>
> The Maven and Gradle plugins can both generate that file. See "Generate build information" for more details.

Writing Custom InfoContributors

To provide custom application information, you can register Spring beans that implement the `InfoContributor` interface.

The following example contributes an `example` entry with a single value:

```
import java.util.Collections;
```

```java
import org.springframework.boot.actuate.info.Info;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

@Component
public class ExampleInfoContributor implements InfoContributor {

 @Override
 public void contribute(Info.Builder builder) {
  builder.withDetail("example",
    Collections.singletonMap("key", "value"));
 }

}
```

If you reach the `info` endpoint, you should see a response that contains the following additional entry:

```json
{
 "example": {
  "key" : "value"
 }
}
```

50. Monitoring and Management over HTTP

If you are developing a web application, Spring Boot Actuator auto-configures all enabled endpoints to be exposed over HTTP. The default convention is to use the `id` of the endpoint with a prefix of `/actuator` as the URL path. For example, `health` is exposed as `/actuator/health`.

Tip

Actuator is supported natively with Spring MVC, Spring WebFlux, and Jersey.

50.1 Customizing the Management Endpoint Paths

Sometimes, it is useful to customize the prefix for the management endpoints. For example, your application might already use `/actuator` for another purpose. You can use the `management.endpoints.web.base-path` property to change the prefix for your management endpoint, as shown in the following example:

```
management.endpoints.web.base-path=/manage
```

The preceding `application.properties` example changes the endpoint from `/actuator/{id}` to `/manage/{id}` (for example, `/manage/info`).

Note

Unless the management port has been configured to expose endpoints by using a different HTTP port, `management.endpoints.web.base-path` is relative to `server.servlet.context-path`. If `management.server.port` is configured, `management.endpoints.web.base-path` is relative to `management.server.servlet.context-path`.

50.2 Customizing the Management Server Port

Exposing management endpoints by using the default HTTP port is a sensible choice for cloud-based deployments. If, however, your application runs inside your own data center, you may prefer to expose endpoints by using a different HTTP port.

You can set the `management.server.port` property to change the HTTP port, as shown in the following example:

```
management.server.port=8081
```

50.3 Configuring Management-specific SSL

When configured to use a custom port, the management server can also be configured with its own SSL by using the various `management.server.ssl.*` properties. For example, doing so lets a management server be available over HTTP while the main application uses HTTPS, as shown in the following property settings:

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:store.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=false
```

Alternatively, both the main server and the management server can use SSL but with different key stores, as follows:

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:main.jks
server.ssl.key-password=secret
management.server.port=8080
management.server.ssl.enabled=true
management.server.ssl.key-store=classpath:management.jks
management.server.ssl.key-password=secret
```

50.4 Customizing the Management Server Address

You can customize the address that the management endpoints are available on by setting the `management.server.address` property. Doing so can be useful if you want to listen only on an internal or ops-facing network or to listen only for connections from `localhost`.

Note

You can listen on a different address only when the port differs from the main server port.

The following example `application.properties` does not allow remote management connections:

```
management.server.port=8081
management.server.address=127.0.0.1
```

50.5 Disabling HTTP Endpoints

If you do not want to expose endpoints over HTTP, you can set the management port to `-1`, as shown in the following example:

```
management.server.port=-1
```

51. Monitoring and Management over JMX

Java Management Extensions (JMX) provide a standard mechanism to monitor and manage applications. By default, Spring Boot exposes management endpoints as JMX MBeans under the `org.springframework.boot` domain.

51.1 Customizing MBean Names

The name of the MBean is usually generated from the `id` of the endpoint. For example, the `health` endpoint is exposed as `org.springframework.boot:type=Endpoint,name=Health`.

If your application contains more than one Spring `ApplicationContext`, you may find that names clash. To solve this problem, you can set the `management.endpoints.jmx.unique-names` property to `true` so that MBean names are always unique.

You can also customize the JMX domain under which endpoints are exposed. The following settings show an example of doing so in `application.properties`:

```
management.endpoints.jmx.domain=com.example.myapp
management.endpoints.jmx.unique-names=true
```

51.2 Disabling JMX Endpoints

If you do not want to expose endpoints over JMX, you can set the `management.endpoints.jmx.exclude` property to `*`, as shown in the following example:

```
management.endpoints.jmx.exclude=*
```

51.3 Using Jolokia for JMX over HTTP

Jolokia is a JMX-HTTP bridge that provides an alternative method of accessing JMX beans. To use Jolokia, include a dependency to `org.jolokia:jolokia-core`. For example, with Maven, you would add the following dependency:

```
<dependency>
 <groupId>org.jolokia</groupId>
 <artifactId>jolokia-core</artifactId>
 </dependency>
```

The Jolokia endpoint can then be exposed by adding `jolokia` or `*` to the `management.endpoints.web.expose` property. You can then access it by using `/actuator/jolokia` on your management HTTP server.

Customizing Jolokia

Jolokia has a number of settings that you would traditionally configure by setting servlet parameters. With Spring Boot, you can use your `application.properties` file. To do so, prefix the parameter with `management.endpoint.jolokia.config.`, as shown in the following example:

```
management.endpoint.jolokia.config.debug=true
```

Disabling Jolokia

If you use Jolokia but do not want Spring Boot to configure it, set the `management.endpoint.jolokia.enabled` property to `false`, as follows:

```
management.jolokia.enabled=false
```

52. Loggers

Spring Boot Actuator includes the ability to view and configure the log levels of your application at runtime. You can view either the entire list or an individual logger’s configuration, which is made up of both the explicitly configured logging level as well as the effective logging level given to it by the logging framework. These levels can be one of:

- `TRACE`
- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`
- `OFF`
- `null`

`null` indicates that there is no explicit configuration.

52.1 Configure a Logger

To configure a given logger, `POST` a partial entity to the resource’s URI, as shown in the following example:

```
{
 "configuredLevel": "DEBUG"
}
```

Tip

To “reset” the specific level of the logger (and use the default configuration instead), you can pass a value of `null` as the `configuredLevel`.

53. Metrics

Spring Boot Actuator provides dependency management and auto-configuration for Micrometer, an application metrics facade that supports numerous monitoring systems, including:

- Atlas
- Datadog
- Ganglia
- Graphite
- Influx
- Prometheus

> **Note**
>
> At the time of this writing, the number of monitoring systems supported by Micrometer is growing rapidly. See the Micrometer project for more information.

Micrometer provides a separate module for each supported monitoring system. Depending on one (or more) of these modules is sufficient to get started with Micrometer in your Spring Boot application. To learn more about Micrometer's capabilities, please refer to its reference documentation.

53.1 Spring MVC Metrics

Auto-configuration enables the instrumentation of requests handled by Spring MVC. When `management.metrics.web.server.auto-time-requests` is `true`, this instrumentation occurs for all requests. Alternatively, when set to `false`, you can enable instrumentation by adding `@Timed` to a request-handling method.

By default, metrics are generated with the name, `http.server.requests`. The name can be customized by setting the `management.metrics.web.server.requests-metric-name` property.

Spring MVC Metric Tags

By default, Spring MVC-related metrics are tagged with the following information:

- The request's method.
- The request's URI (templated if possible).
- The simple class name of any exception that was thrown while handling the request.
- The response's status.

To customize the tags, provide a `@Bean` that implements `WebMvcTagsProvider`.

53.2 WebFlux Metrics

Auto-configuration enables the instrumentation of all requests handled by WebFlux controllers. You can also use a helper class, `RouterFunctionMetrics`, to instrument applications that use WebFlux's functional programming model.

By default, metrics are generated with the name `http.server.requests`. You can customize the name by setting the `management.metrics.web.server.requests-metric-name` property.

WebFlux Metrics Tags

By default, WebFlux-related metrics for the annotation-based programming model are tagged with the following information:

- The request's method.
- The request's URI (templated if possible).
- The simple class name of any exception that was thrown while handling the request.
- The response's status.

To customize the tags, provide a `@Bean` that implements `WebFluxTagsProvider`.

By default, metrics for the functional programming model are tagged with the following information:

- The request's method
- The request's URI (templated if possible).
- The response's status.

To customize the tags, use the `defaultTags` method on your `RouterFunctionMetrics` instance.

53.3 RestTemplate Metrics

The instrumentation of any `RestTemplate` created using the auto-configured `RestTemplateBuilder` is enabled. It is also possible to apply `MetricsRestTemplateCustomizer` manually.

By default, metrics are generated with the name, `http.client.requests`. The name can be customized by setting the `management.metrics.web.client.requests-metric-name` property.

RestTemplate Metric Tags

By default, metrics generated by an instrumented `RestTemplate` are tagged with the following information:

- The request's method.
- The request's URI (templated if possible).
- The response's status.
- The request URI's host.

53.4 Cache metrics

Auto-configuration will enable the instrumentation of all available `Caches` on startup with a metric named `cache`. The prefix can be customized by using the `management.metrics.cache.metric-name` property. Cache instrumentation is specific to each cache library, refer to the micrometer documentation for more details.

The following cache libraries are supported:

- Caffeine
- EhCache 2
- Hazelcast
- Any compliant JCache (JSR-107) implementation

Metrics will also be tagged by the name of the `CacheManager` computed based on the bean name.

> **Note**
>
> Only caches that are available on startup are bound to the registry. For caches created on-the-fly or programmatically after the startup phase, an explicit registration is required. A `CacheMetricsRegistrar` bean is made available to make that process easier.

53.5 DataSource Metrics

Auto-configuration enables the instrumentation of all available `DataSource` objects with a metric named `data.source`. Data source instrumentation results in gauges representing the currently active, maximum allowed, and minimum allowed connections in the pool. Each of these gauges has a name that is prefixed by `data.source` by default. The prefix can be customized by setting the `management.metrics.jdbc.metric-name` property.

Metrics are also tagged by the name of the `DataSource` computed based on the bean name.

53.6 RabbitMQ metrics

Auto-configuration will enable the instrumentation of all available RabbitMQ connection factories with a metric named `rabbitmq`. The prefix can be customized by using the `management.metrics.rabbitmq.metric-name` property.

53.7 Spring Integration Metrics

Auto-configuration enables binding of a number of Spring Integration-related metrics:

Table 53.1. General metrics

Metric	**Description**
`spring.integration.channelNames`	Number of Spring Integration channels
`spring.integration.handlerNames`	Number of Spring Integration handlers
`spring.integration.sourceNames`	Number of Spring Integration sources

Table 53.2. Channel metrics

Metric	**Description**
`spring.integration.channel.receives`	Number of receives
`spring.integration.channel.sendErrors`	Number of failed sends

Metric	Description
`spring.integration.channel.sends`	Number of successful sends

Table 53.3. Handler metrics

Metric	Description
`spring.integration.handler.duration.max`	Maximum handler duration in milliseconds
`spring.integration.handler.duration.min`	Minimum handler duration in milliseconds
`spring.integration.handler.duration.mean`	Mean handler duration in milliseconds
`spring.integration.handler.activeCount`	Number of active handlers

Table 53.4. Source metrics

Metric	Description
`spring.integration.source.messages`	Number of successful source calls

54. Auditing

Once Spring Security is in play, Spring Boot Actuator has a flexible audit framework that publishes events (by default, “authentication success”, “failure” and “access denied” exceptions). This feature can be very useful for reporting and for implementing a lock-out policy based on authentication failures. To customize published security events, you can provide your own implementations of `AbstractAuthenticationAuditListener` and `AbstractAuthorizationAuditListener`.

You can also use the audit services for your own business events. To do so, either inject the existing `AuditEventRepository` into your own components and use that directly or publish an `AuditApplicationEvent` with the Spring `ApplicationEventPublisher` (by implementing `ApplicationEventPublisherAware`).

55. HTTP Tracing

Tracing is automatically enabled for all HTTP requests. You can view the `httptrace` endpoint and obtain basic information about the last 100 request-response exchanges.

55.1 Custom HTTP tracing

To customize the items that are included in each trace, use the `management.httptrace.include` configuration property.

By default, an `InMemoryHttpTraceRepository` that stores traces for the last 100 request-response exchanges is used. If you need to expand the capacity, you can define your own instance of the `InMemoryHttpTraceRepository` bean. You can also create your own alternative `HttpTraceRepository` implementation.

56. Process Monitoring

In the `spring-boot` module, you can find two classes to create files that are often useful for process monitoring:

- `ApplicationPidFileWriter` creates a file containing the application PID (by default, in the application directory with a file name of `application.pid`).
- `EmbeddedServerPortFileWriter` creates a file (or files) containing the ports of the embedded server (by default, in the application directory with a file name of `application.port`).

By default, these writers are not activated, but you can enable:

- By Extending Configuration
- Section 56.2, “Programmatically”

56.1 Extending Configuration

In the `META-INF/spring.factories` file, you can activate the listener(s) that writes a PID file, as shown in the following example:

```
org.springframework.context.ApplicationListener=\
org.springframework.boot.system.ApplicationPidFileWriter,\
org.springframework.boot.system.EmbeddedServerPortFileWriter
```

56.2 Programmatically

You can also activate a listener by invoking the `SpringApplication.addListeners(…)` method and passing the appropriate `Writer` object. This method also lets you customize the file name and path in the `Writer` constructor.

57. Cloud Foundry Support

Spring Boot's actuator module includes additional support that is activated when you deploy to a compatible Cloud Foundry instance. The `/cloudfoundryapplication` path provides an alternative secured route to all `@Endpoint` beans.

The extended support lets Cloud Foundry management UIs (such as the web application that you can use to view deployed applications) be augmented with Spring Boot actuator information. For example, an application status page may include full health information instead of the typical "running" or "stopped" status.

> **Note**
>
> The `/cloudfoundryapplication` path is not directly accessible to regular users. In order to use the endpoint, a valid UAA token must be passed with the request.

57.1 Disabling Extended Cloud Foundry Actuator Support

If you want to fully disable the `/cloudfoundryapplication` endpoints, you can add the following setting to your `application.properties` file:

application.properties.

```
management.cloudfoundry.enabled=false
```

57.2 Cloud Foundry Self-signed Certificates

By default, the security verification for `/cloudfoundryapplication` endpoints makes SSL calls to various Cloud Foundry services. If your Cloud Foundry UAA or Cloud Controller services use self-signed certificates, you need to set the following property:

application.properties.

```
management.cloudfoundry.skip-ssl-validation=true
```

57.3 Custom Security Configuration

If you define custom security configuration and you want extended Cloud Foundry actuator support, you should ensure that `/cloudfoundryapplication/**` paths are open. Without a direct open route, your Cloud Foundry application manager is not able to obtain endpoint data.

For Spring Security, you typically include something like `mvcMatchers("/cloudfoundryapplication/**").permitAll()` in your configuration, as shown in the following example:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
 http
  .authorizeRequests()
   .mvcMatchers("/cloudfoundryapplication/**")
    .permitAll()
   .mvcMatchers("/mypath")
    .hasAnyRole("SUPERUSER")
   .anyRequest()
```

```
    .authenticated().and()
  .httpBasic();
}
```

58. What to Read Next

If you want to explore some of the concepts discussed in this chapter, you can take a look at the actuator sample applications. You also might want to read about graphing tools such as Graphite.

Otherwise, you can continue on, to read about ‘deployment options’ or jump ahead for some in-depth information about Spring Boot’s *build tool plugins*.

Part VI. Deploying Spring Boot Applications

Spring Boot’s flexible packaging options provide a great deal of choice when it comes to deploying your application. You can deploy Spring Boot applications to a variety of cloud platforms, to container images (such as Docker), or to virtual/real machines.

This section covers some of the more common deployment scenarios.

59. Deploying to the Cloud

Spring Boot's executable jars are ready-made for most popular cloud PaaS (Platform-as-a-Service) providers. These providers tend to require that you "bring your own container". They manage application processes (not Java applications specifically), so they need an intermediary layer that adapts *your* application to the *cloud's* notion of a running process.

Two popular cloud providers, Heroku and Cloud Foundry, employ a "buildpack" approach. The buildpack wraps your deployed code in whatever is needed to *start* your application. It might be a JDK and a call to `java`, an embedded web server, or a full-fledged application server. A buildpack is pluggable, but ideally you should be able to get by with as few customizations to it as possible. This reduces the footprint of functionality that is not under your control. It minimizes divergence between development and production environments.

Ideally, your application, like a Spring Boot executable jar, has everything that it needs to run packaged within it.

In this section, we look at what it takes to get the simple application that we developed in the "Getting Started" section up and running in the Cloud.

59.1 Cloud Foundry

Cloud Foundry provides default buildpacks that come into play if no other buildpack is specified. The Cloud Foundry Java buildpack has excellent support for Spring applications, including Spring Boot. You can deploy stand-alone executable jar applications as well as traditional `.war` packaged applications.

Once you have built your application (by using, for example, `mvn clean package`) and have installed the `cf` command line tool, deploy your application by using the `cf push` command, substituting the path to your compiled `.jar`. Be sure to have logged in with your `cf` command line client before pushing an application. The following line shows using the `cf push` command to deploy an application:

```
$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

Note

In the preceding example, we substitute `acloudyspringtime` for whatever value you give `cf` as the name of your application.

See the `cf push` documentation for more options. If there is a Cloud Foundry `manifest.yml` file present in the same directory, it is considered.

At this point, `cf` starts uploading your application, producing output similar to the following example:

```
Uploading acloudyspringtime... OK
Preparing to start acloudyspringtime... OK
-----> Downloaded app package (8.9M)
-----> Java Buildpack Version: v3.12 (offline) | https://github.com/cloudfoundry/java-
buildpack.git#6f25b7e
-----> Downloading Open Jdk JRE 1.8.0_121 from https://java-buildpack.cloudfoundry.org/openjdk/trusty/
x86_64/openjdk-1.8.0_121.tar.gz (found in cache)
       Expanding Open Jdk JRE to .java-buildpack/open_jdk_jre (1.6s)
-----> Downloading Open JDK Like Memory Calculator 2.0.2_RELEASE from https://java-
buildpack.cloudfoundry.org/memory-calculator/trusty/x86_64/memory-calculator-2.0.2_RELEASE.tar.gz (found
 in cache)
       Memory Settings: -Xss349K -Xmx681574K -XX:MaxMetaspaceSize=104857K -Xms681574K -
XX:MetaspaceSize=104857K
```

```
-----> Downloading Container Certificate Trust Store 1.0.0_RELEASE from https://java-
buildpack.cloudfoundry.org/container-certificate-trust-store/container-certificate-trust-
store-1.0.0_RELEASE.jar (found in cache)
       Adding certificates to .java-buildpack/container_certificate_trust_store/truststore.jks (0.6s)
-----> Downloading Spring Auto Reconfiguration 1.10.0_RELEASE from https://java-
buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-1.10.0_RELEASE.jar (found in cache)
Checking status of app 'acloudyspringtime'...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  0 of 1 instances running (1 starting)
  ...
  1 of 1 instances running (1 running)

App started
```

Congratulations! The application is now live!

Once your application is live, you can verify the status of the deployed application by using the `cf apps` command, as shown in the following example:

```
$ cf apps
Getting applications in ...
OK

name                 requested state   instances   memory   disk   urls
...
acloudyspringtime    started           1/1         512M     1G     acloudyspringtime.cfapps.io
...
```

Once Cloud Foundry acknowledges that your application has been deployed, you should be able to find the application at the URI given. In the preceding example, you could find it at `http://acloudyspringtime.cfapps.io/`.

Binding to Services

By default, metadata about the running application as well as service connection information is exposed to the application as environment variables (for example: `$VCAP_SERVICES`). This architecture decision is due to Cloud Foundry's polyglot (any language and platform can be supported as a buildpack) nature. Process-scoped environment variables are language agnostic.

Environment variables do not always make for the easiest API, so Spring Boot automatically extracts them and flattens the data into properties that can be accessed through Spring's `Environment` abstraction, as shown in the following example:

```java
@Component
class MyBean implements EnvironmentAware {

 private String instanceId;

 @Override
 public void setEnvironment(Environment environment) {
  this.instanceId = environment.getProperty("vcap.application.instance_id");
 }

 // ...

}
```

All Cloud Foundry properties are prefixed with `vcap`. You can use `vcap` properties to access application information (such as the public URL of the application) and service information (such as database credentials). See the 'CloudFoundryVcapEnvironmentPostProcessor' Javadoc for complete details.

Tip

The Spring Cloud Connectors project is a better fit for tasks such as configuring a DataSource. Spring Boot includes auto-configuration support and a `spring-boot-starter-cloud-connectors` starter.

59.2 Heroku

Heroku is another popular PaaS platform. To customize Heroku builds, you provide a `Procfile`, which provides the incantation required to deploy an application. Heroku assigns a `port` for the Java application to use and then ensures that routing to the external URI works.

You must configure your application to listen on the correct port. The following example shows the `Procfile` for our starter REST application:

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

Spring Boot makes `-D` arguments available as properties accessible from a Spring `Environment` instance. The `server.port` configuration property is fed to the embedded Tomcat, Jetty, or Undertow instance, which then uses the port when it starts up. The `$PORT` environment variable is assigned to us by the Heroku PaaS.

This should be everything you need. The most common deployment workflow for Heroku deployments is to `git push` the code to production, as shown in the following example:

```
$ git push heroku master

Initializing repository, done.
Counting objects: 95, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (78/78), done.
Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, done.
Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected
-----> Installing OpenJDK 1.8... done
-----> Installing Maven 3.3.1... done
-----> Installing settings.xml... done
-----> Executing: mvn -B -DskipTests=true clean install

       [INFO] Scanning for projects...
       Downloading: http://repo.spring.io/...
       Downloaded: http://repo.spring.io/... (818 B at 1.8 KB/sec)
  ....
       Downloaded: http://s3pository.heroku.com/jvm/... (152 KB at 595.3 KB/sec)
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/...
       [INFO] Installing /tmp/build_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...
       [INFO] ------------------------------------------------------------------------
       [INFO] BUILD SUCCESS
       [INFO] ------------------------------------------------------------------------
       [INFO] Total time: 59.358s
       [INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014
       [INFO] Final Memory: 20M/493M
       [INFO] ------------------------------------------------------------------------

-----> Discovering process types
       Procfile declares types -> web

-----> Compressing... done, 70.4MB
-----> Launching... done, v6
       http://agile-sierra-1405.herokuapp.com/ deployed to Heroku

To git@heroku.com:agile-sierra-1405.git
```

```
 * [new branch]      master -> master
```

Your application should now be up and running on Heroku.

59.3 OpenShift

OpenShift is the Red Hat public (and enterprise) extension of the Kubernetes container orchestration platform. Similarly to Kubernetes, OpenShift has many options for installing Spring Boot based applications.

OpenShift has many resources describing how to deploy Spring Boot applications, including:

- Using the S2I builder
- Architecture guide
- Running as a traditional web application on Wildfly
- OpenShift Commons Briefing

59.4 Amazon Web Services (AWS)

Amazon Web Services offers multiple ways to install Spring Boot-based applications, either as traditional web applications (war) or as executable jar files with an embedded web server. The options include:

- AWS Elastic Beanstalk
- AWS Code Deploy
- AWS OPS Works
- AWS Cloud Formation
- AWS Container Registry

Each has different features and pricing models. In this document, we describe only the simplest option: AWS Elastic Beanstalk.

AWS Elastic Beanstalk

As described in the official Elastic Beanstalk Java guide, there are two main options to deploy a Java application. You can either use the “Tomcat Platform” or the “Java SE platform”.

Using the Tomcat Platform

This option applies to Spring Boot projects that produce a war file. No special configuration is required. You need only follow the official guide.

Using the Java SE Platform

This option applies to Spring Boot projects that produce a jar file and run an embedded web container. Elastic Beanstalk environments run an nginx instance on port 80 to proxy the actual application, running on port 5000. To configure it, add the following line to your `application.properties` file:

```
server.port=5000
```

Upload binaries instead of sources

By default, Elastic Beanstalk uploads sources and compiles them in AWS. However, it is best to upload the binaries instead. To do so, add lines similar to the following to your `.elasticbeanstalk/config.yml` file:

```
deploy:
 artifact: target/demo-0.0.1-SNAPSHOT.jar
```

Reduce costs by setting the environment type

By default an Elastic Beanstalk environment is load balanced. The load balancer has a significant cost. To avoid that cost, set the environment type to “Single instance”, as described in the Amazon documentation. You can also create single instance environments by using the CLI and the following command:

```
eb create -s
```

Summary

This is one of the easiest ways to get to AWS, but there are more things to cover, such as how to integrate Elastic Beanstalk into any CI / CD tool, use the Elastic Beanstalk Maven plugin instead of the CLI, and others. There is a exampledriven.wordpress.com/2017/01/09/spring-boot-aws-elastic-beanstalk-example/ [blog post] covering these topics more in detail.

59.5 Boxfuse and Amazon Web Services

Boxfuse works by turning your Spring Boot executable jar or war into a minimal VM image that can be deployed unchanged either on VirtualBox or on AWS. Boxfuse comes with deep integration for Spring Boot and uses the information from your Spring Boot configuration file to automatically configure ports and health check URLs. Boxfuse leverages this information both for the images it produces as well as for all the resources it provisions (instances, security groups, elastic load balancers, and so on).

Once you have created a Boxfuse account, connected it to your AWS account, installed the latest version of the Boxfuse Client, and ensured that the application has been built by Maven or Gradle (by using, for example, `mvn clean package`), you can deploy your Spring Boot application to AWS with a command similar to the following:

```
$ boxfuse run myapp-1.0.jar -env=prod
```

See the `boxfuse run` documentation for more options. If there is a boxfuse.com/docs/commandline/#configuration [`boxfuse.conf`] file present in the current directory, it is considered.

Tip

By default, Boxfuse activates a Spring profile named `boxfuse` on startup. If your executable jar or war contains an boxfuse.com/docs/payloads/springboot.html#configuration [`application-boxfuse.properties`] file, Boxfuse bases its configuration on the properties it contains.

At this point, `boxfuse` creates an image for your application, uploads it, and configures and starts the necessary resources on AWS, resulting in output similar to the following example:

```
Fusing Image for myapp-1.0.jar ...
Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0
Creating axelfontaine/myapp ...
Pushing axelfontaine/myapp:1.0 ...
Verifying axelfontaine/myapp:1.0 ...
Creating Elastic IP ...
Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...
Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1 (this may take up to 50
 seconds) ...
AMI created in 00:23.557s -> ami-d23f38cf
Creating security group boxfuse-sg_axelfontaine/myapp:1.0 ...
Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-1 ...
Instance launched in 00:30.306s -> i-92ef9f53
Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at http://52.28.235.61/ ...
Payload started in 00:29.266s -> http://52.28.235.61/
Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...
Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...
Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at http://myapp-
axelfontaine.boxfuse.io/
```

Your application should now be up and running on AWS.

See the blog post on deploying Spring Boot apps on EC2 as well as the documentation for the Boxfuse Spring Boot integration to get started with a Maven build to run the app.

59.6 Google Cloud

Google Cloud has several options that can be used to launch Spring Boot applications. The easiest to get started with is probably App Engine, but you could also find ways to run Spring Boot in a container with Container Engine or on a virtual machine with Compute Engine.

To run in App Engine, you can create a project in the UI first, which sets up a unique identifier for you and also sets up HTTP routes. Add a Java app to the project and leave it empty and then use the Google Cloud SDK to push your Spring Boot app into that slot from the command line or CI build.

App Engine needs you to create an `app.yaml` file to describe the resources your app requires. Normally, you put this file in `src/main/appengine`, and it should resemble the following file:

```yaml
service: default

runtime: java
env: flex

runtime_config:
  jdk: openjdk8

handlers:
- url: /.*
  script: this field is required, but ignored

manual_scaling:
  instances: 1

health_check:
  enable_health_check: False

env_variables:
  ENCRYPT_KEY: your_encryption_key_here
```

You can deploy the app (for example, with a Maven plugin) by adding the project ID to the build configuration, as shown in the following example:

```xml
<plugin>
```

```
 <groupId>com.google.cloud.tools</groupId>
 <artifactId>appengine-maven-plugin</artifactId>
 <version>1.3.0</version>
 <configuration>
  <project>myproject</project>
 </configuration>
</plugin>
```

Then deploy with `mvn appengine:deploy` (if you need to authenticate first, the build fails).

Note

Google App Engine Classic is tied to the Servlet 2.5 API, so you cannot deploy a Spring Application there without some modifications. See the Servlet 2.5 section of this guide.

60. Installing Spring Boot Applications

In additional to running Spring Boot applications by using `java -jar`, it is also possible to make fully executable applications for Unix systems. A fully executable jar can be executed like any other executable binary or it can be [registered with `init.d` or `systemd`](). This makes it very easy to install and manage Spring Boot applications in common production environments.

Caution

Fully executable jars work by embedding an extra script at the front of the file. Currently, some tools do not accept this format, so you may not always be able to use this technique. For example, `jar -xf` may silently fail to extract a jar or war that has been made fully executable. It is recommended that you make your jar or war fully executable only if you intend to execute it directly, rather than running it with `java -jar` or deploying it to a servlet container.

To create a ‘fully executable’ jar with Maven, use the following plugin configuration:

```
<plugin>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-maven-plugin</artifactId>
 <configuration>
  <executable>true</executable>
 </configuration>
</plugin>
```

The following example shows the equivalent Gradle configuration:

```
bootJar {
 launchScript()
}
```

You can then run your application by typing `./my-application.jar` (where `my-application` is the name of your artifact). The directory containing the jar is used as your application’s working directory.

60.1 Supported Operating Systems

The default script supports most Linux distributions and is tested on CentOS and Ubuntu. Other platforms, such as OS X and FreeBSD, require the use of a custom `embeddedLaunchScript`.

60.2 Unix/Linux Services

Spring Boot application can be easily started as Unix/Linux services by using either `init.d` or `systemd`.

Installation as an `init.d` Service (System V)

If you configured Spring Boot’s Maven or Gradle plugin to generate a [fully executable jar](), and you do not use a custom `embeddedLaunchScript`, your application can be used as an `init.d` service. To do so, symlink the jar to `init.d` to support the standard `start`, `stop`, `restart`, and `status` commands.

The script supports the following features:

- Starts the services as the user that owns the jar file
- Tracks the application’s PID by using `/var/run/<appname>/<appname>.pid`

- Writes console logs to `/var/log/<appname>.log`

Assuming that you have a Spring Boot application installed in `/var/myapp`, to install a Spring Boot application as an `init.d` service, create a symlink, as follows:

```
$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp
```

Once installed, you can start and stop the service in the usual way. For example, on a Debian-based system, you could start it with the following command:

```
$ service myapp start
```

> **Tip**
>
> If your application fails to start, check the log file written to `/var/log/<appname>.log` for errors.

You can also flag the application to start automatically by using your standard operating system tools. For example, on Debian, you could use the following command:

```
$ update-rc.d myapp defaults <priority>
```

Securing an `init.d` Service

> **Note**
>
> The following is a set of guidelines on how to secure a Spring Boot application that runs as an init.d service. It is not intended to be an exhaustive list of everything that should be done to harden an application and the environment in which it runs.

When executed as root, as is the case when root is being used to start an init.d service, the default executable script runs the application as the user who owns the jar file. You should never run a Spring Boot application as `root`, so your application's jar file should never be owned by root. Instead, create a specific user to run your application and use `chown` to make it the owner of the jar file, as shown in the following example:

```
$ chown bootapp:bootapp your-app.jar
```

In this case, the default executable script runs the application as the `bootapp` user.

> **Tip**
>
> To reduce the chances of the application's user account being compromised, you should consider preventing it from using a login shell. For example, you can set the account's shell to `/usr/sbin/nologin`.

You should also take steps to prevent the modification of your application's jar file. Firstly, configure its permissions so that it cannot be written and can only be read or executed by its owner, as shown in the following example:

```
$ chmod 500 your-app.jar
```

Second, you should also take steps to limit the damage if your application or the account that's running it is compromised. If an attacker does gain access, they could make the jar file writable and change its contents. One way to protect against this is to make it immutable by using `chattr`, as shown in the following example:

```
$ sudo chattr +i your-app.jar
```

This will prevent any user, including root, from modifying the jar.

If root is used to control the application's service and you use a `.conf` file to customize its startup, the `.conf` file is read and evaluated by the root user. It should be secured accordingly. Use `chmod` so that the file can only be read by the owner and use `chown` to make root the owner, as shown in the following example:

```
$ chmod 400 your-app.conf
$ sudo chown root:root your-app.conf
```

Installation as a `systemd` Service

`systemd` is the successor of the System V init system and is now being used by many modern Linux distributions. Although you can continue to use `init.d` scripts with `systemd`, it is also possible to launch Spring Boot applications by using `systemd` 'service' scripts.

Assuming that you have a Spring Boot application installed in `/var/myapp`, to install a Spring Boot application as a `systemd` service, create a script named `myapp.service` and place it in `/etc/systemd/system` directory. The following script offers an example:

```
[Unit]
Description=myapp
After=syslog.target

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

Important

Remember to change the `Description`, `User`, and `ExecStart` fields for your application.

Note

The `ExecStart` field does not declare the script action command, which means that the `run` command is used by default.

Note that, unlike when running as an `init.d` service, the user that runs the application, the PID file, and the console log file are managed by `systemd` itself and therefore must be configured by using appropriate fields in the 'service' script. Consult the service unit configuration man page for more details.

To flag the application to start automatically on system boot, use the following command:

```
$ systemctl enable myapp.service
```

Refer to `man systemctl` for more details.

Customizing the Startup Script

The default embedded startup script written by the Maven or Gradle plugin can be customized in a number of ways. For most people, using the default script along with a few customizations is usually enough. If you find you cannot customize something that you need to, use the `embeddedLaunchScript` option to write your own file entirely.

Customizing the Start Script when It Is Written

It often makes sense to customize elements of the start script as it is written into the jar file. For example, init.d scripts can provide a “description”. Since you know the description up front (and it need not change), you may as well provide it when the jar is generated.

To customize written elements, use the `embeddedLaunchScriptProperties` option of the Spring Boot Maven or Gradle plugins.

The following property substitutions are supported with the default script:

Name	Description
`mode`	The script mode. Defaults to `auto`.
`initInfoProvides`	The `Provides` section of “INIT INFO”. Defaults to `spring-boot-application` for Gradle and to `${project.artifactId}` for Maven.
`initInfoRequiredStart`	The `Required-Start` section of “INIT INFO”. Defaults to `$remote_fs $syslog $network`.
`initInfoRequiredStop`	The `Required-Stop` section of “INIT INFO”. Defaults to `$remote_fs $syslog $network`.
`initInfoDefaultStart`	The `Default-Start` section of “INIT INFO”. Defaults to `2 3 4 5`.
`initInfoDefaultStop`	The `Default-Stop` section of “INIT INFO”. Defaults to `0 1 6`.
`initInfoShortDescription`	The `Short-Description` section of “INIT INFO”. Defaults to `Spring Boot Application` for Gradle and to `${project.name}` for Maven.
`initInfoDescription`	The `Description` section of “INIT INFO”. Defaults to `Spring Boot Application` for Gradle and to `${project.description}` (falling back to `${project.name}`) for Maven.
`initInfoChkconfig`	The `chkconfig` section of “INIT INFO”. Defaults to `2345 99 01`.
`confFolder`	The default value for `CONF_FOLDER`. Defaults to the folder containing the jar.
`inlinedConfScript`	Reference to a file script that should be inlined in the default launch script. This can be used to set environmental variables such as `JAVA_OPTS` before any external config files are loaded.
`logFolder`	The default value for `LOG_FOLDER`. Only valid for an `init.d` service.
`logFilename`	The default value for `LOG_FILENAME`. Only valid for an `init.d` service.
`pidFolder`	The default value for `PID_FOLDER`. Only valid for an `init.d` service.
`pidFilename`	The default value for the name of the PID file in `PID_FOLDER`. Only valid for an `init.d` service.

Name	Description
`useStartStopDaemon`	Whether the `start-stop-daemon` command, when it's available, should be used to control the process. Defaults to `true`.
`stopWaitTime`	The default value for `STOP_WAIT_TIME`. Only valid for an `init.d` service. Defaults to 60 seconds.

Customizing a Script When It Runs

For items of the script that need to be customized *after* the jar has been written, you can use environment variables or a config file.

The following environment properties are supported with the default script:

Variable	Description
`MODE`	The "mode" of operation. The default depends on the way the jar was built but is usually `auto` (meaning it tries to guess if it is an init script by checking if it is a symlink in a directory called `init.d`). You can explicitly set it to `service` so that the `stop|start|status|restart` commands work or to `run` if you want to run the script in the foreground.
`USE_START_STOP_DAEMON`	Whether the `start-stop-daemon` command, when it's available, should be used to control the process. Defaults to `true`.
`PID_FOLDER`	The root name of the pid folder (`/var/run` by default).
`LOG_FOLDER`	The name of the folder in which to put log files (`/var/log` by default).
`CONF_FOLDER`	The name of the folder from which to read .conf files (same folder as jar-file by default).
`LOG_FILENAME`	The name of the log file in the `LOG_FOLDER` (`<appname>.log` by default).
`APP_NAME`	The name of the app. If the jar is run from a symlink, the script guesses the app name. If it is not a symlink or you want to explicitly set the app name, this can be useful.
`RUN_ARGS`	The arguments to pass to the program (the Spring Boot app).
`JAVA_HOME`	The location of the `java` executable is discovered by using the `PATH` by default, but you can set it explicitly if there is an executable file at `$JAVA_HOME/bin/java`.
`JAVA_OPTS`	Options that are passed to the JVM when it is launched.
`JARFILE`	The explicit location of the jar file, in case the script is being used to launch a jar that it is not actually embedded.
`DEBUG`	If not empty, sets the `-x` flag on the shell process, making it easy to see the logic in the script.
`STOP_WAIT_TIME`	The time in seconds to wait when stopping the application before forcing a shutdown (`60` by default).

Note

The `PID_FOLDER`, `LOG_FOLDER`, and `LOG_FILENAME` variables are only valid for an `init.d` service. For `systemd`, the equivalent customizations are made by using the ‘service’ script. See the service unit configuration man page for more details.

With the exception of `JARFILE` and `APP_NAME`, the settings listed in the preceding section can be configured by using a `.conf` file. The file is expected to be next to the jar file and have the same name but suffixed with `.conf` rather than `.jar`. For example, a jar named `/var/myapp/myapp.jar` uses the configuration file named `/var/myapp/myapp.conf`, as shown in the following example:

myapp.conf.

```
JAVA_OPTS=-Xmx1024M
LOG_FOLDER=/custom/log/folder
```

Tip

If you do not like having the config file next to the jar file, you can set a `CONF_FOLDER` environment variable to customize the location of the config file.

To learn about securing this file appropriately, see the guidelines for securing an init.d service.

60.3 Microsoft Windows Services

A Spring Boot application can be started as a Windows service by using `winsw`.

A (separately maintained sample) describes step-by-step how you can create a Windows service for your Spring Boot application.

61. What to Read Next

Check out the Cloud Foundry, Heroku, OpenShift, and Boxfuse web sites for more information about the kinds of features that a PaaS can offer. These are just four of the most popular Java PaaS providers. Since Spring Boot is so amenable to cloud-based deployment, you can freely consider other providers as well.

The next section goes on to cover the *Spring Boot CLI*, or you can jump ahead to read about *build tool plugins*.

Part VII. Spring Boot CLI

The Spring Boot CLI is a command line tool that you can use if you want to quickly develop a Spring application. It lets you run Groovy scripts, which means that you have a familiar Java-like syntax without so much boilerplate code. You can also bootstrap a new project or write your own command for it.

62. Installing the CLI

The Spring Boot CLI (Command-Line Interface) can be installed manually by using SDKMAN! (the SDK Manager) or by using Homebrew or MacPorts if you are an OSX user. See *Section 10.2, “Installing the Spring Boot CLI”* in the “Getting started” section for comprehensive installation instructions.

63. Using the CLI

Once you have installed the CLI, you can run it by typing `spring` and pressing Enter at the command line. If you run `spring` without any arguments, a simple help screen is displayed, as follows:

```
$ spring
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  run [options] <files> [--] [args]
    Run a spring groovy script

  ... more command help is shown here
```

You can type `spring help` to get more details about any of the supported commands, as shown in the following example:

```
$ spring help run
spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args]

Option                     Description
------                     -----------
--autoconfigure [Boolean]  Add autoconfigure compiler
                             transformations (default: true)
--classpath, -cp           Additional classpath entries
-e, --edit                 Open the file with the default system
                             editor
--no-guess-dependencies    Do not attempt to guess dependencies
--no-guess-imports         Do not attempt to guess imports
-q, --quiet                Quiet logging
-v, --verbose              Verbose logging of dependency
                             resolution
--watch                    Watch the specified file for changes
```

The `version` command provides a quick way to check which version of Spring Boot you are using, as follows:

```
$ spring version
Spring CLI v2.0.0.RC1
```

63.1 Running Applications with the CLI

You can compile and run Groovy source code by using the `run` command. The Spring Boot CLI is completely self-contained, so you do not need any external Groovy installation.

The following example shows a “hello world” web application written in Groovy:

hello.groovy.

```
@RestController
class WebApplication {

 @RequestMapping("/")
 String home() {
  "Hello World!"
 }

}
```

To compile and run the application, type the following command:

```
$ spring run hello.groovy
```

To pass command-line arguments to the application, use `--` to separate the commands from the “spring” command arguments, as shown in the following example:

```
$ spring run hello.groovy -- --server.port=9000
```

To set JVM command line arguments, you can use the `JAVA_OPTS` environment variable, as shown in the following example:

```
$ JAVA_OPTS=-Xmx1024m spring run hello.groovy
```

Note

When setting `JAVA_OPTS` on Microsoft Windows, make sure to quote the entire instruction, such as `set "JAVA_OPTS=-Xms256m -Xmx2048m"`. Doing so ensures the values are properly passed to the process.

Deduced “grab” Dependencies

Standard Groovy includes a `@Grab` annotation, which lets you declare dependencies on third-party libraries. This useful technique lets Groovy download jars in the same way as Maven or Gradle would but without requiring you to use a build tool.

Spring Boot extends this technique further and tries to deduce which libraries to “grab” based on your code. For example, since the `WebApplication` code shown previously uses `@RestController` annotations, Spring Boot grabs "Tomcat" and "Spring MVC".

The following items are used as “grab hints”:

Items	Grabs
`JdbcTemplate`, `NamedParameterJdbcTemplate`, `DataSource`	JDBC Application.
`@EnableJms`	JMS Application.
`@EnableCaching`	Caching abstraction.
`@Test`	JUnit.
`@EnableRabbit`	RabbitMQ.
`@EnableReactor`	Project Reactor.
extends `Specification`	Spock test.
`@EnableBatchProcessing`	Spring Batch.
`@MessageEndpoint` `@EnableIntegration`	Spring Integration.
`@Controller` `@RestController` `@EnableWebMvc`	Spring MVC + Embedded Tomcat.

Items	Grabs
`@EnableWebSecurity`	Spring Security.
`@EnableTransactionManagement`	Spring Transaction Management.

> **Tip**
>
> See subclasses of `CompilerAutoConfiguration` in the Spring Boot CLI source code to understand exactly how customizations are applied.

Deduced “grab” Coordinates

Spring Boot extends Groovy’s standard `@Grab` support by letting you specify a dependency without a group or version (for example, `@Grab('freemarker')`). Doing so consults Spring Boot’s default dependency metadata to deduce the artifact’s group and version.

> **Note**
>
> The default metadata is tied to the version of the CLI that you use. it changes only when you move to a new version of the CLI, putting you in control of when the versions of your dependencies may change. A table showing the dependencies and their versions that are included in the default metadata can be found in the appendix.

Default Import Statements

To help reduce the size of your Groovy code, several `import` statements are automatically included. Notice how the preceding example refers to `@Component`, `@RestController`, and `@RequestMapping` without needing to use fully-qualified names or `import` statements.

> **Tip**
>
> Many Spring annotations work without using `import` statements. Try running your application to see what fails before adding imports.

Automatic Main Method

Unlike the equivalent Java application, you do not need to include a `public static void main(String[] args)` method with your `Groovy` scripts. A `SpringApplication` is automatically created, with your compiled code acting as the `source`.

Custom Dependency Management

By default, the CLI uses the dependency management declared in `spring-boot-dependencies` when resolving `@Grab` dependencies. Additional dependency management, which overrides the default dependency management, can be configured by using the `@DependencyManagementBom` annotation. The annotation’s value should specify the coordinates (`groupId:artifactId:version`) of one or more Maven BOMs.

For example, consider the following declaration:

```
@DependencyManagementBom("com.example.custom-bom:1.0.0")
```

The preceding declaration picks up `custom-bom-1.0.0.pom` in a Maven repository under `com/example/custom-versions/1.0.0/`.

When you specify multiple BOMs, they are applied in the order in which you declare them, as shown in the following example:

```
@DependencyManagementBom(["com.example.custom-bom:1.0.0",
  "com.example.another-bom:1.0.0"])
```

The preceding example indicates that the dependency management in `another-bom` overrides the dependency management in `custom-bom`.

You can use `@DependencyManagementBom` anywhere that you can use `@Grab`. However, to ensure consistent ordering of the dependency management, you can use `@DependencyManagementBom` at most once in your application. A useful source of dependency management (which is a superset of Spring Boot's dependency management) is the Spring IO Platform, which you might include with the following line:

```
@DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE')
```

63.2 Applications with Multiple Source Files

You can use "shell globbing" with all commands that accept file input. Doing so lets you use multiple files from a single directory, as shown in the following example:

```
$ spring run *.groovy
```

63.3 Packaging Your Application

You can use the `jar` command to package your application into a self-contained executable jar file, as shown in the following example:

```
$ spring jar my-app.jar *.groovy
```

The resulting jar contains the classes produced by compiling the application and all of the application's dependencies so that it can then be run by using `java -jar`. The jar file also contains entries from the application's classpath. You can add and remove explicit paths to the jar by using `--include` and `--exclude`. Both are comma-separated, and both accept prefixes, in the form of "+" and "-", to signify that they should be removed from the defaults. The default includes are as follows:

```
public/**, resources/**, static/**, templates/**, META-INF/**, *
```

The default excludes are as follows:

```
.*, repository/**, build/**, target/**, **/*.jar, **/*.groovy
```

Type `spring help jar` on the command line for more information.

63.4 Initialize a New Project

The `init` command lets you create a new project by using start.spring.io without leaving the shell, as shown in the following example:

```
$ spring init --dependencies=web,data-jpa my-project
Using service at https://start.spring.io
Project extracted to '/Users/developer/example/my-project'
```

The preceding example creates a `my-project` directory with a Maven-based project that uses `spring-boot-starter-web` and `spring-boot-starter-data-jpa`. You can list the capabilities of the service by using the `--list` flag, as shown in the following example:

```
$ spring init --list
=======================================
Capabilities of https://start.spring.io
=======================================

Available dependencies:
-----------------------
actuator - Actuator: Production ready features to help you monitor and manage your application
...
web - Web: Support for full-stack web development, including Tomcat and spring-webmvc
websocket - Websocket: Support for WebSocket development
ws - WS: Support for Spring Web Services

Available project types:
------------------------
gradle-build -  Gradle Config [format:build, build:gradle]
gradle-project -  Gradle Project [format:project, build:gradle]
maven-build -  Maven POM [format:build, build:maven]
maven-project -  Maven Project [format:project, build:maven] (default)

...
```

The `init` command supports many options. See the `help` output for more details. For instance, the following command creates a Gradle project that uses Java 8 and `war` packaging:

```
$ spring init --build=gradle --java-version=1.8 --dependencies=websocket --packaging=war sample-app.zip
Using service at https://start.spring.io
Content saved to 'sample-app.zip'
```

63.5 Using the Embedded Shell

Spring Boot includes command-line completion scripts for the BASH and zsh shells. If you do not use either of these shells (perhaps you are a Windows user), you can use the `shell` command to launch an integrated shell, as shown in the following example:

```
$ spring shell
Spring Boot (v2.0.0.RC1)
Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.
```

From inside the embedded shell, you can run other commands directly:

```
$ version
Spring CLI v2.0.0.RC1
```

The embedded shell supports ANSI color output as well as `tab` completion. If you need to run a native command, you can use the `!` prefix. To exit the embedded shell, press `ctrl-c`.

63.6 Adding Extensions to the CLI

You can add extensions to the CLI by using the `install` command. The command takes one or more sets of artifact coordinates in the format `group:artifact:version`, as shown in the following example:

```
$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

In addition to installing the artifacts identified by the coordinates you supply, all of the artifacts' dependencies are also installed.

To uninstall a dependency, use the `uninstall` command. As with the `install` command, it takes one or more sets of artifact coordinates in the format of `group:artifact:version`, as shown in the following example:

```
$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE
```

It uninstalls the artifacts identified by the coordinates you supply and their dependencies.

To uninstall all additional dependencies, you can use the `--all` option, as shown in the following example:

```
$ spring uninstall --all
```

64. Developing Applications with the Groovy Beans DSL

Spring Framework 4.0 has native support for a `beans{}` “DSL” (borrowed from Grails), and you can embed bean definitions in your Groovy application scripts by using the same format. This is sometimes a good way to include external features like middleware declarations, as shown in the following example:

```
@Configuration
class Application implements CommandLineRunner {

 @Autowired
 SharedService service

 @Override
 void run(String... args) {
  println service.message
 }

}

import my.company.SharedService

beans {
 service(SharedService) {
  message = "Hello World"
 }
}
```

You can mix class declarations with `beans{}` in the same file as long as they stay at the top level, or, if you prefer, you can put the beans DSL in a separate file.

65. Configuring the CLI with `settings.xml`

The Spring Boot CLI uses Aether, Maven’s dependency resolution engine, to resolve dependencies. The CLI makes use of the Maven configuration found in `~/.m2/settings.xml` to configure Aether. The following configuration settings are honored by the CLI:

- Offline
- Mirrors
- Servers
- Proxies
- Profiles
  - Activation
  - Repositories
- Active profiles

See Maven’s settings documentation for further information.

66. What to Read Next

There are some sample groovy scripts available from the GitHub repository that you can use to try out the Spring Boot CLI. There is also extensive Javadoc throughout the source code.

If you find that you reach the limit of the CLI tool, you probably want to look at converting your application to a full Gradle or Maven built “Groovy project”. The next section covers Spring Boot’s "Build tool plugins", which you can use with Gradle or Maven.

Part VIII. Build tool plugins

Spring Boot provides build tool plugins for Maven and Gradle. The plugins offer a variety of features, including the packaging of executable jars. This section provides more details on both plugins as well as some help should you need to extend an unsupported build system. If you are just getting started, you might want to read “Chapter 13, *Build Systems*” from the “Part III, “Using Spring Boot”” section first.

67. Spring Boot Maven Plugin

The Spring Boot Maven Plugin provides Spring Boot support in Maven, letting you package executable jar or war archives and run an application “in-place”. To use it, you must use Maven 3.2 (or later).

Note

See the Spring Boot Maven Plugin Site for complete plugin documentation.

67.1 Including the Plugin

To use the Spring Boot Maven Plugin, include the appropriate XML in the `plugins` section of your `pom.xml`, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <!-- ... -->
 <build>
  <plugins>
   <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>2.0.0.RC1</version>
    <executions>
     <execution>
      <goals>
       <goal>repackage</goal>
      </goals>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>
</project>
```

The preceding configuration repackages a jar or war that is built during the `package` phase of the Maven lifecycle. The following example shows both the repackaged jar as well as the original jar in the `target` directory:

```
$ mvn package
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

If you do not include the `<execution/>` configuration, as shown in the prior example, you can run the plugin on its own (but only if the package goal is used as well), as shown in the following example:

```
$ mvn package spring-boot:repackage
$ ls target/*.jar
target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original
```

If you use a milestone or snapshot release, you also need to add the appropriate `pluginRepository` elements, as shown in the following listing:

```
<pluginRepositories>
 <pluginRepository>
  <id>spring-snapshots</id>
  <url>http://repo.spring.io/snapshot</url>
 </pluginRepository>
 <pluginRepository>
```

```
    <id>spring-milestones</id>
    <url>http://repo.spring.io/milestone</url>
  </pluginRepository>
</pluginRepositories>
```

67.2 Packaging Executable Jar and War Files

Once `spring-boot-maven-plugin` has been included in your `pom.xml`, it automatically tries to rewrite archives to make them executable by using the `spring-boot:repackage` goal. You should configure your project to build a jar or war (as appropriate) by using the usual `packaging` element, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <!-- ... -->
 <packaging>jar</packaging>
 <!-- ... -->
</project>
```

Your existing archive is enhanced by Spring Boot during the `package` phase. The main class that you want to launch can be specified either by using a configuration option or by adding a `Main-Class` attribute to the manifest in the usual way. If you do not specify a main class, the plugin searches for a class with a `public static void main(String[] args)` method.

To build and run a project artifact, you can type the following:

```
$ mvn package
$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar
```

To build a war file that is both executable and deployable into an external container, you need to mark the embedded container dependencies as “provided”, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <!-- ... -->
 <packaging>war</packaging>
 <!-- ... -->
 <dependencies>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-tomcat</artifactId>
   <scope>provided</scope>
  </dependency>
  <!-- ... -->
 </dependencies>
</project>
```

Tip

See the “Section 86.1, “Create a Deployable War File”” section for more details on how to create a deployable war file.

Advanced configuration options and examples are available in the plugin info page.

68. Spring Boot Gradle Plugin

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, letting you package executable jar or war archives, run Spring Boot applications, and use the dependency management provided by `spring-boot-dependencies`. It requires Gradle 4.0 or later. Please refer to the plugin’s documentation to learn more:

- Reference (HTML and PDF)
- API

69. Spring Boot AntLib Module

The Spring Boot AntLib module provides basic Spring Boot support for Apache Ant. You can use the module to create executable jars. To use the module, you need to declare an additional `spring-boot` namespace in your `build.xml`, as shown in the following example:

```
<project xmlns:ivy="antlib:org.apache.ivy.ant"
 xmlns:spring-boot="antlib:org.springframework.boot.ant"
 name="myapp" default="build">
 ...
</project>
```

You need to remember to start Ant using the `-lib` option, as shown in the following example:

```
$ ant -lib <folder containing spring-boot-antlib-2.0.0.RC1.jar>
```

Tip

The “Using Spring Boot” section includes a more complete example of using Apache Ant with `spring-boot-antlib`.

69.1 Spring Boot Ant Tasks

Once the `spring-boot-antlib` namespace has been declared, the following additional tasks are available:

- the section called “`spring-boot:exejar`”
- Section 69.2, “`spring-boot:findmainclass`”

`spring-boot:exejar`

You can use the `exejar` task to create a Spring Boot executable jar. The following attributes are supported by the task:

Attribute	Description	Required
`destfile`	The destination jar file to create	Yes
`classes`	The root directory of Java class files	Yes
`start-class`	The main application class to run	No *(the default is the first class found that declares a `main` method)*

The following nested elements can be used with the task:

Element	Description
`resources`	One or more Resource Collections describing a set of Resources that should be added to the content of the created jar file.
`lib`	One or more Resource Collections that should be added to the set of jar libraries that make up the runtime dependency classpath of the application.

Examples

This section shows two examples of Ant tasks.

Specify start-class.

```
<spring-boot:exejar destfile="target/my-application.jar"
  classes="target/classes" start-class="com.example.MyApplication">
 <resources>
  <fileset dir="src/main/resources" />
 </resources>
 <lib>
  <fileset dir="lib" />
 </lib>
</spring-boot:exejar>
```

Detect start-class.

```
<exejar destfile="target/my-application.jar" classes="target/classes">
 <lib>
  <fileset dir="lib" />
 </lib>
</exejar>
```

69.2 `spring-boot:findmainclass`

The `findmainclass` task is used internally by `exejar` to locate a class declaring a `main`. If necessary, you can also use this task directly in your build. The following attributes are supported:

Attribute	Description	Required
`classesroot`	The root directory of Java class files	Yes *(unless `mainclass` is specified)*
`mainclass`	Can be used to short-circuit the `main` class search	No
`property`	The Ant property that should be set with the result	No *(result will be logged if unspecified)*

Examples

This section contains three examples of using `findmainclass`.

Find and log.

```
<findmainclass classesroot="target/classes" />
```

Find and set.

```
<findmainclass classesroot="target/classes" property="main-class" />
```

Override and set.

```
<findmainclass mainclass="com.example.MainClass" property="main-class" />
```

70. Supporting Other Build Systems

If you want to use a build tool other than Maven, Gradle, or Ant, you likely need to develop your own plugin. Executable jars need to follow a specific format and certain entries need to be written in an uncompressed form (see the “executable jar format” section in the appendix for details).

The Spring Boot Maven and Gradle plugins both make use of `spring-boot-loader-tools` to actually generate jars. If you need to, you may use this library directly.

70.1 Repackaging Archives

To repackage an existing archive so that it becomes a self-contained executable archive, use `org.springframework.boot.loader.tools.Repackager`. The `Repackager` class takes a single constructor argument that refers to an existing jar or war archive. Use one of the two available `repackage()` methods to either replace the original file or write to a new destination. Various settings can also be configured on the repackager before it is run.

70.2 Nested Libraries

When repackaging an archive, you can include references to dependency files by using the `org.springframework.boot.loader.tools.Libraries` interface. We do not provide any concrete implementations of `Libraries` here as they are usually build-system-specific.

If your archive already includes libraries, you can use `Libraries.NONE`.

70.3 Finding a Main Class

If you do not use `Repackager.setMainClass()` to specify a main class, the repackager uses ASM to read class files and tries to find a suitable class with a `public static void main(String[] args)` method. An exception is thrown if more than one candidate is found.

70.4 Example Repackage Implementation

The following example shows a typical repackage implementation:

```
Repackager repackager = new Repackager(sourceJarFile);
repackager.setBackupSource(false);
repackager.repackage(new Libraries() {
   @Override
   public void doWithLibraries(LibraryCallback callback) throws IOException {
    // Build system specific implementation, callback for each dependency
    // callback.library(new Library(nestedFile, LibraryScope.COMPILE));
   }
  });
```

71. What to Read Next

If you are interested in how the build tool plugins work, you can look at the `spring-boot-tools` module on GitHub. More technical details of the executable jar format are covered in the appendix.

If you have specific build-related questions, you can check out the “how-to” guides.

Part IX. ‘How-to’ guides

This section provides answers to some common ‘how do I do that…’ questions that often arise when using Spring Boot. Its coverage is not exhaustive, but it does cover quite a lot.

If you have a specific problem that we do not cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer. This is also a great place to ask new questions (please use the `spring-boot` tag).

We are also more than happy to extend this section. If you want to add a ‘how-to’, send us a pull request.

72. Spring Boot Application

This section includes topics relating directly to Spring Boot applications.

72.1 Create Your Own FailureAnalyzer

`FailureAnalyzer` is a great way to intercept an exception on startup and turn it into a human-readable message, wrapped in a `FailureAnalysis`. Spring Boot provides such an analyzer for application-context-related exceptions, JSR-303 validations, and more. You can also create your own.

`AbstractFailureAnalyzer` is a convenient extension of `FailureAnalyzer` that checks the presence of a specified exception type in the exception to handle. You can extend from that so that your implementation gets a chance to handle the exception only when it is actually present. If, for whatever reason, you cannot handle the exception, return `null` to give another implementation a chance to handle the exception.

`FailureAnalyzer` implementations must be registered in `META-INF/spring.factories`. The following example registers `ProjectConstraintViolationFailureAnalyzer`:

```
org.springframework.boot.diagnostics.FailureAnalyzer=\
com.example.ProjectConstraintViolationFailureAnalyzer
```

> **Note**
>
> If you need access to the `BeanFactory` or the `Environment`, your `FailureAnalyzer` can simply implement `BeanFactoryAware` or `EnvironmentAware` respectively.

72.2 Troubleshoot Auto-configuration

The Spring Boot auto-configuration tries its best to “do the right thing”, but sometimes things fail, and it can be hard to tell why.

There is a really useful `ConditionEvaluationReport` available in any Spring Boot `ApplicationContext`. You can see it if you enable `DEBUG` logging output. If you use the `spring-boot-actuator` (see the Actuator chapter), there is also a `conditions` endpoint that renders the report in JSON. Use that endpoint to debug the application and see what features have been added (and which have not been added) by Spring Boot at runtime.

Many more questions can be answered by looking at the source code and the Javadoc. When reading the code, remember the following rules of thumb:

- Look for classes called `*AutoConfiguration` and read their sources. Pay special attention to the `@Conditional*` annotations to find out what features they enable and when. Add `--debug` to the command line or a System property `-Ddebug` to get a log on the console of all the auto-configuration decisions that were made in your app. In a running Actuator app, look at the `conditions` endpoint (`/actuator/conditions` or the JMX equivalent) for the same information.
- Look for classes that are `@ConfigurationProperties` (such as `ServerProperties`) and read from there the available external configuration options. The `@ConfigurationProperties` annotation has a `name` attribute that acts as a prefix to external properties. Thus, `ServerProperties` has `prefix="server"` and its configuration properties are `server.port`, `server.address`, and others. In a running Actuator app, look at the `configprops` endpoint.

- Look for uses of the `bind` method on the `Binder` to pull configuration values explicitly out of the `Environment` in a relaxed manner. It is often used with a prefix.
- Look for `@Value` annotations that bind directly to the `Environment`.
- Look for `@ConditionalOnExpression` annotations that switch features on and off in response to SpEL expressions, normally evaluated with placeholders resolved from the `Environment`.

72.3 Customize the Environment or ApplicationContext Before It Starts

A `SpringApplication` has `ApplicationListeners` and `ApplicationContextInitializers` that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from `META-INF/spring.factories`. There is more than one way to register additional customizations:

- Programmatically, per application, by calling the `addListeners` and `addInitializers` methods on `SpringApplication` before you run it.
- Declaratively, per application, by setting the `context.initializer.classes` or `context.listener.classes` properties.
- Declaratively, for all applications, by adding a `META-INF/spring.factories` and packaging a jar file that the applications all use as a library.

The `SpringApplication` sends some special `ApplicationEvents` to the listeners (some even before the context is created) and then registers the listeners for events published by the `ApplicationContext` as well. See "Section 23.5, "Application Events and Listeners"" in the 'Spring Boot features' section for a complete list.

It is also possible to customize the `Environment` before the application context is refreshed by using `EnvironmentPostProcessor`. Each implementation should be registered in `META-INF/spring.factories`, as shown in the following example:

```
org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostProcessor
```

The implementation can load arbitrary files and add them to the `Environment`. For instance, the following example loads a YAML configuration file from the classpath:

```
public class EnvironmentPostProcessorExample implements EnvironmentPostProcessor {

 private final YamlPropertySourceLoader loader = new YamlPropertySourceLoader();

 @Override
 public void postProcessEnvironment(ConfigurableEnvironment environment,
   SpringApplication application) {
  Resource path = new ClassPathResource("com/example/myapp/config.yml");
  PropertySource<?> propertySource = loadYaml(path);
  environment.getPropertySources().addLast(propertySource);
 }

 private PropertySource<?> loadYaml(Resource path) {
  if (!path.exists()) {
   throw new IllegalArgumentException("Resource " + path + " does not exist");
  }
  try {
   return this.loader.load("custom-resource", path, null);
  }
  catch (IOException ex) {
   throw new IllegalStateException(
     "Failed to load yaml configuration from " + path, ex);
  }
 }

}
```

Tip

The `Environment` has already been prepared with all the usual property sources that Spring Boot loads by default. It is therefore possible to get the location of the file from the environment. The preceding example adds the `custom-resource` property source at the end of the list so that a key defined in any of the usual other locations takes precedence. A custom implementation may define another order.

Caution

While using `@PropertySource` on your `@SpringBootApplication` may seem to be a convenient and easy way to load a custom resource in the `Environment`, we do not recommend it, because Spring Boot prepares the `Environment` before the `ApplicationContext` is refreshed. Any key defined with `@PropertySource` is loaded too late to have any effect on auto-configuration.

72.4 Build an ApplicationContext Hierarchy (Adding a Parent or Root Context)

You can use the `ApplicationBuilder` class to create parent/child `ApplicationContext` hierarchies. See "Section 23.4, "Fluent Builder API"" in the 'Spring Boot features' section for more information.

72.5 Create a Non-web Application

Not all Spring applications have to be web applications (or web services). If you want to execute some code in a `main` method but also bootstrap a Spring application to set up the infrastructure to use, you can use the `SpringApplication` features of Spring Boot. A `SpringApplication` changes its `ApplicationContext` class, depending on whether it thinks it needs a web application or not. The first thing you can do to help it is to leave the servlet API dependencies off the classpath. If you cannot do that (for example, you run two applications from the same code base) then you can explicitly call `setWebEnvironment(false)` on your `SpringApplication` instance or set the `applicationContextClass` property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a `CommandLineRunner` and dropped into the context as a `@Bean` definition.

73. Properties and Configuration

This section includes topics about setting and reading properties and configuration settings and their interaction with Spring Boot applications.

73.1 Automatically Expand Properties at Build Time

Rather than hardcoding some properties that are also specified in your project's build configuration, you can automatically expand them by instead using the existing build configuration. This is possible in both Maven and Gradle.

Automatic Property Expansion Using Maven

You can automatically expand properties from the Maven project by using resource filtering. If you use the `spring-boot-starter-parent`, you can then refer to your Maven 'project properties' with `@..@` placeholders, as shown in the following example:

```
app.encoding=@project.build.sourceEncoding@
app.java.version=@java.version@
```

Note

Only production configuration is filtered that way (in other words, no filtering is applied on `src/test/resources`).

Tip

If you enable the `addResources` flag, the `spring-boot:run` goal can add `src/main/resources` directly to the classpath (for hot reloading purposes). Doing so circumvents the resource filtering and this feature. Instead, you can use the `exec:java` goal or customize the plugin's configuration. See the plugin usage page for more details.

If you do not use the starter parent, you need to include the following element inside the `<build/>` element of your `pom.xml`:

```
<resources>
 <resource>
  <directory>src/main/resources</directory>
  <filtering>true</filtering>
 </resource>
</resources>
```

You also need to include the following element inside `<plugins/>`:

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-resources-plugin</artifactId>
 <version>2.7</version>
 <configuration>
  <delimiters>
   <delimiter>@</delimiter>
  </delimiters>
  <useDefaultDelimiters>false</useDefaultDelimiters>
 </configuration>
</plugin>
```

> **Note**
>
> The `useDefaultDelimiters` property is important if you use standard Spring placeholders (such as `${placeholder}`) in your configuration. If that property is not set to `false`, these may be expanded by the build.

Automatic Property Expansion Using Gradle

You can automatically expand properties from the Gradle project by configuring the Java plugin's `processResources` task to do so, as shown in the following example:

```
processResources {
 expand(project.properties)
}
```

You can then refer to your Gradle project's properties by using placeholders, as shown in the following example:

```
app.name=${name}
app.description=${description}
```

> **Note**
>
> Gradle's `expand` method uses Groovy's `SimpleTemplateEngine`, which transforms `${..}` tokens. The `${..}` style conflicts with Spring's own property placeholder mechanism. To use Spring property placeholders together with automatic expansion, escape the Spring property placeholders as follows: `\${..}`.

73.2 Externalize the Configuration of `SpringApplication`

A `SpringApplication` has bean properties (mainly setters), so you can use its Java API as you create the application to modify its behavior. Alternatively, you can externalize the configuration by setting properties in `spring.main.*`. For example, in `application.properties`, you might have the following settings:

```
spring.main.web-environment=false
spring.main.banner-mode=off
```

Then the Spring Boot banner is not printed on startup, and the application is not a web application.

> **Note**
>
> The preceding example also demonstrates how flexible binding allows the use of underscores (_) as well as dashes (`-`) in property names.

Properties defined in external configuration override the values specified with the Java API, with the notable exception of the sources used to create the `ApplicationContext`. Consider the following application:

```
new SpringApplicationBuilder()
 .bannerMode(Banner.Mode.OFF)
 .sources(demo.MyApp.class)
 .run(args);
```

Now consider the following configuration:

```
spring.main.sources=com.acme.Config,com.acme.ExtraConfig
spring.main.banner-mode=console
```

The actual application *now* shows the banner (as overridden by configuration) and uses three sources for the `ApplicationContext` (in the following order): `demo.MyApp`, `com.acme.Config`, and `com.acme.ExtraConfig`.

73.3 Change the Location of External Properties of an Application

By default, properties from different sources are added to the Spring `Environment` in a defined order (see “Chapter 24, *Externalized Configuration*” in the ‘Spring Boot features’ section for the exact order).

A nice way to augment and modify this ordering is to add `@PropertySource` annotations to your application sources. Classes passed to the `SpringApplication` static convenience methods and those added using `setSources()` are inspected to see if they have `@PropertySources`. If they do, those properties are added to the `Environment` early enough to be used in all phases of the `ApplicationContext` lifecycle. Properties added in this way have lower priority than any added by using the default locations (such as `application.properties`), system properties, environment variables, or the command line.

You can also provide the following System properties (or environment variables) to change the behavior:

- `spring.config.name` (`SPRING_CONFIG_NAME`): Defaults to `application` as the root of the file name.
- `spring.config.location` (`SPRING_CONFIG_LOCATION`): The file to load (such as a classpath resource or a URL). A separate `Environment` property source is set up for this document and it can be overridden by system properties, environment variables, or the command line.

No matter what you set in the environment, Spring Boot always loads `application.properties` as described above. By default, if YAML is used, then files with the ‘.yml’ extension are also added to the list.

Spring Boot logs the configuration files that are loaded at the `DEBUG` level and the candidates it has not found at `TRACE` level.

See `ConfigFileApplicationListener` for more detail.

73.4 Use ‘Short’ Command Line Arguments

Some people like to use (for example) `--port=9000` instead of `--server.port=9000` to set configuration properties on the command line. You can enable this behavior by using placeholders in `application.properties`, as shown in the following example:

```
server.port=${port:8080}
```

Tip

If you inherit from the `spring-boot-starter-parent` POM, the default filter token of the `maven-resources-plugins` has been changed from `${*}` to `@` (that is, `@maven.token@` instead of `${maven.token}`) to prevent conflicts with Spring-style placeholders. If you have enabled Maven filtering for the `application.properties` directly, you may want to also change the default filter token to use other delimiters.

> **Note**
>
> In this specific case, the port binding works in a PaaS environment such as Heroku or Cloud Foundry. In those two platforms, the `PORT` environment variable is set automatically and Spring can bind to capitalized synonyms for `Environment` properties.

73.5 Use YAML for External Properties

YAML is a superset of JSON and, as such, is a convenient syntax for storing external properties in a hierarchical format, as shown in the following example:

```
spring:
 application:
  name: cruncher
 datasource:
  driverClassName: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost/test
server:
 port: 9000
```

Create a file called `application.yml` and put it in the root of your classpath. Then add `snakeyaml` to your dependencies (Maven coordinates `org.yaml:snakeyaml`, already included if you use the `spring-boot-starter`). A YAML file is parsed to a Java `Map<String,Object>` (like a JSON object), and Spring Boot flattens the map so that it is one level deep and has period-separated keys, as many people are used to with `Properties` files in Java.

The preceding example YAML corresponds to the following `application.properties` file:

```
spring.application.name=cruncher
spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/test
server.port=9000
```

See “Section 24.6, “Using YAML Instead of Properties”” in the ‘Spring Boot features’ section for more information about YAML.

73.6 Set the Active Spring Profiles

The Spring `Environment` has an API for this, but you would normally set a System property (`spring.profiles.active`) or an OS environment variable (`SPRING_PROFILES_ACTIVE`). Also, you can launch your application with a `-D` argument (remember to put it before the main class or jar archive), as follows:

```
$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

In Spring Boot, you can also set the active profile in `application.properties`, as shown in the following example:

```
spring.profiles.active=production
```

A value set this way is replaced by the System property or environment variable setting but not by the `SpringApplicationBuilder.profiles()` method. Thus, the latter Java API can be used to augment the profiles without changing the defaults.

See “Chapter 25, *Profiles*” in the “Spring Boot features” section for more information.

73.7 Change Configuration Depending on the Environment

A YAML file is actually a sequence of documents separated by `---` lines, and each document is parsed separately to a flattened map.

If a YAML document contains a `spring.profiles` key, then the profiles value (a comma-separated list of profiles) is fed into the Spring `Environment.acceptsProfiles()` method. If any of those profiles is active, that document is included in the final merge (otherwise, it is not), as shown in the following example:

```yaml
server:
 port: 9000
---

spring:
 profiles: development
server:
 port: 9001

---

spring:
 profiles: production
server:
 port: 0
```

In the preceding example, the default port is 9000. However, if the Spring profile called ‘development’ is active, then the port is 9001. If ‘production’ is active, then the port is 0.

Note

The YAML documents are merged in the order in which they are encountered. Later values override earlier values.

To do the same thing with properties files, you can use `application-${profile}.properties` to specify profile-specific values.

73.8 Discover Built-in Options for External Properties

Spring Boot binds external properties from `application.properties` (or `.yml` files and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location, because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a `configprops` endpoint that shows all the bound and bindable properties available through `@ConfigurationProperties`.

The appendix includes an `application.properties` example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for `@ConfigurationProperties` and `@Value` annotations as well as the occasional use of `Binder`. For more about the exact ordering of loading properties, see "Chapter 24, *Externalized Configuration*".

74. Embedded Web Servers

Each Spring Boot web application includes an embedded web server. This feature leads to a number of how-to questions, including how to change the embedded server and how to configure the embedded server. This section answers those questions.

74.1 Use Another Web Server

Many Spring Boot starters include default embedded containers. `spring-boot-starter-web` includes Tomcat by including `spring-boot-starter-tomcat`, but you can use `spring-boot-starter-jetty` or `spring-boot-starter-undertow` instead. `spring-boot-starter-webflux` includes Reactor Netty by including `spring-boot-starter-reactor-netty`, but you can use `spring-boot-starter-tomcat`, `spring-boot-starter-jetty`, or `spring-boot-starter-undertow` instead.

> **Note**
>
> Many starters support only Spring MVC, so they transitively bring `spring-boot-starter-web` into your application classpath.

If you need to use a different HTTP server, you need to exclude the default dependencies and include the one you need. Spring Boot provides separate starters for HTTP servers to help make this process as easy as possible.

The following Maven example shows how to exclude Tomcat and include Jetty for Spring MVC:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 <exclusions>
  <!-- Exclude the Tomcat dependency -->
  <exclusion>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-tomcat</artifactId>
  </exclusion>
 </exclusions>
</dependency>
<!-- Use Jetty instead -->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

The following Gradle example shows how to exclude Netty and include Undertow for Spring WebFlux:

```
configurations {
 // exclude Reactor Netty
 compile.exclude module: 'spring-boot-starter-reactor-netty'
}

dependencies {
 compile 'org.springframework.boot:spring-boot-starter-webflux'
 // Use Undertow instead
 compile 'org.springframework.boot:spring-boot-starter-undertow'
 // ...
}
```

> **Note**
>
> `spring-boot-starter-reactor-netty` is required to use the `WebClient` class, so you may need to keep a dependency on Netty even when you need to include a different HTTP server.

74.2 Configure Jetty

Generally, you can follow the advice from "Section 73.8, "Discover Built-in Options for External Properties"" about `@ConfigurationProperties` (`ServerProperties` is the main one here). However, you should also look at `WebServerFactoryCustomizer`. The Jetty APIs are quite rich, so, once you have access to the `JettyServletWebServerFactory`, you can modify it in a number of ways. Alternatively, if you need more control and customization, you can add your own `JettyServletWebServerFactory`.

74.3 Add a Servlet, Filter, or Listener to an Application

There are two ways to add `Servlet`, `Filter`, `ServletContextListener`, and the other listeners supported by the Servlet spec to your application:

- the section called "Add a Servlet, Filter, or Listener by Using a Spring Bean"
- the section called "Add Servlets, Filters, and Listeners by Using Classpath Scanning"

Add a Servlet, Filter, or Listener by Using a Spring Bean

To add a `Servlet`, `Filter`, or Servlet `*Listener` by using a Spring bean, you must provide a `@Bean` definition for it. Doing so can be very useful when you want to inject configuration or dependencies. However, you must be very careful that they do not cause eager initialization of too many other beans, because they have to be installed in the container very early in the application lifecycle. (For example, it is not a good idea to have them depend on your `DataSource` or JPA configuration.) You can work around such restrictions by initializing the beans lazily when first used instead of on initialization.

In the case of `Filters` and `Servlets`, you can also add mappings and init parameters by adding a `FilterRegistrationBean` or a `ServletRegistrationBean` instead of or in addition to the underlying component.

> **Note**
>
> If no `dispatcherType` is specified on a filter registration, `REQUEST` is used. This aligns with the Servlet Specification's default dispatcher type.

Like any other Spring bean, you can define the order of Servlet filter beans; please make sure to check the "the section called "Registering Servlets, Filters, and Listeners as Spring Beans"" section.

Disable Registration of a Servlet or Filter

As described earlier, any `Servlet` or `Filter` beans are registered with the servlet container automatically. To disable registration of a particular `Filter` or `Servlet` bean, create a registration bean for it and mark it as disabled, as shown in the following example:

```
@Bean
public FilterRegistrationBean registration(MyFilter filter) {
 FilterRegistrationBean registration = new FilterRegistrationBean(filter);
 registration.setEnabled(false);
 return registration;
}
```

Add Servlets, Filters, and Listeners by Using Classpath Scanning

`@WebServlet`, `@WebFilter`, and `@WebListener` annotated classes can be automatically registered with an embedded servlet container by annotating a `@Configuration` class with `@ServletComponentScan` and specifying the package(s) containing the components that you want to register. By default, `@ServletComponentScan` scans from the package of the annotated class.

74.4 Change the HTTP Port

In a standalone application, the main HTTP port defaults to `8080` but can be set with `server.port` (for example, in `application.properties` or as a System property). Thanks to relaxed binding of `Environment` values, you can also use `SERVER_PORT` (for example, as an OS environment variable).

To switch off the HTTP endpoints completely but still create a `WebApplicationContext`, use `server.port=-1`. (Doing so is sometimes useful for testing.)

For more details, see "the section called "Customizing Embedded Servlet Containers"" in the 'Spring Boot features' section, or the `ServerProperties` source code.

74.5 Use a Random Unassigned HTTP Port

To scan for a free port (using OS natives to prevent clashes) use `server.port=0`.

74.6 Discover the HTTP Port at Runtime

You can access the port the server is running on from log output or from the `ServletWebServerApplicationContext` through its `WebServer`. The best way to get that and be sure that it has been initialized is to add a `@Bean` of type `ApplicationListener<ServletWebServerInitializedEvent>` and pull the container out of the event when it is published.

Tests that use `@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)` can also inject the actual port into a field by using the `@LocalServerPort` annotation, as shown in the following example:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class MyWebIntegrationTests {

 @Autowired
 ServletWebServerApplicationContext server;

 @LocalServerPort
 int port;

 // ...

}
```

Note

`@LocalServerPort` is a meta-annotation for `@Value("${local.server.port}")`. Do not try to inject the port in a regular application. As we just saw, the value is set only after the container

> has been initialized. Contrary to a test, application code callbacks are processed early (before the value is actually available).

74.7 Configure SSL

SSL can be configured declaratively by setting the various `server.ssl.*` properties, typically in `application.properties` or `application.yml`. The following example shows setting SSL properties in `application.properties`:

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=another-secret
```

See `Ssl` for details of all of the supported properties.

Using configuration such as the preceding example means the application no longer supports a plain HTTP connector at port 8080. Spring Boot does not support the configuration of both an HTTP connector and an HTTPS connector through `application.properties`. If you want to have both, you need to configure one of them programmatically. We recommend using `application.properties` to configure HTTPS, as the HTTP connector is the easier of the two to configure programmatically. See the `spring-boot-sample-tomcat-multi-connectors` sample project for an example.

74.8 Configure HTTP/2

You can enable HTTP/2 support in your Spring Boot application with the `server.http2.enabled` configuration property. This support depends on the chosen web server and the application environment, since that protocol is not supported out-of-the-box by JDK8.

> **Note**
>
> Spring Boot does not support `h2c`, the cleartext version of the HTTP/2 protocol. So you must configure SSL first.

HTTP/2 with Undertow

As of Undertow 1.4.0+, HTTP/2 is supported without any additional requirement on JDK8.

HTTP/2 with Jetty

As of Jetty 9.4.8, HTTP/2 is also supported with the Conscrypt library. To enable that support, your application needs to have two additional dependencies: `org.eclipse.jetty:jetty-alpn-conscrypt-server` and `org.eclipse.jetty.http2:http2-server`.

HTTP/2 with Tomcat

Spring Boot ships by default with Tomcat 8.5.x. With that version, HTTP/2 is only supported if the `libtcnative` library and its dependencies are installed on the host operating system.

The library folder must be made available, if not already, to the JVM library path. You can do so with a JVM argument such as `-Djava.library.path=/usr/local/opt/tomcat-native/lib`. More on this in the official Tomcat documentation.

Starting Tomcat 8.5.x without that native support logs the following error:

```
ERROR 8787 --- [           main] o.a.coyote.http11.Http11NioProtocol      : The upgrade handler
 [org.apache.coyote.http2.Http2Protocol] for [h2] only supports upgrade via ALPN but has been configured
 for the ["https-jsse-nio-8443"] connector that does not support ALPN.
```

This error is not fatal, and the application still starts with HTTP/1.1 SSL support.

Running your application with Tomcat 9.0.x and JDK9 does not require any native library to be installed. To use Tomcat 9, you can override the `tomcat.version` build property with the version of your choice.

74.9 Configure Access Logging

Access logs can be configured for Tomcat, Undertow, and Jetty through their respective namespaces.

For instance, the following settings log access on Tomcat with a custom pattern.

```
server.tomcat.basedir=my-tomcat
server.tomcat.accesslog.enabled=true
server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)
```

Note

The default location for logs is a `logs` directory relative to the Tomcat base directory. By default, the `logs` directory is a temporary directory, so you may want to fix Tomcat's base directory or use an absolute path for the logs. In the preceding example, the logs are available in `my-tomcat/logs` relative to the working directory of the application.

Access logging for Undertow can be configured in a similar fashion, as shown in the following example:

```
server.undertow.accesslog.enabled=true
server.undertow.accesslog.pattern=%t %a "%r" %s (%D ms)
```

Logs are stored in a `logs` directory relative to the working directory of the application. You can customize this location by setting the `server.undertow.accesslog.directory` property.

Finally, access logging for Jetty can also be configured as follows:

```
server.jetty.accesslog.enabled=true
server.jetty.accesslog.filename=/var/log/jetty-access.log
```

By default, logs are redirected to `System.err`. For more details, see the Jetty documentation.

74.10 Running Behind a Front-end Proxy Server

Your application might need to send `302` redirects or render content with absolute links back to itself. When running behind a proxy, the caller wants a link to the proxy and not to the physical address of the machine hosting your app. Typically, such situations are handled through a contract with the proxy, which adds headers to tell the back end how to construct links to itself.

If the proxy adds conventional `X-Forwarded-For` and `X-Forwarded-Proto` headers (most proxy servers do so), the absolute links should be rendered correctly, provided `server.use-forward-headers` is set to `true` in your `application.properties`.

Note

If your application runs in Cloud Foundry or Heroku, the `server.use-forward-headers` property defaults to `true`. In all other instances, it defaults to `false`.

Customize Tomcat’s Proxy Configuration

If you use Tomcat, you can additionally configure the names of the headers used to carry “forwarded” information, as shown in the following example:

```
server.tomcat.remote-ip-header=x-your-remote-ip-header
server.tomcat.protocol-header=x-your-protocol-header
```

Tomcat is also configured with a default regular expression that matches internal proxies that are to be trusted. By default, IP addresses in `10/8`, `192.168/16`, `169.254/16` and `127/8` are trusted. You can customize the valve’s configuration by adding an entry to `application.properties`, as shown in the following example:

```
server.tomcat.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}
```

Note

The double backslashes are required only when you use a properties file for configuration. If you use YAML, single backslashes are sufficient, and a value equivalent to that shown in the preceding example would be `192\.168\.\d{1,3}\.\d{1,3}`.

Note

You can trust all proxies by setting the `internal-proxies` to empty (but do not do so in production).

You can take complete control of the configuration of Tomcat’s `RemoteIpValve` by switching the automatic one off (to do so, set `server.use-forward-headers=false`) and adding a new valve instance in a `TomcatServletWebServerFactory` bean.

74.11 Configure Tomcat

Generally, you can follow the advice from “Section 73.8, “Discover Built-in Options for External Properties”” about `@ConfigurationProperties` (`ServerProperties` is the main one here). However, you should also look at `WebServerFactoryCustomizer` and various Tomcat-specific `*Customizers` that you can add. The Tomcat APIs are quite rich. Consequently, once you have access to the `TomcatServletWebServerFactory`, you can modify it in a number of ways. Alternatively, if you need more control and customization, you can add your own `TomcatServletWebServerFactory`.

74.12 Enable Multiple Connectors with Tomcat

You can add an `org.apache.catalina.connector.Connector` to the `TomcatServletWebServerFactory`, which can allow multiple connectors, including HTTP and HTTPS connectors, as shown in the following example:

```
@Bean
public ServletWebServerFactory servletContainer() {
 TomcatServletWebServerFactory tomcat = new TomcatServletWebServerFactory();
 tomcat.addAdditionalTomcatConnectors(createSslConnector());
 return tomcat;
}

private Connector createSslConnector() {
 Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
 Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();
 try {
  File keystore = new ClassPathResource("keystore").getFile();
  File truststore = new ClassPathResource("keystore").getFile();
  connector.setScheme("https");
  connector.setSecure(true);
  connector.setPort(8443);
  protocol.setSSLEnabled(true);
  protocol.setKeystoreFile(keystore.getAbsolutePath());
  protocol.setKeystorePass("changeit");
  protocol.setTruststoreFile(truststore.getAbsolutePath());
  protocol.setTruststorePass("changeit");
  protocol.setKeyAlias("apitester");
  return connector;
 }
 catch (IOException ex) {
  throw new IllegalStateException("can't access keystore: [" + "keystore"
    + "] or truststore: [" + "keystore" + "]", ex);
 }
}
```

74.13 Use Tomcat's LegacyCookieProcessor

By default, the embedded Tomcat used by Spring Boot does not support "Version 0" of the Cookie format, so you may see the following error:

```
java.lang.IllegalArgumentException: An invalid character [32] was present in the Cookie value
```

If at all possible, you should consider updating your code to only store values compliant with later Cookie specifications. If, however, you cannot change the way that cookies are written, you can instead configure Tomcat to use a `LegacyCookieProcessor`. To switch to the `LegacyCookieProcessor`, use an `WebServerFactoryCustomizer` bean that adds a `TomcatContextCustomizer`, as shown in the following example:

```
@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory> cookieProcessorCustomizer() {
 return (factory) -> factory.addContextCustomizers(
   (context) -> context.setCookieProcessor(new LegacyCookieProcessor()));
}
```

74.14 Configure Undertow

Generally you can follow the advice from "Section 73.8, "Discover Built-in Options for External Properties"" about `@ConfigurationProperties` (`ServerProperties` and `ServerProperties.Undertow` are the main ones here). However, you should also look at `WebServerFactoryCustomizer`. Once you have access to the `UndertowServletWebServerFactory`, you can use an `UndertowBuilderCustomizer` to modify Undertow's configuration to meet your needs. Alternatively, if you need more control and customization, you can add your own `UndertowServletWebServerFactory`.

74.15 Enable Multiple Listeners with Undertow

Add an `UndertowBuilderCustomizer` to the `UndertowServletWebServerFactory` and add a listener to the `Builder`, as shown in the following example:

```
@Bean
public UndertowServletWebServerFactory servletWebServerFactory() {
 UndertowServletWebServerFactory factory = new UndertowServletWebServerFactory();
 factory.addBuilderCustomizers(new UndertowBuilderCustomizer() {

  @Override
```

```
   public void customize(Builder builder) {
    builder.addHttpListener(8080, "0.0.0.0");
   }

  });
  return factory;
 }
```

74.16 Create WebSocket Endpoints Using @ServerEndpoint

If you want to use `@ServerEndpoint` in a Spring Boot application that used an embedded container, you must declare a single `ServerEndpointExporter` `@Bean`, as shown in the following example:

```
@Bean
public ServerEndpointExporter serverEndpointExporter() {
 return new ServerEndpointExporter();
}
```

The bean shown in the preceding example registers any `@ServerEndpoint` annotated beans with the underlying WebSocket container. When deployed to a standalone servlet container, this role is performed by a servlet container initializer, and the `ServerEndpointExporter` bean is not required.

74.17 Enable HTTP Response Compression

HTTP response compression is supported by Jetty, Tomcat, and Undertow. It can be enabled in `application.properties`, as follows:

```
server.compression.enabled=true
```

By default, responses must be at least 2048 bytes in length for compression to be performed. You can configure this behavior by setting the `server.compression.min-response-size` property.

By default, responses are compressed only if their content type is one of the following:

- `text/html`
- `text/xml`
- `text/plain`
- `text/css`

You can configure this behavior by setting the `server.compression.mime-types` property.

75. Spring MVC

Spring Boot has a number of starters that include Spring MVC. Note that some starters include a dependency on Spring MVC rather than include it directly. This section answers common questions about Spring MVC and Spring Boot.

75.1 Write a JSON REST Service

Any Spring `@RestController` in a Spring Boot application should render JSON response by default as long as Jackson2 is on the classpath, as shown in the following example:

```
@RestController
public class MyController {

 @RequestMapping("/thing")
 public MyThing thing() {
   return new MyThing();
 }

}
```

As long as `MyThing` can be serialized by Jackson2 (true for a normal POJO or Groovy object), then `localhost:8080/thing` serves a JSON representation of it by default. Note that, in a browser, you might sometimes see XML responses, because browsers tend to send accept headers that prefer XML.

75.2 Write an XML REST Service

If you have the Jackson XML extension (`jackson-dataformat-xml`) on the classpath, you can use it to render XML responses. The previous example that we used for JSON would work. To use the Jackson XML renderer, add the following dependency to your project:

```
<dependency>
 <groupId>com.fasterxml.jackson.dataformat</groupId>
 <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

You may also want to add a dependency on Woodstox. It is faster than the default StAX implementation provided by the JDK and also adds pretty-print support and improved namespace handling. The following listing shows how to include a dependency on Woodstox:

```
<dependency>
 <groupId>org.codehaus.woodstox</groupId>
 <artifactId>woodstox-core-asl</artifactId>
</dependency>
```

If Jackson's XML extension is not available, JAXB (provided by default in the JDK) is used, with the additional requirement of having `MyThing` annotated as `@XmlRootElement`, as shown in the following example:

```
@XmlRootElement
public class MyThing {
 private String name;
 // .. getters and setters
}
```

To get the server to render XML instead of JSON, you might have to send an `Accept: text/xml` header (or use a browser).

75.3 Customize the Jackson ObjectMapper

Spring MVC (client and server side) uses `HttpMessageConverters` to negotiate content conversion in an HTTP exchange. If Jackson is on the classpath, you already get the default converter(s) provided by `Jackson2ObjectMapperBuilder`, an instance of which is auto-configured for you.

The `ObjectMapper` (or `XmlMapper` for Jackson XML converter) instance (created by default) has the following customized properties:

- `MapperFeature.DEFAULT_VIEW_INCLUSION` is disabled
- `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` is disabled

Spring Boot also has some features to make it easier to customize this behavior.

You can configure the `ObjectMapper` and `XmlMapper` instances by using the environment. Jackson provides an extensive suite of simple on/off features that can be used to configure various aspects of its processing. These features are described in six enums (in Jackson) that map onto properties in the environment:

Jackson enum	**Environment property**
`com.fasterxml.jackson.databind.DeserializationFeature`	`spring.jackson.deserialization.<feature_name>=tru false`
`com.fasterxml.jackson.core.JsonGenerator.Feature`	`spring.jackson.generator.<feature_name>=true|false`
`com.fasterxml.jackson.databind.MapperFeature`	`spring.jackson.mapper.<feature_name>=true|false`
`com.fasterxml.jackson.core.JsonParser.Feature`	`spring.jackson.parser.<feature_name>=true|false`
`com.fasterxml.jackson.databind.SerializationFeature`	`spring.jackson.serialization.<feature_name>=true|false`
`com.fasterxml.jackson.annotation.JsonInclude.Include`	`spring.jackson.default-property-inclusion=always|non_null|non_absent|non_default|non_empty`

For example, to enable pretty print, set `spring.jackson.serialization.indent_output=true`. Note that, thanks to the use of relaxed binding, the case of `indent_output` does not have to match the case of the corresponding enum constant, which is `INDENT_OUTPUT`.

This environment-based configuration is applied to the auto-configured `Jackson2ObjectMapperBuilder` bean and applies to any mappers created by using the builder, including the auto-configured `ObjectMapper` bean.

The context's `Jackson2ObjectMapperBuilder` can be customized by one or more `Jackson2ObjectMapperBuilderCustomizer` beans. Such customizer beans can be ordered (Boot's own customizer has an order of 0), letting additional customization be applied both before and after Boot's customization.

Any beans of type `com.fasterxml.jackson.databind.Module` are automatically registered with the auto-configured `Jackson2ObjectMapperBuilder` and are applied to any `ObjectMapper` instances that it creates. This provides a global mechanism for contributing custom modules when you add new features to your application.

If you want to replace the default `ObjectMapper` completely, either define a `@Bean` of that type and mark it as `@Primary` or, if you prefer the builder-based approach, define a `Jackson2ObjectMapperBuilder` `@Bean`. Note that, in either case, doing so disables all auto-configuration of the `ObjectMapper`.

If you provide any `@Beans` of type `MappingJackson2HttpMessageConverter`, they replace the default value in the MVC configuration. Also, a convenience bean of type `HttpMessageConverters` is provided (and is always available if you use the default MVC configuration). It has some useful methods to access the default and user-enhanced message converters.

See the "Section 75.4, "Customize the @ResponseBody Rendering"" section and the `WebMvcAutoConfiguration` source code for more details.

75.4 Customize the @ResponseBody Rendering

Spring uses `HttpMessageConverters` to render `@ResponseBody` (or responses from `@RestController`). You can contribute additional converters by adding beans of the appropriate type in a Spring Boot context. If a bean you add is of a type that would have been included by default anyway (such as `MappingJackson2HttpMessageConverter` for JSON conversions), it replaces the default value. A convenience bean of type `HttpMessageConverters` is provided and is always available if you use the default MVC configuration. It has some useful methods to access the default and user-enhanced message converters (For example, it can be useful if you want to manually inject them into a custom `RestTemplate`).

As in normal MVC usage, any `WebMvcConfigurer` beans that you provide can also contribute converters by overriding the `configureMessageConverters` method. However, unlike with normal MVC, you can supply only additional converters that you need (because Spring Boot uses the same mechanism to contribute its defaults). Finally, if you opt out of the Spring Boot default MVC configuration by providing your own `@EnableWebMvc` configuration, you can take control completely and do everything manually by using `getMessageConverters` from `WebMvcConfigurationSupport`.

See the `WebMvcAutoConfiguration` source code for more details.

75.5 Handling Multipart File Uploads

Spring Boot embraces the Servlet 3 `javax.servlet.http.Part` API to support uploading files. By default, Spring Boot configures Spring MVC with a maximum size of 1MB per file and a maximum of 10MB of file data in a single request. You may override these values, the location to which intermediate data is stored (for example, to the `/tmp` directory), and the threshold past which data is flushed to disk by using the properties exposed in the `MultipartProperties` class. For example, if you want to specify that files be unlimited, set the `spring.servlet.multipart.max-file-size` property to `-1`.

The multipart support is helpful when you want to receive multipart encoded file data as a `@RequestParam`-annotated parameter of type `MultipartFile` in a Spring MVC controller handler method.

See the `MultipartAutoConfiguration` source for more details.

75.6 Switch Off the Spring MVC DispatcherServlet

Spring Boot wants to serve all content from the root of your application (`/`) down. If you would rather map your own servlet to that URL, you can do it. However, you may lose some of the other Boot MVC features. To add your own servlet and map it to the root resource, declare a `@Bean` of type `Servlet` and give it the special bean name, `dispatcherServlet`. (You can also create a bean of a different type with that name if you want to switch it off and not replace it.)

75.7 Switch off the Default MVC Configuration

The easiest way to take complete control over MVC configuration is to provide your own `@Configuration` with the `@EnableWebMvc` annotation. Doing so leaves all MVC configuration in your hands.

75.8 Customize ViewResolvers

A `ViewResolver` is a core component of Spring MVC, translating view names in `@Controller` to actual `View` implementations. Note that `ViewResolvers` are mainly used in UI applications, rather than REST-style services (a `View` is not used to render a `@ResponseBody`). There are many implementations of `ViewResolver` to choose from, and Spring on its own is not opinionated about which ones you should use. Spring Boot, on the other hand, installs one or two for you, depending on what it finds on the classpath and in the application context. The `DispatcherServlet` uses all the resolvers it finds in the application context, trying each one in turn until it gets a result, so, if you add your own, you have to be aware of the order and in which position your resolver is added.

`WebMvcAutoConfiguration` adds the following `ViewResolvers` to your context:

- An `InternalResourceViewResolver` named ‘defaultViewResolver’. This one locates physical resources that can be rendered by using the `DefaultServlet` (including static resources and JSP pages, if you use those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (the defaults are both empty but are accessible for external configuration through `spring.mvc.view.prefix` and `spring.mvc.view.suffix`). You can override it by providing a bean of the same type.
- A `BeanNameViewResolver` named ‘beanNameViewResolver’. This is a useful member of the view resolver chain and picks up any beans with the same name as the `View` being resolved. It should not be necessary to override or replace it.
- A `ContentNegotiatingViewResolver` named ‘viewResolver’ is added only if there **are** actually beans of type `View` present. This is a ‘master’ resolver, delegating to all the others and attempting to find a match to the ‘Accept’ HTTP header sent by the client. There is a useful blog about `ContentNegotiatingViewResolver` that you might like to study to learn more, and you might also look at the source code for detail. You can switch off the auto-configured `ContentNegotiatingViewResolver` by defining a bean named ‘viewResolver’.
- If you use Thymeleaf, you also have a `ThymeleafViewResolver` named ‘thymeleafViewResolver’. It looks for resources by surrounding the view name with a prefix and suffix. The prefix is `spring.thymeleaf.prefix`, and the suffix is `spring.thymeleaf.suffix`. The values of the prefix and suffix default to ‘classpath:/templates/’ and ‘.html’, respectively. You can override `ThymeleafViewResolver` by providing a bean of the same name.
- If you use FreeMarker, you also have a `FreeMarkerViewResolver` named ‘freeMarkerViewResolver’. It looks for resources in a loader path (which is externalized to

`spring.freemarker.templateLoaderPath` and has a default value of ‘classpath:/templates/’) by surrounding the view name with a prefix and a suffix. The prefix is externalized to `spring.freemarker.prefix`, and the suffix is externalized to `spring.freemarker.suffix`. The default values of the prefix and suffix are empty and ‘.ftl’, respectively. You can override `FreeMarkerViewResolver` by providing a bean of the same name.

- If you use Groovy templates (actually, if `groovy-templates` is on your classpath), you also have a `GroovyMarkupViewResolver` named ‘groovyMarkupViewResolver’. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to `spring.groovy.template.prefix` and `spring.groovy.template.suffix`). The prefix and suffix have default values of ‘classpath:/templates/’ and ‘.tpl’, respectively. You can override `GroovyMarkupViewResolver` by providing a bean of the same name.

For more detail, see the following sections:

- `WebMvcAutoConfiguration`
- `ThymeleafAutoConfiguration`
- `FreeMarkerAutoConfiguration`
- `GroovyTemplateAutoConfiguration`

76. HTTP Clients

Spring Boot offers a number of starters that work with HTTP clients. This section answers questions related to using them.

76.1 Configure RestTemplate to Use a Proxy

As described in Section 33.1, “RestTemplate Customization”, you can use a `RestTemplateCustomizer` with `RestTemplateBuilder` to build a customized `RestTemplate`. This is the recommended approach for creating a `RestTemplate` configured to use a proxy.

The exact details of the proxy configuration depend on the underlying client request factory that is being used. The following example configures `HttpComponentsClientRequestFactory` with an `HttpClient` that uses a proxy for all hosts except `192.168.0.5`:

```
static class ProxyCustomizer implements RestTemplateCustomizer {

 @Override
 public void customize(RestTemplate restTemplate) {
  HttpHost proxy = new HttpHost("proxy.example.com");
  HttpClient httpClient = HttpClientBuilder.create()
    .setRoutePlanner(new DefaultProxyRoutePlanner(proxy) {

     @Override
     public HttpHost determineProxy(HttpHost target,
       HttpRequest request, HttpContext context)
         throws HttpException {
      if (target.getHostName().equals("192.168.0.5")) {
       return null;
      }
      return super.determineProxy(target, request, context);
     }

    }).build();
  restTemplate.setRequestFactory(
    new HttpComponentsClientHttpRequestFactory(httpClient));
 }

}
```

77. Logging

Spring Boot has no mandatory logging dependency, except for the Commons Logging API, of which there are many implementations to choose from. To use Logback, you need to include it and `jcl-over-slf4j` (which implements the Commons Logging API) on the classpath. The simplest way to do that is through the starters, which all depend on `spring-boot-starter-logging`. For a web application, you need only `spring-boot-starter-web`, since it depends transitively on the logging starter. If you use Maven, the following dependency adds logging for you:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Boot has a `LoggingSystem` abstraction that attempts to configure logging based on the content of the classpath. If Logback is available, it is the first choice.

If the only change you need to make to logging is to set the levels of various loggers, you can do so in `application.properties` by using the "logging.level" prefix, as shown in the following example:

```
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

You can also set the location of a file to which to write the log (in addition to the console) by using "logging.file".

To configure the more fine-grained settings of a logging system, you need to use the native configuration format supported by the `LoggingSystem` in question. By default, Spring Boot picks up the native configuration from its default location for the system (such as `classpath:logback.xml` for Logback), but you can set the location of the config file by using the "logging.config" property.

77.1 Configure Logback for Logging

If you put a `logback.xml` in the root of your classpath, it is picked up from there (or from `logback-spring.xml`, to take advantage of the templating features provided by Boot). Spring Boot provides a default base configuration that you can include if you want to set levels, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <include resource="org/springframework/boot/logging/logback/base.xml"/>
 <logger name="org.springframework.web" level="DEBUG"/>
</configuration>
```

If you look at `base.xml` in the spring-boot jar, you can see that it uses some useful System properties that the `LoggingSystem` takes care of creating for you:

- `${PID}`: The current process ID.
- `${LOG_FILE}`: Whether `logging.file` was set in Boot's external configuration.
- `${LOG_PATH}`: Whether `logging.path` (representing a directory for log files to live in) was set in Boot's external configuration.
- `${LOG_EXCEPTION_CONVERSION_WORD}`: Whether `logging.exception-conversion-word` was set in Boot's external configuration.

Spring Boot also provides some nice ANSI color terminal output on a console (but not in a log file) by using a custom Logback converter. See the default `base.xml` configuration for details.

If Groovy is on the classpath, you should be able to configure Logback with `logback.groovy` as well. If present, this setting is given preference.

Configure Logback for File-only Output

If you want to disable console logging and write output only to a file, you need a custom `logback-spring.xml` that imports `file-appender.xml` but not `console-appender.xml`, as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <include resource="org/springframework/boot/logging/logback/defaults.xml" />
 <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/
tmp}}/}spring.log}"/>
 <include resource="org/springframework/boot/logging/logback/file-appender.xml" />
 <root level="INFO">
  <appender-ref ref="FILE" />
 </root>
</configuration>
```

You also need to add `logging.file` to your `application.properties`, as shown in the following example:

```
logging.file=myapplication.log
```

77.2 Configure Log4j for Logging

Spring Boot supports Log4j 2 for logging configuration if it is on the classpath. If you use the starters for assembling dependencies, you have to exclude Logback and then include log4j 2 instead. If you do not use the starters, you need to provide (at least) `jcl-over-slf4j` in addition to Log4j 2.

The simplest path is probably through the starters, even though it requires some jiggling with excludes. The following example shows how to set up the starters in Maven:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter</artifactId>
 <exclusions>
  <exclusion>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-logging</artifactId>
  </exclusion>
 </exclusions>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

> **Note**
>
> The Log4j starters gather together the dependencies for common logging requirements (such as having Tomcat use `java.util.logging` but configuring the output using Log4j 2). See the Actuator Log4j 2 samples for more detail and to see it in action.

Note

To ensure that debug logging performed using `java.util.logging` is routed into Log4j 2, configure its JDK logging adapter by setting the `java.util.logging.manager` system property to `org.apache.logging.log4j.jul.LogManager`.

Use YAML or JSON to Configure Log4j 2

In addition to its default XML configuration format, Log4j 2 also supports YAML and JSON configuration files. To configure Log4j 2 to use an alternative configuration file format, add the appropriate dependencies to the classpath and name your configuration files to match your chosen file format, as shown in the following example:

Format	Dependencies	File names
YAML	`com.fasterxml.jackson.core:jackson-databind` `com.fasterxml.jackson.dataformat:jackson-dataformat-yaml`	`log4j2.yaml` `log4j2.yml`
JSON	`com.fasterxml.jackson.core:jackson-databind`	`log4j2.json` `log4j2.jsn`

78. Data Access

Spring Boot includes a number of starters for working with data sources. This section answers questions related to doing so.

78.1 Configure a Custom DataSource

To configure your own `DataSource`, define a `@Bean` of that type in your configuration. Spring Boot reuses your `DataSource` anywhere one is required, including database initialization. If you need to externalize some settings, you can bind your `DataSource` to the environment (see "the section called “Third-party Configuration”").

The following example shows how to define a data source in a bean:

```
@Bean
@ConfigurationProperties(prefix="app.datasource")
public DataSource dataSource() {
 return new FancyDataSource();
}
```

The following example shows how to define a data source by setting properties:

```
app.datasource.url=jdbc:h2:mem:mydb
app.datasource.username=sa
app.datasource.pool-size=30
```

Assuming that your `FancyDataSource` has regular JavaBean properties for the URL, the username, and the pool size, these settings are bound automatically before the `DataSource` is made available to other components. The regular database initialization also happens (so the relevant sub-set of `spring.datasource.*` can still be used with your custom configuration).

You can apply the same principle if you configure a custom JNDI `DataSource`, as shown in the following example:

```
@Bean(destroyMethod="")
@ConfigurationProperties(prefix="app.datasource")
public DataSource dataSource() throws Exception {
 JndiDataSourceLookup dataSourceLookup = new JndiDataSourceLookup();
 return dataSourceLookup.getDataSource("java:comp/env/jdbc/YourDS");
}
```

Spring Boot also provides a utility builder class, called `DataSourceBuilder`, that can be used to create one of the standard data sources (if it is on the classpath). The builder can detect the one to use based on what's available on the classpath. It also auto-detects the driver based on the JDBC URL.

The following example shows how to create a data source by using a `DataSourceBuilder`:

```
@Bean
@ConfigurationProperties("app.datasource")
public DataSource dataSource() {
 return DataSourceBuilder.create().build();
}
```

To run an app with that `DataSource`, all you need is the connection information. Pool-specific settings can also be provided. Check the implementation that is going to be used at runtime for more details.

The following example shows how to define a JDBC data source by setting properties:

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

However, there is a catch. Because the actual type of the connection pool is not exposed, no keys are generated in the metadata for your custom `DataSource` and no completion is available in your IDE (because the `DataSource` interface exposes no properties). Also, if you happen to have Hikari on the classpath, this basic setup does not work, because Hikari has no `url` property (but does have a `jdbcUrl` property). In that case, you must rewrite your configuration as follows:

```
app.datasource.jdbc-url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.maximum-pool-size=30
```

You can fix that by forcing the connection pool to use and return a dedicated implementation rather than `DataSource`. You cannot change the implementation at runtime, but the list of options will be explicit.

The following example shows how create a `HikariDataSource` with `DataSourceBuilder`:

```
@Bean
@ConfigurationProperties("app.datasource")
public HikariDataSource dataSource() {
 return DataSourceBuilder.create().type(HikariDataSource.class).build();
}
```

You can even go further by leveraging what `DataSourceProperties` does for you — that is, by providing a default embedded database with a sensible username and password if no URL is provided. You can easily initialize a `DataSourceBuilder` from the state of any `DataSourceProperties` object, so you could also inject the DataSource that Spring Boot creates automatically. However, that would split your configuration into two namespaces: `url`, `username`, `password`, `type`, and `driver` on `spring.datasource` and the rest on your custom namespace (`app.datasource`). To avoid that, you can redefine a custom `DataSourceProperties` on your custom namespace, as shown in the following example:

```
@Bean
@Primary
@ConfigurationProperties("app.datasource")
public DataSourceProperties dataSourceProperties() {
 return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource")
public HikariDataSource dataSource(DataSourceProperties properties) {
 return properties.initializeDataSourceBuilder().type(HikariDataSource.class)
    .build();
}
```

This setup puts you *in sync* with what Spring Boot does for you by default, except that a dedicated connection pool is chosen (in code) and its settings are exposed in the same namespace. Because `DataSourceProperties` is taking care of the `url`/`jdbcUrl` translation for you, you can configure it as follows:

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.maximum-pool-size=30
```

> **Note**
>
> Because your custom configuration chooses to go with Hikari, `app.datasource.type` has no effect. In practice, the builder is initialized with whatever value you might set there and then overridden by the call to `.type()`.

See "Section 29.1, "Configure a DataSource"" in the "Spring Boot features" section and the `DataSourceAutoConfiguration` class for more details.

78.2 Configure Two DataSources

If you need to configure multiple data sources, you can apply the same tricks that are described in the previous section. You must, however, mark one of the `DataSource` instances as `@Primary`, because various auto-configurations down the road expect to be able to get one by type.

If you create your own `DataSource`, the auto-configuration backs off. In the following example, we provide the *exact* same feature set as the auto-configuration provides on the primary data source:

```
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
 return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSource firstDataSource() {
 return firstDataSourceProperties().initializeDataSourceBuilder().build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public BasicDataSource secondDataSource() {
 return DataSourceBuilder.create().type(BasicDataSource.class).build();
}
```

> **Tip**
>
> `firstDataSourceProperties` has to be flagged as `@Primary` so that the database initializer feature uses your copy (if you use the initializer).

Both data sources are also bound for advanced customizations. For instance, you could configure them as follows:

```
app.datasource.first.type=com.zaxxer.hikari.HikariDataSource
app.datasource.first.maximum-pool-size=30

app.datasource.second.url=jdbc:mysql://localhost/test
app.datasource.second.username=dbuser
app.datasource.second.password=dbpass
app.datasource.second.max-total=30
```

You can apply the same concept to the secondary `DataSource` as well, as shown in the following example:

```
@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSourceProperties firstDataSourceProperties() {
 return new DataSourceProperties();
}

@Bean
@Primary
@ConfigurationProperties("app.datasource.first")
public DataSource firstDataSource() {
 return firstDataSourceProperties().initializeDataSourceBuilder().build();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSourceProperties secondDataSourceProperties() {
 return new DataSourceProperties();
}

@Bean
@ConfigurationProperties("app.datasource.second")
public DataSource secondDataSource() {
 return secondDataSourceProperties().initializeDataSourceBuilder().build();
}
```

The preceding example configures two data sources on custom namespaces with the same logic as Spring Boot would use in auto-configuration.

78.3 Use Spring Data Repositories

Spring Data can create implementations of `@Repository` interfaces of various flavors. Spring Boot handles all of that for you, as long as those `@Repositories` are included in the same package (or a sub-package) of your `@EnableAutoConfiguration` class.

For many applications, all you need is to put the right Spring Data dependencies on your classpath (there is a `spring-boot-starter-data-jpa` for JPA and a `spring-boot-starter-data-mongodb` for Mongodb) and create some repository interfaces to handle your `@Entity` objects. Examples are in the JPA sample and the Mongodb sample.

Spring Boot tries to guess the location of your `@Repository` definitions, based on the `@EnableAutoConfiguration` it finds. To get more control, use the `@EnableJpaRepositories` annotation (from Spring Data JPA).

For more about Spring Data, see the Spring Data project page.

78.4 Separate @Entity Definitions from Spring Configuration

Spring Boot tries to guess the location of your `@Entity` definitions, based on the `@EnableAutoConfiguration` it finds. To get more control, you can use the `@EntityScan` annotation, as shown in the following example:

```
@Configuration
@EnableAutoConfiguration
@EntityScan(basePackageClasses=City.class)
public class Application {

 //...

}
```

78.5 Configure JPA Properties

Spring Data JPA already provides some vendor-independent configuration options (such as those for SQL logging), and Spring Boot exposes those options and a few more for Hibernate as external configuration properties. Some of them are automatically detected according to the context so you should not have to set them.

The `spring.jpa.hibernate.ddl-auto` is a special case, because, depending on runtime conditions, it has different defaults. If an embedded database is used and no schema manager (such as Liquibase or Flyway) is handling the `DataSource`, it defaults to `create-drop`. In all other cases, it defaults to `none`.

The dialect to use is also automatically detected based on the current `DataSource`, but you can set `spring.jpa.database` yourself if you want to be explicit and bypass that check on startup.

> **Note**
>
> Specifying a `database` leads to the configuration of a well-defined Hibernate dialect. Several databases have more than one `Dialect`, and this may not suit your needs. In that case, you can either set `spring.jpa.database` to `default` to let Hibernate figure things out or set the dialect by setting the `spring.jpa.database-platform` property.

The most common options to set are shown in the following example:

```
spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy
spring.jpa.show-sql=true
```

In addition, all properties in `spring.jpa.properties.*` are passed through as normal JPA properties (with the prefix stripped) when the local `EntityManagerFactory` is created.

> **Tip**
>
> If you need to apply advanced customization to Hibernate properties, consider registering a `HibernatePropertiesCustomizer` bean that will be invoked prior to creating the `EntityManagerFactory`. This takes precedence to anything that is applied by the auto-configuration.

78.6 Configure Hibernate Naming Strategy

Hibernate uses two different naming strategies to map names from the object model to the corresponding database names. The fully qualified class name of the physical and the implicit strategy implementations can be configured by setting the `spring.jpa.hibernate.naming.physical-strategy` and `spring.jpa.hibernate.naming.implicit-strategy` properties, respectively. Alternatively, if `ImplicitNamingStrategy` or `PhysicalNamingStrategy` beans are available in the application context, Hibernate will be automatically configured to use them.

By default, Spring Boot configures the physical naming strategy with `SpringPhysicalNamingStrategy`. This implementation provides the same table structure as Hibernate 4: all dots are replaced by underscores and camel casing is replaced by underscores as well. By default, all table names are generated in lower case, but it is possible to override that flag if your schema requires it.

For example, a `TelephoneNumber` entity is mapped to the `telephone_number` table.

If you prefer to use Hibernate 5's default instead, set the following property:

```
spring.jpa.hibernate.naming.physical-
strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

Alternatively, you can configure the following bean:

```
@Bean
public PhysicalNamingStrategy physicalNamingStrategy() {
 return new PhysicalNamingStrategyStandardImpl();
}
```

See `HibernateJpaAutoConfiguration` and `JpaBaseConfiguration` for more details.

78.7 Use a Custom EntityManagerFactory

To take full control of the configuration of the `EntityManagerFactory`, you need to add a `@Bean` named 'entityManagerFactory'. Spring Boot auto-configuration switches off its entity manager in the presence of a bean of that type.

78.8 Use Two EntityManagers

Even if the default `EntityManagerFactory` works fine, you need to define a new one. Otherwise, the presence of the second bean of that type switches off the default. To make it easy to do, you can use the convenient `EntityManagerBuilder` provided by Spring Boot. Alternatively, you can just the `LocalContainerEntityManagerFactoryBean` directly from Spring ORM, as shown in the following example:

```
// add two data sources configured as above

@Bean
public LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(
  EntityManagerFactoryBuilder builder) {
 return builder
   .dataSource(customerDataSource())
   .packages(Customer.class)
   .persistenceUnit("customers")
   .build();
}

@Bean
public LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(
  EntityManagerFactoryBuilder builder) {
 return builder
   .dataSource(orderDataSource())
   .packages(Order.class)
   .persistenceUnit("orders")
   .build();
}
```

The configuration above almost works on its own. To complete the picture, you need to configure `TransactionManagers` for the two `EntityManagers` as well. If you mark one of them as `@Primary`, it could be picked up by the default `JpaTransactionManager` in Spring Boot. The other would have to be explicitly injected into a new instance. Alternatively, you might be able to use a JTA transaction manager that spans both.

If you use Spring Data, you need to configure `@EnableJpaRepositories` accordingly, as shown in the following example:

```
@Configuration
@EnableJpaRepositories(basePackageClasses = Customer.class,
  entityManagerFactoryRef = "customerEntityManagerFactory")
public class CustomerConfiguration {
 ...
}
```

```
@Configuration
@EnableJpaRepositories(basePackageClasses = Order.class,
  entityManagerFactoryRef = "orderEntityManagerFactory")
public class OrderConfiguration {
 ...
}
```

78.9 Use a Traditional `persistence.xml` File

Spring does not require the use of XML to configure the JPA provider, and Spring Boot assumes you want to take advantage of that feature. If you prefer to use `persistence.xml`, you need to define your own `@Bean` of type `LocalEntityManagerFactoryBean` (with an ID of 'entityManagerFactory') and set the persistence unit name there.

See `JpaBaseConfiguration` for the default settings.

78.10 Use Spring Data JPA and Mongo Repositories

Spring Data JPA and Spring Data Mongo can both automatically create `Repository` implementations for you. If they are both present on the classpath, you might have to do some extra configuration to tell Spring Boot which repositories to create. The most explicit way to do that is to use the standard Spring Data `@EnableJpaRepositories` and `@EnableMongoRepositories` annotations and provide the location of your `Repository` interfaces.

There are also flags (`spring.data.*.repositories.enabled` and `spring.data.*.repositories.type`) that you can use to switch the auto-configured repositories on and off in external configuration. Doing so is useful, for instance, in case you want to switch off the Mongo repositories and still use the auto-configured `MongoTemplate`.

The same obstacle and the same features exist for other auto-configured Spring Data repository types (Elasticsearch, Solr, and others). To work with them, change the names of the annotations and flags accordingly.

78.11 Expose Spring Data Repositories as REST Endpoint

Spring Data REST can expose the `Repository` implementations as REST endpoints for you, provided Spring MVC has been enabled for the application.

Spring Boot exposes a set of useful properties (from the `spring.data.rest` namespace) that customize the `RepositoryRestConfiguration`. If you need to provide additional customization, you should use a `RepositoryRestConfigurer` bean.

Note

If you do not specify any order on your custom `RepositoryRestConfigurer`, it runs after the one Spring Boot uses internally. If you need to specify an order, make sure it is higher than 0.

78.12 Configure a Component that is Used by JPA

If you want to configure a component that JPA uses, then you need to ensure that the component is initialized before JPA. When the component is auto-configured, Spring Boot takes care of this for you. For example, when Flyway is auto-configured, Hibernate is configured to depend upon Flyway so that Flyway has a chance to initialize the database before Hibernate tries to use it.

If you are configuring a component yourself, you can use an `EntityManagerFactoryDependsOnPostProcessor` subclass as a convenient way of setting up the necessary dependencies. For example, if you use Hibernate Search with Elasticsearch as its index manager, any `EntityManagerFactory` beans must be configured to depend on the `elasticsearchClient` bean, as shown in the following example:

```
/**
 * {@link EntityManagerFactoryDependsOnPostProcessor} that ensures that
 * {@link EntityManagerFactory} beans depend on the {@code elasticsearchClient} bean.
 */
@Configuration
static class ElasticsearchJpaDependencyConfiguration
  extends EntityManagerFactoryDependsOnPostProcessor {

 ElasticsearchJpaDependencyConfiguration() {
  super("elasticsearchClient");
 }

}
```

78.13 Configure jOOQ with Two DataSources

If you need to use jOOQ with multiple data sources, you should create your own `DSLContext` for each one. Refer to JooqAutoConfiguration for more details.

Tip

In particular, `JooqExceptionTranslator` and `SpringTransactionProvider` can be reused to provide similar features to what the auto-configuration does with a single `DataSource`.

79. Database Initialization

An SQL database can be initialized in different ways depending on what your stack is. Of course, you can also do it manually, provided the database is a separate process.

79.1 Initialize a Database Using JPA

JPA has features for DDL generation, and these can be set up to run on startup against the database. This is controlled through two external properties:

- `spring.jpa.generate-ddl` (boolean) switches the feature on and off and is vendor independent.
- `spring.jpa.hibernate.ddl-auto` (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. This feature is described in more detail later in this guide.

79.2 Initialize a Database Using Hibernate

You can set `spring.jpa.hibernate.ddl-auto` explicitly and the standard Hibernate property values are `none`, `validate`, `update`, `create`, and `create-drop`. Spring Boot chooses a default value for you based on whether it thinks your database is embedded. It defaults to `create-drop` if no schema manager has been detected or `none` in all other cases. An embedded database is detected by looking at the `Connection` type. `hsqldb`, `h2`, and `derby` are embedded, and others are not. Be careful when switching from in-memory to a ‘real’ database that you do not make assumptions about the existence of the tables and data in the new platform. You either have to set `ddl-auto` explicitly or use one of the other mechanisms to initialize the database.

> **Note**
>
> You can output the schema creation by enabling the `org.hibernate.SQL` logger. This is done for you automatically if you enable the debug mode.

In addition, a file named `import.sql` in the root of the classpath is executed on startup if Hibernate creates the schema from scratch (that is, if the `ddl-auto` property is set to `create` or `create-drop`). This can be useful for demos and for testing if you are careful but is probably not something you want to be on the classpath in production. It is a Hibernate feature (and has nothing to do with Spring).

79.3 Initialize a Database

Spring Boot can automatically create the schema (DDL scripts) of your `DataSource` and initialize it (DML scripts). It loads SQL from the standard root classpath locations: `schema.sql` and `data.sql`, respectively. In addition, Spring Boot processes the `schema-${platform}.sql` and `data-${platform}.sql` files (if present), where `platform` is the value of `spring.datasource.platform`. This allows you to switch to database-specific scripts if necessary. For example, you might choose to set it to the vendor name of the database (`hsqldb`, `h2`, `oracle`, `mysql`, `postgresql`, and so on).

Spring Boot automatically creates the schema of an embedded `DataSource`. This behavior can be customized by using the `spring.datasource.initialization-mode` property (and it can also be `always` or `never`).

By default, Spring Boot enables the fail-fast feature of the Spring JDBC initializer. This means that, if the scripts cause exceptions, the application fails to start. You can tune that behavior by setting `spring.datasource.continue-on-error`.

> **Note**
>
> In a JPA-based app, you can choose to let Hibernate create the schema or use `schema.sql`, but you cannot do both. Make sure to disable `spring.jpa.hibernate.ddl-auto` if you use `schema.sql`.

79.4 Initialize a Spring Batch Database

If you use Spring Batch, it comes pre-packaged with SQL initialization scripts for most popular database platforms. Spring Boot can detect your database type and execute those scripts on startup. If you use an embedded database, this happens by default. You can also enable it for any database type, as shown in the following example:

```
spring.batch.initialize-schema=always
```

You can also switch off the initialization explicitly by setting `spring.batch.initialize-schema=never`.

79.5 Use a Higher-level Database Migration Tool

Spring Boot supports two higher-level migration tools: Flyway and Liquibase.

Execute Flyway Database Migrations on Startup

To automatically run Flyway database migrations on startup, add the `org.flywaydb:flyway-core` to your classpath.

The migrations are scripts in the form `V<VERSION>__<NAME>.sql` (with `<VERSION>` an underscore-separated version, such as ‘1’ or ‘2_1’). By default, they are in a folder called `classpath:db/migration`, but you can modify that location by setting `spring.flyway.locations`. You can also add a special `{vendor}` placeholder to use vendor-specific scripts. Assume the following:

```
spring.flyway.locations=db/migration/{vendor}
```

Rather than using `db/migration`, the preceding configuration sets the folder to use according to the type of the database (such as `db/migration/mysql` for MySQL). The list of supported databases is available in `DatabaseDriver`.

See the Flyway class from flyway-core for details of available settings such as schemas and others. In addition, Spring Boot provides a small set of properties (in `FlywayProperties`) that can be used to disable the migrations or switch off the location checking. Spring Boot calls `Flyway.migrate()` to perform the database migration. If you would like more control, provide a `@Bean` that implements `FlywayMigrationStrategy`.

Flyway supports SQL and Java callbacks. To use SQL-based callbacks, place the callback scripts in the `classpath:db/migration` folder. To use Java-based callbacks, create one or more beans that implement `FlywayCallback` or, preferably, extend `BaseFlywayCallback`. Any such beans are automatically registered with `Flyway`. They can be ordered by using `@Order` or by implementing `Ordered`.

By default, Flyway autowires the (`@Primary`) `DataSource` in your context and uses that for migrations. If you like to use a different `DataSource`, you can create one and mark its `@Bean` as `@FlywayDataSource`. If you do so and want two data sources, remember to create another one and mark it as `@Primary`. Alternatively, you can use Flyway's native `DataSource` by setting `spring.flyway.[url,user,password]` in external properties. Setting either `spring.flyway.url` or `spring.flyway.user` is sufficient to cause Flyway to use its own `DataSource`. If any of the three properties has not be set, the value of its equivalent `spring.datasource` property will be used.

There is a Flyway sample so that you can see how to set things up.

You can also use Flyway to provide data for specific scenarios. For example, you can place test-specific migrations in `src/test/resources` and they are run only when your application starts for testing. Also, you can use profile-specific configuration to customize `spring.flyway.locations` so that certain migrations run only when a particular profile is active. For example, in `application-dev.properties`, you might specify the following setting:

```
spring.flyway.locations=classpath:/db/migration,classpath:/dev/db/migration
```

With that setup, migrations in `dev/db/migration` run only when the `dev` profile is active.

Execute Liquibase Database Migrations on Startup

To automatically run Liquibase database migrations on startup, add the `org.liquibase:liquibase-core` to your classpath.

By default, the master change log is read from `db/changelog/db.changelog-master.yaml`, but you can change the location by setting `spring.liquibase.change-log`. In addition to YAML, Liquibase also supports JSON, XML, and SQL change log formats.

By default, Liquibase autowires the (`@Primary`) `DataSource` in your context and uses that for migrations. If you need to use a different `DataSource`, you can create one and mark its `@Bean` as `@LiquibaseDataSource`. If you do so and you want two data sources, remember to create another one and mark it as `@Primary`. Alternatively, you can use Liquibase's native `DataSource` by setting `spring.liquibase.[url,user,password]` in external properties. Setting either `spring.liquibase.url` or `spring.liquibase.user` is sufficient to cause Liquibase to use its own `DataSource`. If any of the three properties has not be set, the value of its equivalent `spring.datasource` property will be used.

See `LiquibaseProperties` for details about available settings such as contexts, the default schema, and others.

There is a Liquibase sample so that you can see how to set things up.

80. Messaging

Spring Boot offers a number of starters that include messaging. This section answers questions that arise from using messaging with Spring Boot.

80.1 Disable Transacted JMS Session

If your JMS broker does not support transacted sessions, you have to disable the support of transactions altogether. If you create your own `JmsListenerContainerFactory`, there is nothing to do, since, by default it cannot be transacted. If you want to use the `DefaultJmsListenerContainerFactoryConfigurer` to reuse Spring Boot's default, you can disable transacted sessions, as follows:

```
@Bean
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory(
  ConnectionFactory connectionFactory,
  DefaultJmsListenerContainerFactoryConfigurer configurer) {
 DefaultJmsListenerContainerFactory listenerFactory =
   new DefaultJmsListenerContainerFactory();
 configurer.configure(listenerFactory, connectionFactory);
 listenerFactory.setTransactionManager(null);
 listenerFactory.setSessionTransacted(false);
 return listenerFactory;
}
```

The preceding example overrides the default factory, and it should be applied to any other factory that your application defines, if any.

81. Batch Applications

This section answers questions that arise from using Spring Batch with Spring Boot.

> **Note**
>
> By default, batch applications require a `DataSource` to store job details. If you want to deviate from that, you need to implement `BatchConfigurer`. See The Javadoc of `@EnableBatchProcessing` for more details.

For more about Spring Batch, see the Spring Batch project page.

81.1 Execute Spring Batch Jobs on Startup

Spring Batch auto-configuration is enabled by adding `@EnableBatchProcessing` (from Spring Batch) somewhere in your context.

By default, it executes **all** `Jobs` in the application context on startup (see JobLauncherCommandLineRunner for details). You can narrow down to a specific job or jobs by specifying `spring.batch.job.names` (which takes a comma-separated list of job name patterns).

If the application context includes a `JobRegistry`, the jobs in `spring.batch.job.names` are looked up in the registry instead of being autowired from the context. This is a common pattern with more complex systems, where multiple jobs are defined in child contexts and registered centrally.

See BatchAutoConfiguration and @EnableBatchProcessing for more details.

82. Actuator

Spring Boot includes the Spring Boot Actuator. This section answers questions that often arise from its use.

82.1 Change the HTTP Port or Address of the Actuator Endpoints

In a standalone application, the Actuator HTTP port defaults to the same as the main HTTP port. To make the application listen on a different port, set the external property: `management.server.port`. To listen on a completely different network address (such as when you have an internal network for management and an external one for user applications), you can also set `management.server.address` to a valid IP address to which the server is able to bind.

For more detail, see the `ManagementServerProperties` source code and "Section 50.2, "Customizing the Management Server Port"" in the "Production-ready features" section.

82.2 Customize the 'whitelabel' Error Page

Spring Boot installs a 'whitelabel' error page that you see in a browser client if you encounter a server error (machine clients consuming JSON and other media types should see a sensible response with the right error code).

> **Note**
>
> Set `server.error.whitelabel.enabled=false` to switch the default error page off. Doing so restores the default of the servlet container that you are using. Note that Spring Boot still tries to resolve the error view, so you should probably add your own error page rather than disabling it completely.

Overriding the error page with your own depends on the templating technology that you use. For example, if you use Thymeleaf, you can add an `error.html` template. If you use FreeMarker, you can add an `error.ftl` template. In general, you need a `View` that resolves with a name of `error` or a `@Controller` that handles the `/error` path. Unless you replaced some of the default configuration, you should find a `BeanNameViewResolver` in your `ApplicationContext`, so a `@Bean` named `error` would be a simple way of doing that. See `ErrorMvcAutoConfiguration` for more options.

See also the section on "Error Handling" for details of how to register handlers in the servlet container.

83. Security

This section addresses questions about security when working with Spring Boot, including questions that arise from using Spring Security with Spring Boot.

For more about Spring Security, see the Spring Security project page.

83.1 Switch off the Spring Boot Security Configuration

If you define a `@Configuration` with a `WebSecurityConfigurerAdapter` in your application, it switches off the default webapp security settings in Spring Boot.

83.2 Change the AuthenticationManager and Add User Accounts

If you provide a `@Bean` of type `AuthenticationManager`, `AuthenticationProvider`, or `UserDetailsService`, the default `@Bean` for `InMemoryUserDetailsManager` is not created, so you have the full feature set of Spring Security available (such as various authentication options).

The easiest way to add user accounts is to provide your own `UserDetailsService` bean.

83.3 Enable HTTPS When Running behind a Proxy Server

Ensuring that all your main endpoints are only available over HTTPS is an important chore for any application. If you use Tomcat as a servlet container, then Spring Boot adds Tomcat's own `RemoteIpValve` automatically if it detects some environment settings, and you should be able to rely on the `HttpServletRequest` to report whether it is secure or not (even downstream of a proxy server that handles the real SSL termination). The standard behavior is determined by the presence or absence of certain request headers (`x-forwarded-for` and `x-forwarded-proto`), whose names are conventional, so it should work with most front-end proxies. You can switch on the valve by adding some entries to `application.properties`, as shown in the following example:

```
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.protocol-header=x-forwarded-proto
```

(The presence of either of those properties switches on the valve. Alternatively, you can add the `RemoteIpValve` by adding a `TomcatServletWebServerFactory` bean.)

To configure Spring Security to require a secure channel for all (or some) requests, consider adding your own `WebSecurityConfigurerAdapter` that adds the following `HttpSecurity` configuration:

```
@Configuration
public class SslWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
  // Customize the application security
  http.requiresChannel().anyRequest().requiresSecure();
 }

}
```

84. Hot Swapping

Spring Boot supports hot swapping. This section answers questions about how it works.

84.1 Reload Static Content

There are several options for hot reloading. The recommended approach is to use `spring-boot-devtools`, as it provides additional development-time features, such as support for fast application restarts and LiveReload as well as sensible development-time configuration (such as template caching). Devtools works by monitoring the classpath for changes. This means that static resource changes must be "built" for the change to take affect. By default, this happens automatically in Eclipse when you save your changes. In IntelliJ IDEA, the Make Project command triggers the necessary build. Due to the default restart exclusions, changes to static resources do not trigger a restart of your application. They do, however, trigger a live reload.

Alternatively, running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also allow hot-swapping of Java class changes).

Finally, the Maven and Gradle plugins can be configured (see the `addResources` property) to support running from the command line with reloading of static files directly from source. You can use that with an external css/js compiler process if you are writing that code with higher-level tools.

84.2 Reload Templates without Restarting the Container

Most of the templating technologies supported by Spring Boot include a configuration option to disable caching (described later in this document). If you use the `spring-boot-devtools` module, these properties are automatically configured for you at development time.

Thymeleaf Templates

If you use Thymeleaf, set `spring.thymeleaf.cache` to `false`. See `ThymeleafAutoConfiguration` for other Thymeleaf customization options.

FreeMarker Templates

If you use FreeMarker, set `spring.freemarker.cache` to `false`. See `FreeMarkerAutoConfiguration` for other FreeMarker customization options.

Groovy Templates

If you use Groovy templates, set `spring.groovy.template.cache` to `false`. See `GroovyTemplateAutoConfiguration` for other Groovy customization options.

84.3 Fast Application Restarts

The `spring-boot-devtools` module includes support for automatic application restarts. While not as fast as technologies such as JRebel it is usually significantly faster than a “cold start”. You should probably give it a try before investigating some of the more complex reload options discussed later in this document.

For more details, see the Chapter 20, *Developer Tools* section.

84.4 Reload Java Classes without Restarting the Container

Many modern IDEs (Eclipse, IDEA, and others) support hot swapping of bytecode. Consequently, if you make a change that does not affect class or method signatures, it should reload cleanly with no side effects.

85. Build

Spring Boot includes build plugins for Maven and Gradle. This section answers common questions about these plugins.

85.1 Generate Build Information

Both the Maven plugin and the Gradle plugin allow generating build information containing the coordinates, name, and version of the project. The plugins can also be configured to add additional properties through configuration. When such a file is present, Spring Boot auto-configures a `BuildProperties` bean.

To generate build information with Maven, add an execution for the `build-info` goal, as shown in the following example:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <version>2.0.0.RC1</version>
   <executions>
    <execution>
     <goals>
      <goal>build-info</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

Tip

See the Spring Boot Maven Plugin documentation for more details.

The following example does the same with Gradle:

```
springBoot {
 buildInfo()
}
```

Additional properties can be added by using the DSL, as shown in the following example:

```
springBoot  {
 buildInfo {
  additionalProperties = [
   'acme': 'test'
  ]
 }
}
```

85.2 Generate Git Information

Both Maven and Gradle allow generating a `git.properties` file containing information about the state of your `git` source code repository when the project was built.

For Maven users, the `spring-boot-starter-parent` POM includes a pre-configured plugin to generate a `git.properties` file. To use it, add the following declaration to your POM:

```
<build>
 <plugins>
  <plugin>
   <groupId>pl.project13.maven</groupId>
   <artifactId>git-commit-id-plugin</artifactId>
  </plugin>
 </plugins>
</build>
```

Gradle users can achieve the same result by using the `gradle-git-properties` plugin, as shown in the following example:

```
plugins {
 id "com.gorylenko.gradle-git-properties" version "1.4.17"
}
```

Tip

The commit time in `git.properties` is expected to match the following format: `yyyy-MM-dd’T’HH:mm:ssZ`. This is the default format for both plugins listed above. Using this format lets the time be parsed into a `Date` and its format, when serialized to JSON, to be controlled by Jackson’s date serialization configuration settings.

85.3 Customize Dependency Versions

If you use a Maven build that inherits directly or indirectly from `spring-boot-dependencies` (for instance, `spring-boot-starter-parent`) but you want to override a specific third-party dependency, you can add appropriate `<properties>` elements. Browse the `spring-boot-dependencies` POM for a complete list of properties. For example, to pick a different `slf4j` version, you would add the following property:

```
<properties>
 <slf4j.version>1.7.5<slf4j.version>
</properties>
```

Note

Doing so only works if your Maven project inherits (directly or indirectly) from `spring-boot-dependencies`. If you have added `spring-boot-dependencies` in your own `dependencyManagement` section with `<scope>import</scope>`, you have to redefine the artifact yourself instead of overriding the property.

Warning

Each Spring Boot release is designed and tested against this specific set of third-party dependencies. Overriding versions may cause compatibility issues.

85.4 Create an Executable JAR with Maven

The `spring-boot-maven-plugin` can be used to create an executable “fat” JAR. If you use the `spring-boot-starter-parent` POM, you can declare the plugin and your jars are repackaged as follows:

```
<build>
```

```
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
  </plugin>
 </plugins>
</build>
```

If you do not use the parent POM, you can still use the plugin. However, you must additionally add an `<executions>` section, as follows:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <version>2.0.0.RC1</version>
   <executions>
    <execution>
     <goals>
      <goal>repackage</goal>
     </goals>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

See the plugin documentation for full usage details.

85.5 Use a Spring Boot Application as a Dependency

Like a war file, a Spring Boot application is not intended to be used as a dependency. If your application contains classes that you want to share with other projects, the recommended approach is to move that code into a separate module. The separate module can then be depended upon by your application and other projects.

If you cannot rearrange your code as recommended above, Spring Boot's Maven and Gradle plugins must be configured to produce a separate artifact that is suitable for use as a dependency. The executable archive cannot be used as a dependency as the executable jar format packages application classes in `BOOT-INF/classes`. This means that they cannot be found when the executable jar is used as a dependency.

To produce the two artifacts, one that can be used as a dependency and one that is executable, a classifier must be specified. This classifier is applied to the name of the executable archive, leaving the default archive for use as a dependency.

To configure a classifier of `exec` in Maven, you can use the following configuration:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
    <classifier>exec</classifier>
   </configuration>
  </plugin>
 </plugins>
</build>
```

85.6 Extract Specific Libraries When an Executable Jar Runs

Most nested libraries in an executable jar do not need to be unpacked in order to run. However, certain libraries can have problems. For example, JRuby includes its own nested jar support, which assumes that the `jruby-complete.jar` is always directly available as a file in its own right.

To deal with any problematic libraries, you can flag that specific nested jars should be automatically unpacked to the "temp folder" when the executable jar first runs.

For example, to indicate that JRuby should be flagged for unpacking by using the Maven Plugin, you would add the following configuration:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
   <configuration>
    <requiresUnpack>
     <dependency>
      <groupId>org.jruby</groupId>
      <artifactId>jruby-complete</artifactId>
     </dependency>
    </requiresUnpack>
   </configuration>
  </plugin>
 </plugins>
</build>
```

85.7 Create a Non-executable JAR with Exclusions

Often, if you have an executable and a non-executable jar as two separate build products, the executable version has additional configuration files that are not needed in a library jar. For example, the `application.yml` configuration file might by excluded from the non-executable JAR.

In Maven, the executable jar must be the main artifact and you can add a classified jar for the library, as follows:

```
<build>
 <plugins>
  <plugin>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-maven-plugin</artifactId>
  </plugin>
  <plugin>
   <artifactId>maven-jar-plugin</artifactId>
   <executions>
    <execution>
     <id>lib</id>
     <phase>package</phase>
     <goals>
      <goal>jar</goal>
     </goals>
     <configuration>
      <classifier>lib</classifier>
      <excludes>
       <exclude>application.yml</exclude>
      </excludes>
     </configuration>
    </execution>
   </executions>
  </plugin>
 </plugins>
</build>
```

85.8 Remote Debug a Spring Boot Application Started with Maven

To attach a remote debugger to a Spring Boot application that was started with Maven, you can use the `jvmArguments` property of the maven plugin.

See this example for more details.

85.9 Build an Executable Archive from Ant without Using `spring-boot-antlib`

To build with Ant, you need to grab dependencies, compile, and then create a jar or war archive. To make it executable, you can either use the `spring-boot-antlib` module or you can follow these instructions:

1. If you are building a jar, package the application's classes and resources in a nested `BOOT-INF/classes` directory. If you are building a war, package the application's classes in a nested `WEB-INF/classes` directory as usual.
2. Add the runtime dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib` for a war. Remember **not** to compress the entries in the archive.
3. Add the `provided` (embedded container) dependencies in a nested `BOOT-INF/lib` directory for a jar or `WEB-INF/lib-provided` for a war. Remember **not** to compress the entries in the archive.
4. Add the `spring-boot-loader` classes at the root of the archive (so that the `Main-Class` is available).
5. Use the appropriate launcher (such as `JarLauncher` for a jar file) as a `Main-Class` attribute in the manifest and specify the other properties it needs as manifest entries — principally, by setting a `Start-Class` property.

The following example shows how to build an executable archive with Ant:

```xml
<target name="build" depends="compile">
 <jar destfile="target/${ant.project.name}-${spring-boot.version}.jar" compress="false">
  <mappedresources>
   <fileset dir="target/classes" />
   <globmapper from="*" to="BOOT-INF/classes/*"/>
  </mappedresources>
  <mappedresources>
   <fileset dir="src/main/resources" erroronmissingdir="false"/>
   <globmapper from="*" to="BOOT-INF/classes/*"/>
  </mappedresources>
  <mappedresources>
   <fileset dir="${lib.dir}/runtime" />
   <globmapper from="*" to="BOOT-INF/lib/*"/>
  </mappedresources>
  <zipfileset src="${lib.dir}/loader/spring-boot-loader-jar-${spring-boot.version}.jar" />
  <manifest>
   <attribute name="Main-Class" value="org.springframework.boot.loader.JarLauncher" />
   <attribute name="Start-Class" value="${start-class}" />
  </manifest>
 </jar>
</target>
```

The Ant Sample has a `build.xml` file with a `manual` task that should work if you run it with the following command:

```
$ ant -lib <folder containing ivy-2.2.jar> clean manual
```

Then you can run the application with the following command:

```
$ java -jar target/*.jar
```

86. Traditional Deployment

Spring Boot supports traditional deployment as well as more modern forms of deployment. This section answers common questions about traditional deployment.

86.1 Create a Deployable War File

The first step in producing a deployable war file is to provide a `SpringBootServletInitializer` subclass and override its `configure` method. Doing so makes use of Spring Framework's Servlet 3.0 support and lets you configure your application when it is launched by the servlet container. Typically, you should update your application's main class to extend `SpringBootServletInitializer`, as shown in the following example:

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
  return application.sources(Application.class);
 }

 public static void main(String[] args) throws Exception {
  SpringApplication.run(Application.class, args);
 }

}
```

The next step is to update your build configuration such that your project produces a war file rather than a jar file. If you use Maven and `spring-boot-starter-parent` (which configures Maven's war plugin for you), all you need to do is to modify `pom.xml` to change the packaging to war, as follows:

```
<packaging>war</packaging>
```

If you use Gradle, you need to modify `build.gradle` to apply the war plugin to the project, as follows:

```
apply plugin: 'war'
```

The final step in the process is to ensure that the embedded servlet container does not interfere with the servlet container to which the war file is deployed. To do so, you need to mark the embedded servlet container dependency as being provided.

If you use Maven, the following example marks the servlet container (Tomcat, in this case) as being provided:

```
<dependencies>
 <!-- … -->
 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <scope>provided</scope>
 </dependency>
 <!-- … -->
</dependencies>
```

If you use Gradle, the following example marks the servlet container (Tomcat, in this case) as being provided:

```
dependencies {
 // …
 providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
 // …
}
```

Tip

`providedRuntime` is preferred to Gradle’s `compileOnly` configuration. Among other limitations, `compileOnly` dependencies are not on the test classpath, so any web-based integration tests fail.

If you use the Spring Boot build tools, marking the embedded servlet container dependency as provided produces an executable war file with the provided dependencies packaged in a `lib-provided` directory. This means that, in addition to being deployable to a servlet container, you can also run your application by using `java -jar` on the command line.

Tip

Take a look at Spring Boot’s sample applications for a Maven-based example of the previously described configuration.

86.2 Create a Deployable War File for Older Servlet Containers

Older Servlet containers do not have support for the `ServletContextInitializer` bootstrap process used in Servlet 3.0. You can still use Spring and Spring Boot in these containers, but you are going to need to add a `web.xml` to your application and configure it to load an `ApplicationContext` via a `DispatcherServlet`.

86.3 Convert an Existing Application to Spring Boot

For a non-web application, it should be easy to convert an existing Spring application to a Spring Boot application. To do so, throw away the code that creates your `ApplicationContext` and replace it with calls to `SpringApplication` or `SpringApplicationBuilder`. Spring MVC web applications are generally amenable to first creating a deployable war application and then migrating it later to an executable war or jar. See the Getting Started Guide on Converting a jar to a war.

To create a deployable war by extending `SpringBootServletInitializer` (for example, in a class called `Application`) and adding the Spring Boot `@SpringBootApplication` annotation, use code similar to that shown in the following example:

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
  // Customize the application or call application.sources(...) to add sources
  // Since our example is itself a @Configuration class (via @SpringBootApplication)
  // we actually don't need to override this method.
  return application;
 }

}
```

Remember that, whatever you put in the `sources` is merely a Spring `ApplicationContext`. Normally, anything that already works should work here. There might be some beans you can remove later and let Spring Boot provide its own defaults for them, but it should be possible to get something working before you need to do that.

Static resources can be moved to `/public` (or `/static` or `/resources` or `/META-INF/resources`) in the classpath root. The same applies to `messages.properties` (which Spring Boot automatically detects in the root of the classpath).

Vanilla usage of Spring `DispatcherServlet` and Spring Security should require no further changes. If you have other features in your application (for instance, using other servlets or filters), you may need to add some configuration to your `Application` context, by replacing those elements from the `web.xml`, as follows:

- A `@Bean` of type `Servlet` or `ServletRegistrationBean` installs that bean in the container as if it were a `<servlet/>` and `<servlet-mapping/>` in `web.xml`.
- A `@Bean` of type `Filter` or `FilterRegistrationBean` behaves similarly (as a `<filter/>` and `<filter-mapping/>`).
- An `ApplicationContext` in an XML file can be added through an `@ImportResource` in your `Application`. Alternatively, simple cases where annotation configuration is heavily used already can be recreated in a few lines as `@Bean` definitions.

Once the war file is working, you can make it executable by adding a `main` method to your `Application`, as shown in the following example:

```
public static void main(String[] args) {
 SpringApplication.run(Application.class, args);
}
```

Note

If you intend to start your application as a war or as an executable application, you need to share the customizations of the builder in a method that is both available to the `SpringBootServletInitializer` callback and in the `main` method in a class similar to the following:

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {

 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
  return configureApplication(builder);
 }

 public static void main(String[] args) {
  configureApplication(new SpringApplicationBuilder()).run(args);
 }

 private static SpringApplicationBuilder configureApplication(SpringApplicationBuilder builder) {
  return builder.sources(Application.class).bannerMode(Banner.Mode.OFF);
 }

}
```

Applications can fall into more than one category:

- Servlet 3.0+ applications with no `web.xml`.
- Applications with a `web.xml`.
- Applications with a context hierarchy.

- Applications without a context hierarchy.

All of these should be amenable to translation, but each might require slightly different techniques.

Servlet 3.0+ applications might translate pretty easily if they already use the Spring Servlet 3.0+ initializer support classes. Normally, all the code from an existing `WebApplicationInitializer` can be moved into a `SpringBootServletInitializer`. If your existing application has more than one `ApplicationContext` (for example, if it uses `AbstractDispatcherServletInitializer`) then you might be able to combine all your context sources into a single `SpringApplication`. The main complication you might encounter is if combining does not work and you need to maintain the context hierarchy. See the entry on building a hierarchy for examples. An existing parent context that contains web-specific features usually needs to be broken up so that all the `ServletContextAware` components are in the child context.

Applications that are not already Spring applications might be convertible to Spring Boot applications, and the previously mentioned guidance may help. However, you may yet encounter problems. In that case, we suggest asking questions on Stack Overflow with a tag of `spring-boot`.

86.4 Deploying a WAR to WebLogic

To deploy a Spring Boot application to WebLogic, you must ensure that your servlet initializer **directly** implements `WebApplicationInitializer` (even if you extend from a base class that already implements it).

A typical initializer for WebLogic should resemble the following example:

```java
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.WebApplicationInitializer;

@SpringBootApplication
public class MyApplication extends SpringBootServletInitializer implements WebApplicationInitializer {

}
```

If you use Logback, you also need to tell WebLogic to prefer the packaged version rather than the version that was pre-installed with the server. You can do so by adding a `WEB-INF/weblogic.xml` file with the following contents:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wls:weblogic-web-app
 xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd
  http://xmlns.oracle.com/weblogic/weblogic-web-app
  http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">
 <wls:container-descriptor>
  <wls:prefer-application-packages>
   <wls:package-name>org.slf4j</wls:package-name>
  </wls:prefer-application-packages>
 </wls:container-descriptor>
</wls:weblogic-web-app>
```

86.5 Deploying a WAR in an Old (Servlet 2.5) Container

Spring Boot uses Servlet 3.0 APIs to initialize the `ServletContext` (register `Servlets` and so on), so you cannot use the same application in a Servlet 2.5 container. It **is**, however, possible to run a Spring Boot application on an older container with some special tools. If you include `org.springframework.boot:spring-boot-legacy` as a dependency (maintained separately to the core of Spring Boot and currently available at 1.1.0.RELEASE), all you need to do is create a `web.xml` and declare a context listener to create the application context and your filters and servlets. The context listener is a special purpose one for Spring Boot, but the rest of it is normal for a Spring application in Servlet 2.5. The following Maven example shows how to set up a Spring Boot project to run in a Servlet 2.5 container:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">

 <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>demo.Application</param-value>
 </context-param>

 <listener>
  <listener-class>org.springframework.boot.legacy.context.web.SpringBootContextLoaderListener</listener-
class>
 </listener>

 <filter>
  <filter-name>metricsFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
 </filter>

 <filter-mapping>
  <filter-name>metricsFilter</filter-name>
  <url-pattern>/*</url-pattern>
 </filter-mapping>

 <servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
   <param-name>contextAttribute</param-name>
   <param-value>org.springframework.web.context.WebApplicationContext.ROOT</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
 </servlet>

 <servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/</url-pattern>
 </servlet-mapping>

</web-app>
```

In the preceding example, we use a single application context (the one created by the context listener) and attach it to the `DispatcherServlet` by using an `init` parameter. This is normal in a Spring Boot application (you normally only have one application context).

86.6 Use Jedis Instead of Lettuce

By default, the Spring Boot starter (`spring-boot-starter-data-redis`) uses Lettuce. You need to exclude that dependency and include the Jedis one instead. Spring Boot manages these dependencies to help make this process as easy as possible.

The following example shows how to do so in Maven:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
 <exclusions>
  <exclusion>
   <groupId>io.lettuce</groupId>
   <artifactId>lettuce-core</artifactId>
  </exclusion>
 </exclusions>
</dependency>
<dependency>
 <groupId>redis.clients</groupId>
 <artifactId>jedis</artifactId>
</dependency>
```

The following example shows how to do so in Gradle:

```
configurations {
 compile.exclude module: "lettuce"
}

dependencies {
 compile("redis.clients:jedis")
 // ...
}
```

Part X. Appendices

Appendix A. Common application properties

Various properties can be specified inside your `application.properties` file, inside your `application.yml` file, or as command line switches. This appendix provides a list of common Spring Boot properties and references to the underlying classes that consume them.

Note

Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

Warning

This sample file is meant as a guide only. Do **not** copy and paste the entire content into your application. Rather, pick only the properties that you need.

```
# ===================================================================
# COMMON SPRING BOOT PROPERTIES
#
# This sample file is provided as a guideline. Do NOT copy it in its
# entirety to your own application.      ^^^
# ===================================================================


# ----------------------------------------
# CORE PROPERTIES
# ----------------------------------------
debug=false # Enable debug logs.
trace=false # Enable trace logs.

# LOGGING
logging.config= # Location of the logging configuration file. For instance, `classpath:logback.xml` for
 Logback
logging.exception-conversion-word=%wEx # Conversion word used when logging exceptions.
logging.file= # Log file name. For instance, `myapp.log`
logging.file.max-history= # Maximum of archive log files to keep. Only supported with the default
 logback setup.
logging.file.max-size= # Maximum log file size. Only supported with the default logback setup.
logging.level.*= # Log levels severity mapping. For instance, `logging.level.org.springframework=DEBUG`
logging.path= # Location of the log file. For instance, `/var/log`.
logging.pattern.console= # Appender pattern for output to the console. Supported only with the default
 Logback setup.
logging.pattern.dateformat=yyyy-MM-dd HH:mm:ss.SSS # Appender pattern for log date format. Supported
 only with the default Logback setup.
logging.pattern.file= # Appender pattern for output to a file. Supported only with the default Logback
 setup.
logging.pattern.level= # Appender pattern for log level (default: %5p). Supported only with the default
 Logback setup.
logging.register-shutdown-hook=false # Register a shutdown hook for the logging system when it is
 initialized.

# AOP
spring.aop.auto=true # Add @EnableAspectJAutoProxy.
spring.aop.proxy-target-class=true # Whether subclass-based (CGLIB) proxies are to be created (true), as
 opposed to standard Java interface-based proxies (false).

# IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name= # Application name.

# ADMIN (SpringApplicationAdminJmxAutoConfiguration)
```

```
spring.application.admin.enabled=false # Whether to enable admin features for the application.
spring.application.admin.jmx-name=org.springframework.boot:type=Admin,name=SpringApplication # JMX name
 of the application admin MBean.

# AUTO-CONFIGURATION
spring.autoconfigure.exclude= # Auto-configuration classes to exclude.

# BANNER
spring.banner.charset=UTF-8 # Banner file encoding.
spring.banner.location=classpath:banner.txt # Banner file location.
spring.banner.image.location=classpath:banner.gif # Banner image file location (jpg or png can also be
 used).
spring.banner.image.width= # Width of the banner image in chars (default 76)
spring.banner.image.height= # Height of the banner image in chars (default based on image height)
spring.banner.image.margin= # Left hand image margin in chars (default 2)
spring.banner.image.invert= # Whether images should be inverted for dark terminal themes (default false)

# SPRING CORE
spring.beaninfo.ignore=true # Whether to skip search of BeanInfo classes.

# SPRING CACHE (CacheProperties)
spring.cache.cache-names= # Comma-separated list of cache names to create if supported by the underlying
 cache manager.
spring.cache.caffeine.spec= # The spec to use to create caches. See CaffeineSpec for more details on the
 spec format.
spring.cache.couchbase.expiration=0ms # Entry expiration in milliseconds. By default, the entries never
 expire.
spring.cache.ehcache.config= # The location of the configuration file to use to initialize EhCache.
spring.cache.infinispan.config= # The location of the configuration file to use to initialize
 Infinispan.
spring.cache.jcache.config= # The location of the configuration file to use to initialize the cache
 manager.
spring.cache.jcache.provider= # Fully qualified name of the CachingProvider implementation to use to
 retrieve the JSR-107 compliant cache manager. Needed only if more than one JSR-107 implementation is
 available on the classpath.
spring.cache.redis.cache-null-values=true # Allow caching null values.
spring.cache.redis.key-prefix= # Key prefix.
spring.cache.redis.time-to-live=0ms # Entry expiration. By default the entries never expire.
spring.cache.redis.use-key-prefix=true # Whether to use the key prefix when writing to Redis.
spring.cache.type= # Cache type. By default, auto-detected according to the environment.

# SPRING CONFIG - using environment property only (ConfigFileApplicationListener)
spring.config.additional-location= # Config file locations used in addition to the defaults.
spring.config.location= # Config file locations.
spring.config.name=application # Config file name.

# HAZELCAST (HazelcastProperties)
spring.hazelcast.config= # The location of the configuration file to use to initialize Hazelcast.

# PROJECT INFORMATION (ProjectInfoProperties)
spring.info.build.location=classpath:META-INF/build-info.properties # Location of the generated build-
info.properties file.
spring.info.git.location=classpath:git.properties # Location of the generated git.properties file.

# JMX
spring.jmx.default-domain= # JMX domain name.
spring.jmx.enabled=true # Expose management beans to the JMX domain.
spring.jmx.server=mbeanServer # MBeanServer bean name.

# Email (MailProperties)
spring.mail.default-encoding=UTF-8 # Default MimeMessage encoding.
spring.mail.host= # SMTP server host. For instance, `smtp.example.com`
spring.mail.jndi-name= # Session JNDI name. When set, takes precedence over other mail settings.
spring.mail.password= # Login password of the SMTP server.
spring.mail.port= # SMTP server port.
spring.mail.properties.*= # Additional JavaMail session properties.
spring.mail.protocol=smtp # Protocol used by the SMTP server.
spring.mail.test-connection=false # Whether to test that the mail server is available on startup.
spring.mail.username= # Login user of the SMTP server.

# APPLICATION SETTINGS (SpringApplication)
```

```
spring.main.banner-mode=console # Mode used to display the banner when the application runs.
spring.main.sources= # Sources (class names, package names, or XML resource locations) to include in the
 ApplicationContext.
spring.main.web-application-type= # Flag to explicitly request a specific type of web application. If
 not set, auto-detected based on the classpath.

# FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding= # Expected character encoding the application must use.

# INTERNATIONALIZATION (MessageSourceProperties)
spring.messages.always-use-message-format=false # Whether to always apply the MessageFormat rules,
 parsing even messages without arguments.
spring.messages.basename=messages # Comma-separated list of basenames (essentially a fully-qualified
 classpath location), each following the ResourceBundle convention with relaxed support for slash based
 locations.
spring.messages.cache-duration=-1 # Loaded resource bundle files cache duration. When not set, bundles
 are cached forever.
spring.messages.encoding=UTF-8 # Message bundles encoding.
spring.messages.fallback-to-system-locale=true # Whether to fall back to the system Locale if no files
 for a specific Locale have been found.
spring.messages.use-code-as-default-message=false # Whether to use the message code as the default
 message instead of throwing a "NoSuchMessageException". Recommended during development only.

# OUTPUT
spring.output.ansi.enabled=detect # Configures the ANSI output.

# PID FILE (ApplicationPidFileWriter)
spring.pid.fail-on-write-error= # Fails if ApplicationPidFileWriter is used but it cannot write the PID
 file.
spring.pid.file= # Location of the PID file to write (if ApplicationPidFileWriter is used).

# PROFILES
spring.profiles.active= # Comma-separated list (or list if using YAML) of active profiles.
spring.profiles.include= # Unconditionally activate the specified comma-separated list of profiles (or
 list of profiles if using YAML).

# QUARTZ SCHEDULER (QuartzProperties)
spring.quartz.jdbc.initialize-schema=embedded # Database schema initialization mode.
spring.quartz.jdbc.schema=classpath:org/quartz/impl/jdbcjobstore/tables_@@platform@@.sql # Path to the
 SQL file to use to initialize the database schema.
spring.quartz.job-store-type=memory # Quartz job store type.
spring.quartz.properties.*= # Additional Quartz Scheduler properties.

# REACTOR (ReactorCoreProperties)
spring.reactor.stacktrace-mode.enabled=false # Whether Reactor should collect stacktrace information at
 runtime.

# SENDGRID (SendGridAutoConfiguration)
spring.sendgrid.api-key= # SendGrid API key.
spring.sendgrid.proxy.host= # SendGrid proxy host.
spring.sendgrid.proxy.port= # SendGrid proxy port.


# ----------------------------------------
# WEB PROPERTIES
# ----------------------------------------

# EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.address= # Network address to which the server should bind.
server.compression.enabled=false # Whether response compression is enabled.
server.compression.excluded-user-agents= # List of user-agents to exclude from compression.
server.compression.mime-types=text/html,text/xml,text/plain,text/css,text/javascript,application/
javascript # Comma-separated list of MIME types that should be compressed.
server.compression.min-response-size=2048 # Minimum response size that is required for compression to be
 performed.
server.connection-timeout= # Time that connectors wait for another HTTP request before closing the
 connection. When not set, the connector's container-specific default is used. Use a value of -1 to
 indicate no (that is, an infinite) timeout.
server.display-name=application # Display name of the application.
server.error.include-exception=false # Include the "exception" attribute.
server.error.include-stacktrace=never # When to include a "stacktrace" attribute.
server.error.path=/error # Path of the error controller.
server.error.whitelabel.enabled=true # Enable the default error page displayed in browsers in case of a
 server error.
server.http2.enabled=false # Whether to enable HTTP/2 support, if the current environment supports it.
server.jetty.acceptors= # Number of acceptor threads to use.
server.jetty.accesslog.append=false # Append to log.
server.jetty.accesslog.date-format=dd/MMM/yyyy:HH:mm:ss Z # Timestamp format of the request log.
server.jetty.accesslog.enabled=false # Enable access log.
server.jetty.accesslog.extended-format=false # Enable extended NCSA format.
server.jetty.accesslog.file-date-format= # Date format to place in log file name.
server.jetty.accesslog.filename= # Log filename. If not specified, logs redirect to "System.err".
server.jetty.accesslog.locale= # Locale of the request log.
server.jetty.accesslog.log-cookies=false # Enable logging of the request cookies.
server.jetty.accesslog.log-latency=false # Enable logging of request processing time.
server.jetty.accesslog.log-server=false # Enable logging of the request hostname.
server.jetty.accesslog.retention-period=31 # Number of days before rotated log files are deleted.
server.jetty.accesslog.time-zone=GMT # Timezone of the request log.
server.jetty.max-http-post-size=0 # Maximum size, in bytes, of the HTTP post or put content.
server.jetty.selectors= # Number of selector threads to use.
server.max-http-header-size=0 # Maximum size, in bytes, of the HTTP message header.
server.port=8080 # Server HTTP port.
server.server-header= # Value to use for the Server response header (if empty, no header is sent)
server.use-forward-headers= # Whether X-Forwarded-* headers should be applied to the HttpRequest.
server.servlet.context-parameters.*= # Servlet context init parameters
server.servlet.context-path= # Context path of the application.
server.servlet.jsp.class-name=org.apache.jasper.servlet.JspServlet # The class name of the JSP servlet.
server.servlet.jsp.init-parameters.*= # Init parameters used to configure the JSP servlet.
server.servlet.jsp.registered=true # Whether the JSP servlet is registered.
server.servlet.path=/ # Path of the main dispatcher servlet.
server.servlet.session.cookie.comment= # Comment for the session cookie.
server.servlet.session.cookie.domain= # Domain for the session cookie.
server.servlet.session.cookie.http-only= # "HttpOnly" flag for the session cookie.
server.servlet.session.cookie.max-age= # Maximum age of the session cookie. If a duration suffix is not
 specified, seconds will be used.
server.servlet.session.cookie.name= # Session cookie name.
server.servlet.session.cookie.path= # Path of the session cookie.
server.servlet.session.cookie.secure= # "Secure" flag for the session cookie.
server.servlet.session.persistent=false # Whether to persist session data between restarts.
server.servlet.session.store-dir= # Directory used to store session data.
server.servlet.session.timeout= # Session timeout. If a duration suffix is not specified, seconds will
 be used.
server.servlet.session.tracking-modes= # Session tracking modes (one or more of the following: "cookie",
 "url", "ssl").
server.ssl.ciphers= # Supported SSL ciphers.
server.ssl.client-auth= # Whether client authentication is wanted ("want") or needed ("need"). Requires
 a trust store.
server.ssl.enabled= # Enable SSL support.
server.ssl.enabled-protocols= # Enabled SSL protocols.
server.ssl.key-alias= # Alias that identifies the key in the key store.
server.ssl.key-password= # Password used to access the key in the key store.
server.ssl.key-store= # Path to the key store that holds the SSL certificate (typically a jks file).
server.ssl.key-store-password= # Password used to access the key store.
server.ssl.key-store-provider= # Provider for the key store.
server.ssl.key-store-type= # Type of the key store.
server.ssl.protocol=TLS # SSL protocol to use.
server.ssl.trust-store= # Trust store that holds SSL certificates.
server.ssl.trust-store-password= # Password used to access the trust store.
server.ssl.trust-store-provider= # Provider for the trust store.
server.ssl.trust-store-type= # Type of the trust store.
server.tomcat.accept-count= # Maximum queue length for incoming connection requests when all possible
 request processing threads are in use.
server.tomcat.accesslog.buffered=true # Whether to buffer output such that it is flushed only
 periodically.
server.tomcat.accesslog.directory=logs # Directory in which log files are created. Can be absolute or
 relative to the Tomcat base dir.
server.tomcat.accesslog.enabled=false # Enable access log.
server.tomcat.accesslog.file-date-format=.yyyy-MM-dd # Date format to place in the log file name.
server.tomcat.accesslog.pattern=common # Format pattern for access logs.
server.tomcat.accesslog.prefix=access_log # Log file name prefix.
server.tomcat.accesslog.rename-on-rotate=false # Whether to defer inclusion of the date stamp in the
 file name until rotate time.
```

```
server.tomcat.accesslog.request-attributes-enabled=false # Set request attributes for the IP address,
 Hostname, protocol, and port used for the request.
server.tomcat.accesslog.rotate=true # Whether to enable access log rotation.
server.tomcat.accesslog.suffix=.log # Log file name suffix.
server.tomcat.additional-tld-skip-patterns= # Comma-separated list of additional patterns that match
 jars to ignore for TLD scanning.
server.tomcat.background-processor-delay=30s # Delay between the invocation of backgroundProcess
 methods. If a duration suffix is not specified, seconds will be used.
server.tomcat.basedir= # Tomcat base directory. If not specified, a temporary directory is used.
server.tomcat.internal-proxies=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
  192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\
  169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\
  127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\
  172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
  172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\
  172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3} # regular expression matching trusted IP addresses.
server.tomcat.max-connections= # Maximum number of connections that the server accepts and processes at
 any given time.
server.tomcat.max-http-header-size=0 # Maximum size, in bytes, of the HTTP message header.
server.tomcat.max-http-post-size=0 # Maximum size, in bytes, of the HTTP post content.
server.tomcat.max-threads=0 # Maximum number of worker threads.
server.tomcat.min-spare-threads=0 # Minimum number of worker threads.
server.tomcat.port-header=X-Forwarded-Port # Name of the HTTP header used to override the original port
 value.
server.tomcat.protocol-header= # Header that holds the incoming protocol, usually named "X-Forwarded-
Proto".
server.tomcat.protocol-header-https-value=https # Value of the protocol header indicating whether the
 incoming request uses SSL.
server.tomcat.redirect-context-root= # Whether requests to the context root should be redirected by
 appending a / to the path.
server.tomcat.remote-ip-header= # Name of the HTTP header from which the remote IP is extracted. For
 instance, `X-FORWARDED-FOR`.
server.tomcat.resource.cache-ttl= # Time-to-live of the static resource cache.
server.tomcat.uri-encoding=UTF-8 # Character encoding to use to decode the URI.
server.tomcat.use-relative-redirects= # Whether HTTP 1.1 and later location headers generated by a call
 to sendRedirect will use relative or absolute redirects.
server.undertow.accesslog.dir= # Undertow access log directory.
server.undertow.accesslog.enabled=false # Whether to enable the access log.
server.undertow.accesslog.pattern=common # Format pattern for access logs.
server.undertow.accesslog.prefix=access_log. # Log file name prefix.
server.undertow.accesslog.rotate=true # Whether to enable access log rotation.
server.undertow.accesslog.suffix=log # Log file name suffix.
server.undertow.buffer-size= # Size of each buffer, in bytes.
server.undertow.direct-buffers= # Whether to allocate buffers outside the Java heap.
server.undertow.io-threads= # Number of I/O threads to create for the worker.
server.undertow.eager-filter-init=true # Whether servlet filters should be initialized on startup.
server.undertow.max-http-post-size=0 # Maximum size, in bytes, of the HTTP post content.
server.undertow.worker-threads= # Number of worker threads.

# FREEMARKER (FreeMarkerProperties)
spring.freemarker.allow-request-override=false # Whether HttpServletRequest attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.freemarker.allow-session-override=false # Whether HttpSession attributes are allowed to override
 (hide) controller generated model attributes of the same name.
spring.freemarker.cache=false # Whether to enable template caching.
spring.freemarker.charset=UTF-8 # Template encoding.
spring.freemarker.check-template-location=true # Whether to check that the templates location exists.
spring.freemarker.content-type=text/html # Content-Type value.
spring.freemarker.enabled=true # Whether to enable MVC view resolution for this technology.
spring.freemarker.expose-request-attributes=false # Whether all request attributes should be added to
 the model prior to merging with the template.
spring.freemarker.expose-session-attributes=false # Whether all HttpSession attributes should be added
 to the model prior to merging with the template.
spring.freemarker.expose-spring-macro-helpers=true # Whether to expose a RequestContext for use by
 Spring's macro library, under the name "springMacroRequestContext".
spring.freemarker.prefer-file-system-access=true # Whether to prefer file system access for template
 loading. File system access enables hot detection of template changes.
spring.freemarker.prefix= # Prefix that gets prepended to view names when building a URL.
spring.freemarker.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.freemarker.settings.*= # Well-known FreeMarker keys which are passed to FreeMarker's
 Configuration.
```

```
spring.freemarker.suffix=.ftl # Suffix that gets appended to view names when building a URL.
spring.freemarker.template-loader-path=classpath:/templates/ # Comma-separated list of template paths.
spring.freemarker.view-names= # White list of view names that can be resolved.

# GROOVY TEMPLATES (GroovyTemplateProperties)
spring.groovy.template.allow-request-override=false # Whether HttpServletRequest attributes are allowed
 to override (hide) controller generated model attributes of the same name.
spring.groovy.template.allow-session-override=false # Whether HttpSession attributes are allowed to
 override (hide) controller generated model attributes of the same name.
spring.groovy.template.cache= # Whether to enable template caching.
spring.groovy.template.charset=UTF-8 # Template encoding.
spring.groovy.template.check-template-location=true # Check that the templates location exists.
spring.groovy.template.configuration.*= # See GroovyMarkupConfigurer
spring.groovy.template.content-type=test/html # Content-Type value.
spring.groovy.template.enabled=true # Whether to enable MVC view resolution for this technology.
spring.groovy.template.expose-request-attributes=false # Whether all request attributes should be added
 to the model prior to merging with the template.
spring.groovy.template.expose-session-attributes=false # Whether all HttpSession attributes should be
 added to the model prior to merging with the template.
spring.groovy.template.expose-spring-macro-helpers=true # Whether to expose a RequestContext for use by
 Spring's macro library, under the name "springMacroRequestContext".
spring.groovy.template.prefix= # Prefix that gets prepended to view names when building a URL.
spring.groovy.template.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.groovy.template.resource-loader-path=classpath:/templates/ # Template path.
spring.groovy.template.suffix=.tpl # Suffix that gets appended to view names when building a URL.
spring.groovy.template.view-names= # White list of view names that can be resolved.

# SPRING HATEOAS (HateoasProperties)
spring.hateoas.use-hal-as-default-json-media-type=true # Whether application/hal+json responses should
 be sent to requests that accept application/json.

# HTTP message conversion
spring.http.converters.preferred-json-mapper= # Preferred JSON mapper to use for HTTP message
 conversion. By default, auto-detected according to the environment.

# HTTP encoding (HttpEncodingProperties)
spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses. Added to the "Content-Type"
 header if not set explicitly.
spring.http.encoding.enabled=true # Whether to enable http encoding support.
spring.http.encoding.force= # Whether to force the encoding to the configured charset on HTTP requests
 and responses.
spring.http.encoding.force-request= # Whether to force the encoding to the configured charset on HTTP
 requests. Defaults to true when "force" has not been specified.
spring.http.encoding.force-response= # Whether to force the encoding to the configured charset on HTTP
 responses.
spring.http.encoding.mapping= # Locale in which to encode mapping.

# MULTIPART (MultipartProperties)
spring.servlet.multipart.enabled=true # Whether to enable support of multipart uploads.
spring.servlet.multipart.file-size-threshold=0 # Threshold after which files are written to disk. Values
 can use the suffixes "MB" or "KB" to indicate megabytes or kilobytes, respectively.
spring.servlet.multipart.location= # Intermediate location of uploaded files.
spring.servlet.multipart.max-file-size=1MB # Max file size. Values can use the suffixes "MB" or "KB" to
 indicate megabytes or kilobytes, respectively.
spring.servlet.multipart.max-request-size=10MB # Max request size. Values can use the suffixes "MB" or
 "KB" to indicate megabytes or kilobytes, respectively.
spring.servlet.multipart.resolve-lazily=false # Whether to resolve the multipart request lazily at the
 time of file or parameter access.

# JACKSON (JacksonProperties)
spring.jackson.date-format= # Date format string or a fully-qualified date format class name. For
 instance, `yyyy-MM-dd HH:mm:ss`.
spring.jackson.default-property-inclusion= # Controls the inclusion of properties during serialization.
 Configured with one of the values in Jackson's JsonInclude.Include enumeration.
spring.jackson.deserialization.*= # Jackson on/off features that affect the way Java objects are
 deserialized.
spring.jackson.generator.*= # Jackson on/off features for generators.
spring.jackson.joda-date-time-format= # Joda date time format string. If not configured, "date-format"
 is used as a fallback if it is configured with a format string.
spring.jackson.locale= # Locale used for formatting.
spring.jackson.mapper.*= # Jackson general purpose on/off features.
spring.jackson.parser.*= # Jackson on/off features for parsers.
spring.jackson.property-naming-strategy= # One of the constants on Jackson's PropertyNamingStrategy. Can
 also be a fully-qualified class name of a PropertyNamingStrategy subclass.
spring.jackson.serialization.*= # Jackson on/off features that affect the way Java objects are
 serialized.
spring.jackson.time-zone= #  Time zone used when formatting dates. For instance, "America/Los_Angeles"
 or "GMT+10".

# GSON (GsonProperties)
spring.gson.date-format= # Format to use when serializing Date objects.
spring.gson.disable-html-escaping= # Whether to disable the escaping of HTML characters such as '<',
 '>', etc.
spring.gson.disable-inner-class-serialization= # Whether to exclude inner classes during serialization.
spring.gson.enable-complex-map-key-serialization= # Whether to enable serialization of complex map keys
 (i.e. non-primitives).
spring.gson.exclude-fields-without-expose-annotation= # Whether to exclude all fields from consideration
 for serialization or deserialization that do not have the "Expose" annotation.
spring.gson.field-naming-policy= # Naming policy that should be applied to an object's field during
 serialization and deserialization.
spring.gson.generate-non-executable-json= # Whether to generate non executable JSON by prefixing the
 output with some special text.
spring.gson.lenient= # Whether to be lenient about parsing JSON that doesn't conform to RFC 4627.
spring.gson.long-serialization-policy= # Serialization policy for Long and long types.
spring.gson.pretty-printing= # Whether to output serialized JSON that fits in a page for pretty
 printing.
spring.gson.serialize-nulls= # Whether to serialize null fields.

# JERSEY (JerseyProperties)
spring.jersey.application-path= # Path that serves as the base URI for the application. If specified,
 overrides the value of "@ApplicationPath".
spring.jersey.filter.order=0 # Jersey filter chain order.
spring.jersey.init.*= # Init parameters to pass to Jersey through the servlet or filter.
spring.jersey.servlet.load-on-startup=-1 # Load on startup priority of the Jersey servlet.
spring.jersey.type=servlet # Jersey integration type.

# SPRING LDAP (LdapProperties)
spring.ldap.anonymous-read-only=false # Whether read-only operations should use an anonymous
 environment.
spring.ldap.base= # Base suffix from which all operations should originate.
spring.ldap.base-environment.*= # LDAP specification settings.
spring.ldap.password= # Login password of the server.
spring.ldap.urls= # LDAP URLs of the server.
spring.ldap.username= # Login username of the server.

# EMBEDDED LDAP (EmbeddedLdapProperties)
spring.ldap.embedded.base-dn= # The base DN
spring.ldap.embedded.credential.username= # Embedded LDAP username.
spring.ldap.embedded.credential.password= # Embedded LDAP password.
spring.ldap.embedded.ldif=classpath:schema.ldif # Schema (LDIF) script resource reference.
spring.ldap.embedded.port= # Embedded LDAP port.
spring.ldap.embedded.validation.enabled=true # Whether to enable LDAP schema validation.
spring.ldap.embedded.validation.schema= # Path to the custom schema.

# MUSTACHE TEMPLATES (MustacheAutoConfiguration)
spring.mustache.allow-request-override= # Whether HttpServletRequest attributes are allowed to override
 (hide) controller generated model attributes of the same name.
spring.mustache.allow-session-override= # Whether HttpSession attributes are allowed to override (hide)
 controller generated model attributes of the same name.
spring.mustache.cache= # Whether to enable template caching.
spring.mustache.charset= # Template encoding.
spring.mustache.check-template-location= # Whether to check that the templates location exists.
spring.mustache.content-type= # Content-Type value.
spring.mustache.enabled= # Whether to enable MVC view resolution for this technology.
spring.mustache.expose-request-attributes= # Whether all request attributes should be added to the model
 prior to merging with the template.
spring.mustache.expose-session-attributes= # Whether all HttpSession attributes should be added to the
 model prior to merging with the template.
spring.mustache.expose-spring-macro-helpers= # Whether to expose a RequestContext for use by Spring's
 macro library under the name "springMacroRequestContext".
spring.mustache.prefix=classpath:/templates/ # Prefix to apply to template names.
spring.mustache.request-context-attribute= # Name of the RequestContext attribute for all views.
spring.mustache.suffix=.mustache # Suffix to apply to template names.
spring.mustache.view-names= # White list of view names that can be resolved.

# SPRING MVC (WebMvcProperties)
spring.mvc.async.request-timeout= # Amount of time before asynchronous request handling times out.
spring.mvc.content-negotiation.favor-parameter=false # Whether a request parameter ("format" by default)
 should be used to determine the requested media type.
spring.mvc.content-negotiation.favor-path-extension=false # Whether the path extension in the URL path
 should be used to determine the requested media type.
spring.mvc.content-negotiation.media-types.*= # Maps file extensions to media types for content
 negotiation.
spring.mvc.content-negotiation.parameter-name= # Query parameter name to use when "favor-parameter" is
 enabled.
spring.mvc.date-format= # Date format to use. For instance, `dd/MM/yyyy`.
spring.mvc.dispatch-trace-request=false # Whether to dispatch TRACE requests to the FrameworkServlet
 doService method.
spring.mvc.dispatch-options-request=true # Whether to dispatch OPTIONS requests to the FrameworkServlet
 doService method.
spring.mvc.favicon.enabled=true # Whether to enable resolution of favicon.ico.
spring.mvc.formcontent.putfilter.enabled=true # Whether to enable Spring's HttpPutFormContentFilter.
spring.mvc.ignore-default-model-on-redirect=true # Whether the content of the "default" model should be
 ignored during redirect scenarios.
spring.mvc.locale= # Locale to use. By default, this locale is overridden by the "Accept-Language"
 header.
spring.mvc.locale-resolver=accept-header # Define how the locale should be resolved.
spring.mvc.log-resolved-exception=false # Whether to enable warn logging of exceptions resolved by a
 "HandlerExceptionResolver".
spring.mvc.message-codes-resolver-format= # Formatting strategy for message codes. For instance,
 `PREFIX_ERROR_CODE`.
spring.mvc.path-match.use-registered-suffix-pattern=false # Whether suffix pattern matching should work
 only against extensions registered with "spring.mvc.content-negotiation.media-types.*".
spring.mvc.path-match.use-suffix-pattern=false # Whether to use suffix pattern match (".*") when
 matching patterns to requests.
spring.mvc.servlet.load-on-startup=-1 # Load on startup priority of the Spring Web Services servlet.
spring.mvc.static-path-pattern=/** # Path pattern used for static resources.
spring.mvc.throw-exception-if-no-handler-found=false # Whether a "NoHandlerFoundException" should be
 thrown if no Handler was found to process a request.
spring.mvc.view.prefix= # Spring MVC view prefix.
spring.mvc.view.suffix= # Spring MVC view suffix.

# SPRING RESOURCES HANDLING (ResourceProperties)
spring.resources.add-mappings=true # Whether to enable default resource handling.
spring.resources.cache.cachecontrol.cache-private= # Indicate that the response message is intended for
 a single user and must not be stored by a shared cache.
spring.resources.cache.cachecontrol.cache-public= # Indicate that any cache may store the response.
spring.resources.cache.cachecontrol.max-age= # Maximum time the response should be cached, in seconds if
 no duration suffix is not specified.
spring.resources.cache.cachecontrol.must-revalidate= # Indicate that once it has become stale, a cache
 must not use the response without re-validating it with the server.
spring.resources.cache.cachecontrol.no-cache= # Indicate that the cached response can be reused only if
 re-validated with the server.
spring.resources.cache.cachecontrol.no-store= # Indicate to not cache the response in any case.
spring.resources.cache.cachecontrol.no-transform= # Indicate intermediaries (caches and others) that
 they should not transform the response content.
spring.resources.cache.cachecontrol.proxy-revalidate= # Same meaning as the "must-revalidate" directive,
 except that it does not apply to private caches.
spring.resources.cache.cachecontrol.s-max-age= # Maximum time the response should be cached by shared
 caches, in seconds if no duration suffix is not specified.
spring.resources.cache.cachecontrol.stale-if-error= # Maximum time the response may be used when errors
 are encountered, in seconds if no duration suffix is not specified.
spring.resources.cache.cachecontrol.stale-while-revalidate= # Maximum time the response can be served
 after it becomes stale, in seconds if no duration suffix is not specified.
spring.resources.cache.period= # Cache period for the resources served by the resource handler. If a
 duration suffix is not specified, seconds will be used.
spring.resources.chain.cache=true # Whether to enable caching in the Resource chain.
spring.resources.chain.enabled= # Whether to enable the Spring Resource Handling chain. By default,
 disabled unless at least one strategy has been enabled.
spring.resources.chain.gzipped=false # Whether to enable resolution of already gzipped resources.
spring.resources.chain.html-application-cache=false # Whether to enable HTML5 application cache manifest
 rewriting.
spring.resources.chain.strategy.content.enabled=false # Whether to enable the content Version Strategy.
spring.resources.chain.strategy.content.paths=/** # Comma-separated list of patterns to apply to the
 content Version Strategy.
spring.resources.chain.strategy.fixed.enabled=false # Whether to enable the fixed Version Strategy.
spring.resources.chain.strategy.fixed.paths=/** # Comma-separated list of patterns to apply to the fixed
 Version Strategy.
spring.resources.chain.strategy.fixed.version= # Version string to use for the fixed Version Strategy.
spring.resources.static-locations=classpath:/META-INF/resources/,classpath:/resources/,classpath:/
static/,classpath:/public/ # Locations of static resources.

# SPRING SESSION (SessionProperties)
spring.session.store-type= # Session store type.
spring.session.servlet.filter-order=-2147483598 # Session repository filter order.
spring.session.servlet.filter-dispatcher-types=ASYNC,ERROR,REQUEST # Session repository filter
 dispatcher types.

# SPRING SESSION HAZELCAST (HazelcastSessionProperties)
spring.session.hazelcast.flush-mode=on-save # Sessions flush mode.
spring.session.hazelcast.map-name=spring:session:sessions # Name of the map used to store sessions.

# SPRING SESSION JDBC (JdbcSessionProperties)
spring.session.jdbc.cleanup-cron=0 * * * * * # Cron expression for expired session cleanup job.
spring.session.jdbc.initialize-schema=embedded # Database schema initialization mode.
spring.session.jdbc.schema=classpath:org/springframework/session/jdbc/schema-@@platform@@.sql # Path to
 the SQL file to use to initialize the database schema.
spring.session.jdbc.table-name=SPRING_SESSION # Name of the database table used to store sessions.

# SPRING SESSION MONGODB (MongoSessionProperties)
spring.session.mongodb.collection-name=sessions # Collection name used to store sessions.

# SPRING SESSION REDIS (RedisSessionProperties)
spring.session.redis.cleanup-cron=0 * * * * * # Cron expression for expired session cleanup job.
spring.session.redis.flush-mode=on-save # Sessions flush mode.
spring.session.redis.namespace=spring:session # Namespace for keys used to store sessions.

# THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.cache=true # Whether to enable template caching.
spring.thymeleaf.check-template=true # Whether to check that the template exists before rendering it.
spring.thymeleaf.check-template-location=true # Whether to check that the templates location exists.
spring.thymeleaf.enabled=true # Whether to enable Thymeleaf view resolution for Web frameworks.
spring.thymeleaf.enable-spring-el-compiler=false # Enable the SpringEL compiler in SpringEL expressions.
spring.thymeleaf.encoding=UTF-8 # Template files encoding.
spring.thymeleaf.excluded-view-names= # Comma-separated list of view names that should be excluded from
 resolution.
spring.thymeleaf.mode=HTML5 # Template mode to be applied to templates. See also Thymeleaf's
 TemplateMode enum.
spring.thymeleaf.prefix=classpath:/templates/ # Prefix that gets prepended to view names when building a
 URL.
spring.thymeleaf.reactive.chunked-mode-view-names= # Comma-separated list of view names (patterns
 allowed) that should be the only ones executed in CHUNKED mode when a max chunk size is set.
spring.thymeleaf.reactive.full-mode-view-names= # Comma-separated list of view names (patterns allowed)
 that should be executed in FULL mode even if a max chunk size is set.
spring.thymeleaf.reactive.max-chunk-size= # Maximum size of data buffers used for writing to the
 response, in bytes.
spring.thymeleaf.reactive.media-types= # Media types supported by the view technology.
spring.thymeleaf.servlet.content-type=text/html # Content-Type value written to HTTP responses.
spring.thymeleaf.suffix=.html # Suffix that gets appended to view names when building a URL.
spring.thymeleaf.template-resolver-order= # Order of the template resolver in the chain.
spring.thymeleaf.view-names= # Comma-separated list of view names that can be resolved.

# SPRING WEBFLUX (WebFluxProperties)
spring.webflux.date-format= # Date format to use. For instance, `dd/MM/yyyy`.
spring.webflux.static-path-pattern=/** # Path pattern used for static resources.

# SPRING WEB SERVICES (WebServicesProperties)
spring.webservices.path=/services # Path that serves as the base URI for the services.
spring.webservices.servlet.init= # Servlet init parameters to pass to Spring Web Services.
spring.webservices.servlet.load-on-startup=-1 # Load on startup priority of the Spring Web Services
 servlet.
spring.webservices.wsdl-locations= # Comma-separated list of locations of WSDLs and accompanying XSDs to
 be exposed as beans.
```

```
# ----------------------------------------
# SECURITY PROPERTIES
# ----------------------------------------
# SECURITY (SecurityProperties)
spring.security.filter.order=0 # Security filter chain order.
spring.security.filter.dispatcher-types=ASYNC,ERROR,REQUEST # Security filter chain dispatcher types.
spring.security.user.name=user # Default user name.
spring.security.user.password= # Password for the default user name.
spring.security.user.roles= # Granted roles for the default user name.

# SECURITY OAUTH2 CLIENT (OAuth2ClientProperties)
spring.security.oauth2.client.provider.*= # OAuth provider details.
spring.security.oauth2.client.registration.*= # OAuth client registrations.

# ----------------------------------------
# DATA PROPERTIES
# ----------------------------------------

# FLYWAY (FlywayProperties)
spring.flyway.baseline-description= #
spring.flyway.baseline-on-migrate= #
spring.flyway.baseline-version=1 # Version to start migration
spring.flyway.check-location=true # Whether to check that migration scripts location exists.
spring.flyway.clean-disabled= #
spring.flyway.clean-on-validation-error= #
spring.flyway.dry-run-output= #
spring.flyway.enabled=true # Whether to enable flyway.
spring.flyway.encoding= #
spring.flyway.error-handlers= #
spring.flyway.group= #
spring.flyway.ignore-future-migrations= #
spring.flyway.ignore-missing-migrations= #
spring.flyway.init-sqls= # SQL statements to execute to initialize a connection immediately after
 obtaining it.
spring.flyway.installed-by= #
spring.flyway.locations=classpath:db/migration # The locations of migrations scripts.
spring.flyway.mixed= #
spring.flyway.out-of-order= #
spring.flyway.password= # JDBC password to use if you want Flyway to create its own DataSource.
spring.flyway.placeholder-prefix= #
spring.flyway.placeholder-replacement= #
spring.flyway.placeholder-suffix= #
spring.flyway.placeholders.*= #
spring.flyway.repeatable-sql-migration-prefix= #
spring.flyway.schemas= # schemas to update
spring.flyway.skip-default-callbacks= #
spring.flyway.skip-default-resolvers= #
spring.flyway.sql-migration-prefix=V #
spring.flyway.sql-migration-separator= #
spring.flyway.sql-migration-suffix=.sql #
spring.flyway.sql-migration-suffixes= #
spring.flyway.table= #
spring.flyway.target= #
spring.flyway.undo-sql-migration-prefix= #
spring.flyway.url= # JDBC url of the database to migrate. If not set, the primary configured data source
 is used.
spring.flyway.user= # Login user of the database to migrate.
spring.flyway.validate-on-migrate= #

# LIQUIBASE (LiquibaseProperties)
spring.liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml # Change log configuration
 path.
spring.liquibase.check-change-log-location=true # Whether to check that the change log location exists.
spring.liquibase.contexts= # Comma-separated list of runtime contexts to use.
spring.liquibase.default-schema= # Default database schema.
spring.liquibase.drop-first=false # Whether to first drop the database schema.
spring.liquibase.enabled=true # Whether to enable Liquibase support.
spring.liquibase.labels= # Comma-separated list of runtime labels to use.
spring.liquibase.parameters.*= # Change log parameters.
spring.liquibase.password= # Login password of the database to migrate.
spring.liquibase.rollback-file= # File to which rollback SQL is written when an update is performed.
spring.liquibase.url= # JDBC URL of the database to migrate. If not set, the primary configured data
 source is used.
spring.liquibase.user= # Login user of the database to migrate.

# COUCHBASE (CouchbaseProperties)
spring.couchbase.bootstrap-hosts= # Couchbase nodes (host or IP address) to bootstrap from.
spring.couchbase.bucket.name=default # Name of the bucket to connect to.
spring.couchbase.bucket.password=  # Password of the bucket.
spring.couchbase.env.endpoints.key-value=1 # Number of sockets per node against the Key/value service.
spring.couchbase.env.endpoints.query=1 # Number of sockets per node against the Query (N1QL) service.
spring.couchbase.env.endpoints.view=1 # Number of sockets per node against the view service.
spring.couchbase.env.ssl.enabled= # Whether to enable SSL support. Enabled automatically if a "keyStore"
 is provided, unless specified otherwise.
spring.couchbase.env.ssl.key-store= # Path to the JVM key store that holds the certificates.
spring.couchbase.env.ssl.key-store-password= # Password used to access the key store.
spring.couchbase.env.timeouts.connect=5000ms # Bucket connections timeouts.
spring.couchbase.env.timeouts.key-value=2500ms # Blocking operations performed on a specific key
 timeout.
spring.couchbase.env.timeouts.query=7500ms # N1QL query operations timeout.
spring.couchbase.env.timeouts.socket-connect=1000ms # Socket connect connections timeout.
spring.couchbase.env.timeouts.view=7500ms # Regular and geospatial view operations timeout.

# DAO (PersistenceExceptionTranslationAutoConfiguration)
spring.dao.exceptiontranslation.enabled=true # Whether to enable the
 PersistenceExceptionTranslationPostProcessor.

# CASSANDRA (CassandraProperties)
spring.data.cassandra.cluster-name= # Name of the Cassandra cluster.
spring.data.cassandra.compression=none # Compression supported by the Cassandra binary protocol.
spring.data.cassandra.connect-timeout= # Socket option: connection time out.
spring.data.cassandra.consistency-level= # Queries consistency level.
spring.data.cassandra.contact-points=localhost # Comma-separated list of cluster node addresses.
spring.data.cassandra.fetch-size= # Queries default fetch size.
spring.data.cassandra.keyspace-name= # Keyspace name to use.
spring.data.cassandra.load-balancing-policy= # Class name of the load balancing policy.
spring.data.cassandra.port= # Port of the Cassandra server.
spring.data.cassandra.password= # Login password of the server.
spring.data.cassandra.pool.heartbeat-interval=30 # Heartbeat interval after which a message is sent on
 an idle connection to make sure it's still alive. If a duration suffix is not specified, seconds will
 be used.
spring.data.cassandra.pool.idle-timeout=120 # Idle timeout before an idle connection is removed. If a
 duration suffix is not specified, seconds will be used.
spring.data.cassandra.pool.max-queue-size=256 # Maximum number of requests that get queued if no
 connection is available.
spring.data.cassandra.pool.pool-timeout=5000ms # Pool timeout when trying to acquire a connection from a
 host's pool.
spring.data.cassandra.read-timeout= # Socket option: read time out.
spring.data.cassandra.reconnection-policy= # Reconnection policy class.
spring.data.cassandra.repositories.type=auto # Type of Cassandra repositories to enable.
spring.data.cassandra.retry-policy= # Class name of the retry policy.
spring.data.cassandra.serial-consistency-level= # Queries serial consistency level.
spring.data.cassandra.schema-action=none # Schema action to take at startup.
spring.data.cassandra.ssl=false # Enable SSL support.
spring.data.cassandra.username= # Login user of the server.

# DATA COUCHBASE (CouchbaseDataProperties)
spring.data.couchbase.auto-index=false # Automatically create views and indexes.
spring.data.couchbase.consistency=read-your-own-writes # Consistency to apply by default on generated
 queries.
spring.data.couchbase.repositories.type=auto # Type of Couchbase repositories to enable.

# ELASTICSEARCH (ElasticsearchProperties)
spring.data.elasticsearch.cluster-name=elasticsearch # Elasticsearch cluster name.
spring.data.elasticsearch.cluster-nodes= # Comma-separated list of cluster node addresses.
spring.data.elasticsearch.properties.*= # Additional properties used to configure the client.
spring.data.elasticsearch.repositories.enabled=true # Whether to enable Elasticsearch repositories.

# DATA LDAP
spring.data.ldap.repositories.enabled=true # Enable LDAP repositories.
```

```
# MONGODB (MongoProperties)
spring.data.mongodb.authentication-database= # Authentication database name.
spring.data.mongodb.database=test # Database name.
spring.data.mongodb.field-naming-strategy= # Fully qualified name of the FieldNamingStrategy to use.
spring.data.mongodb.grid-fs-database= # GridFS database name.
spring.data.mongodb.host=localhost # Mongo server host. Cannot be set with URI.
spring.data.mongodb.password= # Login password of the mongo server. Cannot be set with URI.
spring.data.mongodb.port=27017 # Mongo server port. Cannot be set with URI.
spring.data.mongodb.repositories.type=true # Type of Mongo repositories to enable.
spring.data.mongodb.uri=mongodb://localhost/test # Mongo database URI. Cannot be set with host, port and
 credentials.
spring.data.mongodb.username= # Login user of the mongo server. Cannot be set with URI.

# DATA REDIS
spring.data.redis.repositories.enabled=true # Whether to enable Redis repositories.

# NEO4J (Neo4jProperties)
spring.data.neo4j.auto-index=none # Auto index mode.
spring.data.neo4j.embedded.enabled=true # Whether to enable embedded mode if the embedded driver is
 available.
spring.data.neo4j.open-in-view=true # Register OpenSessionInViewInterceptor. Binds a Neo4j Session to
 the thread for the entire processing of the request.
spring.data.neo4j.password= # Login password of the server.
spring.data.neo4j.repositories.enabled=true # Whether to enable Neo4j repositories.
spring.data.neo4j.uri= # URI used by the driver. Auto-detected by default.
spring.data.neo4j.username= # Login user of the server.

# DATA REST (RepositoryRestProperties)
spring.data.rest.base-path= # Base path to be used by Spring Data REST to expose repository resources.
spring.data.rest.default-media-type= # Content type to use as a default when none is specified.
spring.data.rest.default-page-size= # Default size of pages.
spring.data.rest.detection-strategy=default # Strategy to use to determine which repositories get
 exposed.
spring.data.rest.enable-enum-translation= # Whether to enable enum value translation through the Spring
 Data REST default resource bundle.
spring.data.rest.limit-param-name= # Name of the URL query string parameter that indicates how many
 results to return at once.
spring.data.rest.max-page-size= # Maximum size of pages.
spring.data.rest.page-param-name= # Name of the URL query string parameter that indicates what page to
 return.
spring.data.rest.return-body-on-create= # Whether to return a response body after creating an entity.
spring.data.rest.return-body-on-update= # Whether to return a response body after updating an entity.
spring.data.rest.sort-param-name= # Name of the URL query string parameter that indicates what direction
 to sort results.

# SOLR (SolrProperties)
spring.data.solr.host=http://127.0.0.1:8983/solr # Solr host. Ignored if "zk-host" is set.
spring.data.solr.repositories.enabled=true # Whether to enable Solr repositories.
spring.data.solr.zk-host= # ZooKeeper host address in the form HOST:PORT.

# DATA WEB (SpringDataWebProperties)
spring.data.web.pageable.default-page-size=20 # Default page size.
spring.data.web.pageable.max-page-size=2000 # Maximum page size to be accepted.
spring.data.web.pageable.one-indexed-parameters=false # Whether to expose and assume 1-based page number
 indexes.
spring.data.web.pageable.page-parameter=page # Page index parameter name.
spring.data.web.pageable.prefix= # General prefix to be prepended to the page number and page size
 parameters.
spring.data.web.pageable.qualifier-delimiter=_ # Delimiter to be used between the qualifier and the
 actual page number and size properties.
spring.data.web.pageable.size-parameter=size # Page size parameter name.
spring.data.web.sort.sort-parameter=sort # Sort parameter name.

# DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.continue-on-error=false # Whether to stop if an error occurs while initializing the
 database.
spring.datasource.data= # Data (DML) script resource references.
spring.datasource.data-username= # Username of the database to execute DML scripts (if different).
spring.datasource.data-password= # Password of the database to execute DML scripts (if different).
spring.datasource.dbcp2.*= # Commons DBCP2 specific settings
spring.datasource.driver-class-name= # Fully qualified name of the JDBC driver. Auto-detected based on
 the URL by default.
spring.datasource.generate-unique-name=false # Whether to generate a random datasource name.
spring.datasource.hikari.*= # Hikari specific settings
spring.datasource.initialization-mode=embedded # Initialize the datasource with available DDL and DML
 scripts.
spring.datasource.jmx-enabled=false # Whether to enable JMX support (if provided by the underlying
 pool).
spring.datasource.jndi-name= # JNDI location of the datasource. Class, url, username & password are
 ignored when set.
spring.datasource.name= # Name of the datasource. Default to "testdb" when using an embedded database.
spring.datasource.password= # Login password of the database.
spring.datasource.platform=all # Platform to use in the DDL or DML scripts (such as schema-
${platform}.sql or data-${platform}.sql).
spring.datasource.schema= # Schema (DDL) script resource references.
spring.datasource.schema-username= # Username of the database to execute DDL scripts (if different).
spring.datasource.schema-password= # Password of the database to execute DDL scripts (if different).
spring.datasource.separator=; # Statement separator in SQL initialization scripts.
spring.datasource.sql-script-encoding= # SQL scripts encoding.
spring.datasource.tomcat.*= # Tomcat datasource specific settings
spring.datasource.type= # Fully qualified name of the connection pool implementation to use. By default,
 it is auto-detected from the classpath.
spring.datasource.url= # JDBC URL of the database.
spring.datasource.username= # Login username of the database.
spring.datasource.xa.data-source-class-name= # XA datasource fully qualified name.
spring.datasource.xa.properties= # Properties to pass to the XA data source.

# JEST (Elasticsearch HTTP client) (JestProperties)
spring.elasticsearch.jest.connection-timeout=3s # Connection timeout.
spring.elasticsearch.jest.multi-threaded=true # Whether to enable connection requests from multiple
 execution threads.
spring.elasticsearch.jest.password= # Login password.
spring.elasticsearch.jest.proxy.host= # Proxy host the HTTP client should use.
spring.elasticsearch.jest.proxy.port= # Proxy port the HTTP client should use.
spring.elasticsearch.jest.read-timeout=3s # Read timeout.
spring.elasticsearch.jest.uris=http://localhost:9200 # Comma-separated list of the Elasticsearch
 instances to use.
spring.elasticsearch.jest.username= # Login username.

# H2 Web Console (H2ConsoleProperties)
spring.h2.console.enabled=false # Whether to enable the console.
spring.h2.console.path=/h2-console # Path at which the console is available.
spring.h2.console.settings.trace=false # Whether to enable trace output.
spring.h2.console.settings.web-allow-others=false # Whether to enable remote access.

# InfluxDB (InfluxDbProperties)
spring.influx.password= # Login password.
spring.influx.url= # URL of the InfluxDB instance to which to connect.
spring.influx.user= # Login user.

# JOOQ (JooqProperties)
spring.jooq.sql-dialect= # SQL dialect to use. Auto-detected by default.

# JDBC (JdbcProperties)
spring.jdbc.template.fetch-size=-1 # Number of rows that should be fetched from the database when more
 rows are needed.
spring.jdbc.template.max-rows=-1 # Maximum number of rows.
spring.jdbc.template.query-timeout= # Query timeout. If a duration suffix is not specified, seconds will
 be used.

# JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.data.jpa.repositories.enabled=true # Whether to enable JPA repositories.
spring.jpa.database= # Target database to operate on, auto-detected by default. Can be alternatively set
 using the "databasePlatform" property.
spring.jpa.database-platform= # Name of the target database to operate on, auto-detected by default. Can
 be alternatively set using the "Database" enum.
spring.jpa.generate-ddl=false # Whether to initialize the schema on startup.
spring.jpa.hibernate.ddl-auto= # DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto"
 property. Defaults to "create-drop" when using an embedded database and no schema manager was detected.
 Otherwise, defaults to "none".
```

```
spring.jpa.hibernate.naming.implicit-strategy= # Hibernate 5 implicit naming strategy fully qualified
 name.
spring.jpa.hibernate.naming.physical-strategy= # Hibernate 5 physical naming strategy fully qualified
 name.
spring.jpa.hibernate.use-new-id-generator-mappings= # Whether to use Hibernate's newer
 IdentifierGenerator for AUTO, TABLE and SEQUENCE.
spring.jpa.mapping-resources= # Mapping resources (equivalent to "mapping-file" entries in
 persistence.xml).
spring.jpa.open-in-view=true # Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to
 the thread for the entire processing of the request.
spring.jpa.properties.*= # Additional native properties to set on the JPA provider.
spring.jpa.show-sql=false # Whether to enable logging of SQL statements.

# JTA (JtaAutoConfiguration)
spring.jta.enabled=true # Whether to enable JTA support.
spring.jta.log-dir= # Transaction logs directory.
spring.jta.transaction-manager-id= # Transaction manager unique identifier.

# ATOMIKOS (AtomikosProperties)
spring.jta.atomikos.connectionfactory.borrow-connection-timeout=30 # Timeout, in seconds, for borrowing
 connections from the pool.
spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag=true # Whether to ignore the
 transacted flag when creating session.
spring.jta.atomikos.connectionfactory.local-transaction-mode=false # Whether local transactions are
 desired.
spring.jta.atomikos.connectionfactory.maintenance-interval=60 # The time, in seconds, between runs of
 the pool's maintenance thread.
spring.jta.atomikos.connectionfactory.max-idle-time=60 # The time, in seconds, after which connections
 are cleaned up from the pool.
spring.jta.atomikos.connectionfactory.max-lifetime=0 # The time, in seconds, that a connection can be
 pooled for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.max-pool-size=1 # The maximum size of the pool.
spring.jta.atomikos.connectionfactory.min-pool-size=1 # The minimum size of the pool.
spring.jta.atomikos.connectionfactory.reap-timeout=0 # The reap timeout, in seconds, for borrowed
 connections. 0 denotes no limit.
spring.jta.atomikos.connectionfactory.unique-resource-name=jmsConnectionFactory # The unique name used
 to identify the resource during recovery.
spring.jta.atomikos.connectionfactory.xa-connection-factory-class-name= # Vendor-specific implementation
 of XAConnectionFactory.
spring.jta.atomikos.connectionfactory.xa-properties= # Vendor-specific XA properties.
spring.jta.atomikos.datasource.borrow-connection-timeout=30 # Timeout, in seconds, for borrowing
 connections from the pool.
spring.jta.atomikos.datasource.concurrent-connection-validation= # Whether to use concurrent connection
 validation.
spring.jta.atomikos.datasource.default-isolation-level= # Default isolation level of connections
 provided by the pool.
spring.jta.atomikos.datasource.login-timeout= # Timeout, in seconds, for establishing a database
 connection.
spring.jta.atomikos.datasource.maintenance-interval=60 # The time, in seconds, between runs of the
 pool's maintenance thread.
spring.jta.atomikos.datasource.max-idle-time=60 # The time, in seconds, after which connections are
 cleaned up from the pool.
spring.jta.atomikos.datasource.max-lifetime=0 # The time, in seconds, that a connection can be pooled
 for before being destroyed. 0 denotes no limit.
spring.jta.atomikos.datasource.max-pool-size=1 # The maximum size of the pool.
spring.jta.atomikos.datasource.min-pool-size=1 # The minimum size of the pool.
spring.jta.atomikos.datasource.reap-timeout=0 # The reap timeout, in seconds, for borrowed connections.
 0 denotes no limit.
spring.jta.atomikos.datasource.test-query= # SQL query or statement used to validate a connection before
 returning it.
spring.jta.atomikos.datasource.unique-resource-name=dataSource # The unique name used to identify the
 resource during recovery.
spring.jta.atomikos.datasource.xa-data-source-class-name= # Vendor-specific implementation of
 XAConnectionFactory.
spring.jta.atomikos.datasource.xa-properties= # Vendor-specific XA properties.
spring.jta.atomikos.properties.allow-sub-transactions=true # Specify whether sub-transactions are
 allowed.
spring.jta.atomikos.properties.checkpoint-interval=500 # Interval between checkpoints, in milliseconds.
spring.jta.atomikos.properties.default-jta-timeout=10000 # Default timeout for JTA transactions, in
 milliseconds.
```

```
spring.jta.atomikos.properties.default-max-wait-time-on-shutdown=9223372036854775807 # How long should
 normal shutdown (no-force) wait for transactions to complete.
spring.jta.atomikos.properties.enable-logging=true # Whether to enable disk logging.
spring.jta.atomikos.properties.force-shutdown-on-vm-exit=false # Whether a VM shutdown should trigger
 forced shutdown of the transaction core.
spring.jta.atomikos.properties.log-base-dir= # Directory in which the log files should be stored.
spring.jta.atomikos.properties.log-base-name=tmlog # Transactions log file base name.
spring.jta.atomikos.properties.max-actives=50 # Maximum number of active transactions.
spring.jta.atomikos.properties.max-timeout=30m # Maximum timeout that can be allowed for transactions.
spring.jta.atomikos.properties.recovery.delay=10000ms # Delay between two recovery scans.
spring.jta.atomikos.properties.recovery.forget-orphaned-log-entries-delay=86400000 # Delay after which
 recovery can cleanup pending ('orphaned') log entries.
spring.jta.atomikos.properties.recovery.max-retries=5 # Number of retry attempts to commit the
 transaction before throwing an exception.
spring.jta.atomikos.properties.recovery.retry-interval=10000ms # Delay between retry attempts.
spring.jta.atomikos.properties.serial-jta-transactions=true # Whether sub-transactions should be joined
 when possible.
spring.jta.atomikos.properties.service= # Transaction manager implementation that should be started.
spring.jta.atomikos.properties.threaded-two-phase-commit=false # Whether to use different (and
 concurrent) threads for two-phase commit on the participating resources.
spring.jta.atomikos.properties.transaction-manager-unique-name= # The transaction manager's unique name.

# BITRONIX
spring.jta.bitronix.connectionfactory.acquire-increment=1 # Number of connections to create when growing
 the pool.
spring.jta.bitronix.connectionfactory.acquisition-interval=1 # Time, in seconds, to wait before trying
 to acquire a connection again after an invalid connection was acquired.
spring.jta.bitronix.connectionfactory.acquisition-timeout=30 # Timeout, in seconds, for acquiring
 connections from the pool.
spring.jta.bitronix.connectionfactory.allow-local-transactions=true # Whether the transaction manager
 should allow mixing XA and non-XA transactions.
spring.jta.bitronix.connectionfactory.apply-transaction-timeout=false # Whether the transaction timeout
 should be set on the XAResource when it is enlisted.
spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled=true # Whether resources should be
 enlisted and delisted automatically.
spring.jta.bitronix.connectionfactory.cache-producers-consumers=true # Whether producers and consumers
 should be cached.
spring.jta.bitronix.connectionfactory.class-name= # Underlying implementation class name of the XA
 resource.
spring.jta.bitronix.connectionfactory.defer-connection-release=true # Whether the provider can run many
 transactions on the same connection and supports transaction interleaving.
spring.jta.bitronix.connectionfactory.disabled= # Whether this resource is disabled, meaning it's
 temporarily forbidden to acquire a connection from its pool.
spring.jta.bitronix.connectionfactory.driver-properties= # Properties that should be set on the
 underlying implementation.
spring.jta.bitronix.connectionfactory.failed= # Mark this resource producer as failed.
spring.jta.bitronix.connectionfactory.ignore-recovery-failures=false # Whether recovery failures should
 be ignored.
spring.jta.bitronix.connectionfactory.max-idle-time=60 # The time, in seconds, after which connections
 are cleaned up from the pool.
spring.jta.bitronix.connectionfactory.max-pool-size=10 # The maximum size of the pool. 0 denotes no
 limit.
spring.jta.bitronix.connectionfactory.min-pool-size=0 # The minimum size of the pool.
spring.jta.bitronix.connectionfactory.password= # The password to use to connect to the JMS provider.
spring.jta.bitronix.connectionfactory.share-transaction-connections=false #  Whether connections in the
 ACCESSIBLE state can be shared within the context of a transaction.
spring.jta.bitronix.connectionfactory.test-connections=true # Whether connections should be tested when
 acquired from the pool.
spring.jta.bitronix.connectionfactory.two-pc-ordering-position=1 # The position that this
 resource should take during two-phase commit (always first is Integer.MIN_VALUE, always last is
 Integer.MAX_VALUE).
spring.jta.bitronix.connectionfactory.unique-name=jmsConnectionFactory # The unique name used to
 identify the resource during recovery.
spring.jta.bitronix.connectionfactory.use-tm-join=true Whether TMJOIN should be used when starting
 XAResources.
spring.jta.bitronix.connectionfactory.user= # The user to use to connect to the JMS provider.
spring.jta.bitronix.datasource.acquire-increment=1 # Number of connections to create when growing the
 pool.
spring.jta.bitronix.datasource.acquisition-interval=1 # Time, in seconds, to wait before trying to
 acquire a connection again after an invalid connection was acquired.
```

```
spring.jta.bitronix.datasource.acquisition-timeout=30 # Timeout, in seconds, for acquiring connections
 from the pool.
spring.jta.bitronix.datasource.allow-local-transactions=true # Whether the transaction manager should
 allow mixing XA and non-XA transactions.
spring.jta.bitronix.datasource.apply-transaction-timeout=false # Whether the transaction timeout should
 be set on the XAResource when it is enlisted.
spring.jta.bitronix.datasource.automatic-enlisting-enabled=true # Whether resources should be enlisted
 and delisted automatically.
spring.jta.bitronix.datasource.class-name= # Underlying implementation class name of the XA resource.
spring.jta.bitronix.datasource.cursor-holdability= # The default cursor holdability for connections.
spring.jta.bitronix.datasource.defer-connection-release=true # Whether the database can run many
 transactions on the same connection and supports transaction interleaving.
spring.jta.bitronix.datasource.disabled= # Whether this resource is disabled, meaning it's temporarily
 forbidden to acquire a connection from its pool.
spring.jta.bitronix.datasource.driver-properties= # Properties that should be set on the underlying
 implementation.
spring.jta.bitronix.datasource.enable-jdbc4-connection-test= # Whether Connection.isValid() is called
 when acquiring a connection from the pool.
spring.jta.bitronix.datasource.failed= # Mark this resource producer as failed.
spring.jta.bitronix.datasource.ignore-recovery-failures=false # Whether recovery failures should be
 ignored.
spring.jta.bitronix.datasource.isolation-level= # The default isolation level for connections.
spring.jta.bitronix.datasource.local-auto-commit= # The default auto-commit mode for local transactions.
spring.jta.bitronix.datasource.login-timeout= # Timeout, in seconds, for establishing a database
 connection.
spring.jta.bitronix.datasource.max-idle-time=60 # The time, in seconds, after which connections are
 cleaned up from the pool.
spring.jta.bitronix.datasource.max-pool-size=10 # The maximum size of the pool. 0 denotes no limit.
spring.jta.bitronix.datasource.min-pool-size=0 # The minimum size of the pool.
spring.jta.bitronix.datasource.prepared-statement-cache-size=0 # The target size of the prepared
 statement cache. 0 disables the cache.
spring.jta.bitronix.datasource.share-transaction-connections=false #  Whether connections in the
 ACCESSIBLE state can be shared within the context of a transaction.
spring.jta.bitronix.datasource.test-query= # SQL query or statement used to validate a connection before
 returning it.
spring.jta.bitronix.datasource.two-pc-ordering-position=1 # The position that this resource should take
 during two-phase commit (always first is Integer.MIN_VALUE, and always last is Integer.MAX_VALUE).
spring.jta.bitronix.datasource.unique-name=dataSource # The unique name used to identify the resource
 during recovery.
spring.jta.bitronix.datasource.use-tm-join=true Whether TMJOIN should be used when starting XAResources.
spring.jta.bitronix.properties.allow-multiple-lrc=false # Whether to allow multiple LRC resources to be
 enlisted into the same transaction.
spring.jta.bitronix.properties.asynchronous2-pc=false # Enable asynchronously execution of two phase
 commit.
spring.jta.bitronix.properties.background-recovery-interval-seconds=60 # Interval in seconds at which to
 run the recovery process in the background.
spring.jta.bitronix.properties.current-node-only-recovery=true # Whether to recover only the current
 node.
spring.jta.bitronix.properties.debug-zero-resource-transaction=false # Whether to log the creation and
 commit call stacks of transactions executed without a single enlisted resource.
spring.jta.bitronix.properties.default-transaction-timeout=60 # Default transaction timeout, in seconds.
spring.jta.bitronix.properties.disable-jmx=false # Whether to enable JMX support.
spring.jta.bitronix.properties.exception-analyzer= # Set the fully qualified name of the exception
 analyzer implementation to use.
spring.jta.bitronix.properties.filter-log-status=false # Whether to enable filtering of logs so that
 only mandatory logs are written.
spring.jta.bitronix.properties.force-batching-enabled=true #  Whether disk forces are batched.
spring.jta.bitronix.properties.forced-write-enabled=true # Whether logs are forced to disk.
spring.jta.bitronix.properties.graceful-shutdown-interval=60 # Maximum amount of seconds the TM waits
 for transactions to get done before aborting them at shutdown time.
spring.jta.bitronix.properties.jndi-transaction-synchronization-registry-name= # JNDI name of the
 TransactionSynchronizationRegistry.
spring.jta.bitronix.properties.jndi-user-transaction-name= # JNDI name of the UserTransaction.
spring.jta.bitronix.properties.journal=disk # Name of the journal. Can be 'disk', 'null', or a class
 name.
spring.jta.bitronix.properties.log-part1-filename=btm1.tlog # Name of the first fragment of the journal.
spring.jta.bitronix.properties.log-part2-filename=btm2.tlog # Name of the second fragment of the
 journal.
spring.jta.bitronix.properties.max-log-size-in-mb=2 # Maximum size in megabytes of the journal
 fragments.
```

```
spring.jta.bitronix.properties.resource-configuration-filename= # ResourceLoader configuration file
 name.
spring.jta.bitronix.properties.server-id= # ASCII ID that must uniquely identify this TM instance.
 Defaults to the machine's IP address.
spring.jta.bitronix.properties.skip-corrupted-logs=false # Skip corrupted transactions log entries.
spring.jta.bitronix.properties.warn-about-zero-resource-transaction=true # Whether to log a warning for
 transactions executed without a single enlisted resource.

# NARAYANA (NarayanaProperties)
spring.jta.narayana.default-timeout=60s # Transaction timeout. If a duration suffix is not specified,
 seconds will be used.
spring.jta.narayana.expiry-
scanners=com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner # Comma-
separated list of expiry scanners.
spring.jta.narayana.log-dir= # Transaction object store directory.
spring.jta.narayana.one-phase-commit=true # Whether to enable one phase commit optimization.
spring.jta.narayana.periodic-recovery-period=120s # Interval in which periodic recovery scans are
 performed. If a duration suffix is not specified, seconds will be used.
spring.jta.narayana.recovery-backoff-period=10s # Back off period between first and second phases of the
 recovery scan. If a duration suffix is not specified, seconds will be used.
spring.jta.narayana.recovery-db-pass= # Database password to be used by the recovery manager.
spring.jta.narayana.recovery-db-user= # Database username to be used by the recovery manager.
spring.jta.narayana.recovery-jms-pass= # JMS password to be used by the recovery manager.
spring.jta.narayana.recovery-jms-user= # JMS username to be used by the recovery manager.
spring.jta.narayana.recovery-modules= # Comma-separated list of recovery modules.
spring.jta.narayana.transaction-manager-id=1 # Unique transaction manager id.
spring.jta.narayana.xa-resource-orphan-filters= # Comma-separated list of orphan filters.

# EMBEDDED MONGODB (EmbeddedMongoProperties)
spring.mongodb.embedded.features=SYNC_DELAY # Comma-separated list of features to enable.
spring.mongodb.embedded.storage.database-dir= # Directory used for data storage.
spring.mongodb.embedded.storage.oplog-size= # Maximum size of the oplog, in megabytes.
spring.mongodb.embedded.storage.repl-set-name= # Name of the replica set.
spring.mongodb.embedded.version=2.6.10 # Version of Mongo to use.

# REDIS (RedisProperties)
spring.redis.cluster.max-redirects= # Maximum number of redirects to follow when executing commands
 across the cluster.
spring.redis.cluster.nodes= # Comma-separated list of "host:port" pairs to bootstrap from.
spring.redis.database=0 # Database index used by the connection factory.
spring.redis.url= # Connection URL. Overrides host, port, and password. User is ignored. Example:
 redis://user:password@example.com:6379
spring.redis.host=localhost # Redis server host.
spring.redis.jedis.pool.max-active=8 # Max number of connections that can be allocated by the pool at a
 given time. Use a negative value for no limit.
spring.redis.jedis.pool.max-idle=8 # Max number of "idle" connections in the pool. Use a negative value
 to indicate an unlimited number of idle connections.
spring.redis.jedis.pool.max-wait=-1ms # Maximum amount of time a connection allocation should block
 before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.
spring.redis.jedis.pool.min-idle=0 # Target for the minimum number of idle connections to maintain in
 the pool. This setting only has an effect if it is positive.
spring.redis.lettuce.pool.max-active=8 # Maximum number of connections that can be allocated by the pool
 at a given time. Use a negative value for no limit.
spring.redis.lettuce.pool.max-idle=8 # Maximum number of "idle" connections in the pool. Use a negative
 value to indicate an unlimited number of idle connections.
spring.redis.lettuce.pool.max-wait=-1ms # Maximum amount of time a connection allocation should block
 before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.
spring.redis.lettuce.pool.min-idle=0 # Target for the minimum number of idle connections to maintain in
 the pool. This setting only has an effect if it is positive.
spring.redis.lettuce.shutdown-timeout=100ms # Shutdown timeout.
spring.redis.password= # Login password of the redis server.
spring.redis.port=6379 # Redis server port.
spring.redis.sentinel.master= # Name of the Redis server.
spring.redis.sentinel.nodes= # Comma-separated list of "host:port" pairs.
spring.redis.ssl=false # Whether to enable SSL support.
spring.redis.timeout=0 # Connection timeout.

# TRANSACTION (TransactionProperties)
spring.transaction.default-timeout= # Default transaction timeout. If a duration suffix is not
 specified, seconds will be used.
spring.transaction.rollback-on-commit-failure= # Whether to roll back on commit failures.
```

```
# ----------------------------------------
# INTEGRATION PROPERTIES
# ----------------------------------------

# ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url= # URL of the ActiveMQ broker. Auto-generated by default.
spring.activemq.close-timeout=15s # Time to wait before considering a close complete.
spring.activemq.in-memory=true # Whether the default broker URL should be in memory. Ignored if an
 explicit broker has been specified.
spring.activemq.non-blocking-redelivery=false # Whether to stop message delivery before re-delivering
 messages from a rolled back transaction. This implies that message order is not preserved when this is
 enabled.
spring.activemq.password= # Login password of the broker.
spring.activemq.send-timeout=0 # Time to wait on message sends for a response. Set it to 0 to wait
 forever.
spring.activemq.user= # Login user of the broker.
spring.activemq.packages.trust-all= # Whether to trust all packages.
spring.activemq.packages.trusted= # Comma-separated list of specific packages to trust (when not
 trusting all packages).
spring.activemq.pool.block-if-full=true # Whether to block when a connection is requested and the pool
 is full. Set it to false to throw a "JMSException" instead.
spring.activemq.pool.block-if-full-timeout=-1ms # Blocking period before throwing an exception if the
 pool is still full.
spring.activemq.pool.create-connection-on-startup=true # Whether to create a connection on startup. Can
 be used to warm up the pool on startup.
spring.activemq.pool.enabled=false # Whether a PooledConnectionFactory should be created, instead of a
 regular ConnectionFactory.
spring.activemq.pool.expiry-timeout=0ms # Connection expiration timeout.
spring.activemq.pool.idle-timeout=30s # Connection idle timeout.
spring.activemq.pool.max-connections=1 # Maximum number of pooled connections.
spring.activemq.pool.maximum-active-session-per-connection=500 # Maximum number of active sessions per
 connection.
spring.activemq.pool.reconnect-on-exception=true # Reset the connection when a "JMSException" occurs.
spring.activemq.pool.time-between-expiration-check=-1ms # Time to sleep between runs of the idle
 connection eviction thread. When negative, no idle connection eviction thread runs.
spring.activemq.pool.use-anonymous-producers=true # Whether to use only one anonymous "MessageProducer"
 instance. Set it to false to create one "MessageProducer" every time one is required.

# ARTEMIS (ArtemisProperties)
spring.artemis.embedded.cluster-password= # Cluster password. Randomly generated on startup by default.
spring.artemis.embedded.data-directory= # Journal file directory. Not necessary if persistence is turned
 off.
spring.artemis.embedded.enabled=true # Whether to enable embedded mode if the Artemis server APIs are
 available.
spring.artemis.embedded.persistent=false # Whether to enable persistent store.
spring.artemis.embedded.queues= # Comma-separated list of queues to create on startup.
spring.artemis.embedded.server-id= # Server ID. By default, an auto-incremented counter is used.
spring.artemis.embedded.topics= # Comma-separated list of topics to create on startup.
spring.artemis.host=localhost # Artemis broker host.
spring.artemis.mode= # Artemis deployment mode, auto-detected by default.
spring.artemis.password= # Login password of the broker.
spring.artemis.port=61616 # Artemis broker port.
spring.artemis.user= # Login user of the broker.

# SPRING BATCH (BatchProperties)
spring.batch.initialize-schema=embedded # Database schema initialization mode.
spring.batch.job.enabled=true # Execute all Spring Batch jobs in the context on startup.
spring.batch.job.names= # Comma-separated list of job names to execute on startup (for instance,
 `job1,job2`). By default, all Jobs found in the context are executed.
spring.batch.schema=classpath:org/springframework/batch/core/schema-@@platform@@.sql # Path to the SQL
 file to use to initialize the database schema.
spring.batch.table-prefix= # Table prefix for all the batch meta-data tables.

# SPRING INTEGRATION (IntegrationProperties)
spring.integration.jdbc.initialize-schema=embedded # Database schema initialization mode.
spring.integration.jdbc.schema=classpath:org/springframework/integration/jdbc/schema-@@platform@@.sql #
 Path to the SQL file to use to initialize the database schema.
```

```
# JMS (JmsProperties)
spring.jms.jndi-name= # Connection factory JNDI name. When set, takes precedence to others connection
 factory auto-configurations.
spring.jms.listener.acknowledge-mode= # Acknowledge mode of the container. By default, the listener is
 transacted with automatic acknowledgment.
spring.jms.listener.auto-startup=true # Start the container automatically on startup.
spring.jms.listener.concurrency= # Minimum number of concurrent consumers.
spring.jms.listener.max-concurrency= # Maximum number of concurrent consumers.
spring.jms.pub-sub-domain=false # Whether the default destination type is topic.
spring.jms.template.default-destination= # Default destination to use on send and receive operations
 that do not have a destination parameter.
spring.jms.template.delivery-delay= # Delivery delay to use for send calls.
spring.jms.template.delivery-mode= # Delivery mode. Enables QoS (Quality of Service) when set.
spring.jms.template.priority= # Priority of a message when sending. Enables QoS (Quality of Service)
 when set.
spring.jms.template.qos-enabled= # Whether to enable explicit QoS (Quality of Service) when sending a
 message.
spring.jms.template.receive-timeout= # Timeout to use for receive calls.
spring.jms.template.time-to-live= # Time-to-live of a message when sending. Enable QoS (Quality of
 Service) when set.

# APACHE KAFKA (KafkaProperties)
spring.kafka.admin.client-id= # ID to pass to the server when making requests. Used for server-side
 logging.
spring.kafka.admin.fail-fast=false # Whether to fail fast if the broker is not available on startup.
spring.kafka.admin.properties.*= # Additional admin-specific properties used to configure the client.
spring.kafka.admin.ssl.key-password= # Password of the private key in the key store file.
spring.kafka.admin.ssl.keystore-location= # Location of the key store file.
spring.kafka.admin.ssl.keystore-password= # Password of the key store file.
spring.kafka.admin.ssl.truststore-location= # Location of the trust store file.
spring.kafka.admin.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.bootstrap-servers= # Comma-delimited list of host:port pairs to use for establishing the
 initial connection to the Kafka cluster.
spring.kafka.client-id= # ID to pass to the server when making requests. Used for server-side logging.
spring.kafka.consumer.auto-commit-interval= # Frequency with which the consumer offsets are auto-
committed to Kafka if 'enable.auto.commit' is set to true.
spring.kafka.consumer.auto-offset-reset= # What to do when there is no initial offset in Kafka or if the
 current offset no longer exists on the server.
spring.kafka.consumer.bootstrap-servers= # Comma-delimited list of host:port pairs to use for
 establishing the initial connection to the Kafka cluster.
spring.kafka.consumer.client-id= # ID to pass to the server when making requests. Used for server-side
 logging.
spring.kafka.consumer.enable-auto-commit= # Whether the consumer's offset is periodically committed in
 the background.
spring.kafka.consumer.fetch-max-wait= # Maximum amount of time the server blocks before answering
 the fetch request if there isn't sufficient data to immediately satisfy the requirement given by
 "fetch.min.bytes".
spring.kafka.consumer.fetch-min-size= # Minimum amount of data, in bytes, the server should return for a
 fetch request.
spring.kafka.consumer.group-id= # Unique string that identifies the consumer group to which this
 consumer belongs.
spring.kafka.consumer.heartbeat-interval= # Expected time between heartbeats to the consumer
 coordinator.
spring.kafka.consumer.key-deserializer= # Deserializer class for keys.
spring.kafka.consumer.max-poll-records= # Maximum number of records returned in a single call to poll().
spring.kafka.consumer.properties.*= # Additional consumer-specific properties used to configure the
 client.
spring.kafka.consumer.ssl.key-password= # Password of the private key in the key store file.
spring.kafka.consumer.ssl.keystore-location= # Location of the key store file.
spring.kafka.consumer.ssl.keystore-password= # Store password for the key store file.
spring.kafka.consumer.ssl.truststore-location= # Location of the trust store file.
spring.kafka.consumer.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.consumer.value-deserializer= # Deserializer class for values.
spring.kafka.jaas.control-flag=required # Control flag for login configuration.
spring.kafka.jaas.enabled= # Whether to enable JAAS configuration.
spring.kafka.jaas.login-module=com.sun.security.auth.module.Krb5LoginModule # Login module.
spring.kafka.jaas.options= # Additional JAAS options.
spring.kafka.listener.ack-count= # Number of records between offset commits when ackMode is "COUNT" or
 "COUNT_TIME".
spring.kafka.listener.ack-mode= # Listener AckMode. See the spring-kafka documentation.
spring.kafka.listener.ack-time= # Time between offset commits when ackMode is "TIME" or "COUNT_TIME".
spring.kafka.listener.client-id= # Prefix for the listener's consumer client.id property.
spring.kafka.listener.concurrency= # Number of threads to run in the listener containers.
spring.kafka.listener.idle-event-interval= # Time between publishing idle consumer events (no data
 received).
spring.kafka.listener.log-container-config= # Whether to log the container configuration during
 initialization (INFO level).
spring.kafka.listener.monitor-interval= # Time between checks for non-responsive consumers. If a
 duration suffix is not specified, seconds will be used.
spring.kafka.listener.no-poll-threshold= # Multiplier applied to "pollTimeout" to determine if a
 consumer is non-responsive.
spring.kafka.listener.poll-timeout= # Timeout to use when polling the consumer.
spring.kafka.listener.type=single # Listener type.
spring.kafka.producer.acks= # Number of acknowledgments the producer requires the leader to have
 received before considering a request complete.
spring.kafka.producer.batch-size= # Number of records to batch before sending.
spring.kafka.producer.bootstrap-servers= # Comma-delimited list of host:port pairs to use for
 establishing the initial connection to the Kafka cluster.
spring.kafka.producer.buffer-memory= # Total bytes of memory the producer can use to buffer records
 waiting to be sent to the server.
spring.kafka.producer.client-id= # ID to pass to the server when making requests. Used for server-side
 logging.
spring.kafka.producer.compression-type= # Compression type for all data generated by the producer.
spring.kafka.producer.key-serializer= # Serializer class for keys.
spring.kafka.producer.properties.*= # Additional producer-specific properties used to configure the
 client.
spring.kafka.producer.retries= # When greater than zero, enables retrying of failed sends.
spring.kafka.producer.ssl.key-password= # Password of the private key in the key store file.
spring.kafka.producer.ssl.keystore-location= # Location of the key store file.
spring.kafka.producer.ssl.keystore-password= # Store password for the key store file.
spring.kafka.producer.ssl.truststore-location= # Location of the trust store file.
spring.kafka.producer.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.producer.transaction-id-prefix= # When non empty, enables transaction support for producer.
spring.kafka.producer.value-serializer= # Serializer class for values.
spring.kafka.properties.*= # Additional properties, common to producers and consumers, used to configure
 the client.
spring.kafka.ssl.key-password= # Password of the private key in the key store file.
spring.kafka.ssl.keystore-location= # Location of the key store file.
spring.kafka.ssl.keystore-password= # Store password for the key store file.
spring.kafka.ssl.truststore-location= # Location of the trust store file.
spring.kafka.ssl.truststore-password= # Store password for the trust store file.
spring.kafka.template.default-topic= # Default topic to which messages are sent.

# RABBIT (RabbitProperties)
spring.rabbitmq.addresses= # Comma-separated list of addresses to which the client should connect.
spring.rabbitmq.cache.channel.checkout-timeout= # Duration to wait to obtain a channel if the cache size
 has been reached.
spring.rabbitmq.cache.channel.size= # Number of channels to retain in the cache.
spring.rabbitmq.cache.connection.mode=channel # Connection factory cache mode.
spring.rabbitmq.cache.connection.size= # Number of connections to cache.
spring.rabbitmq.connection-timeout= # Connection timeout. Set it to zero to wait forever.
spring.rabbitmq.dynamic=true # Whether to create an AmqpAdmin bean.
spring.rabbitmq.host=localhost # RabbitMQ host.
spring.rabbitmq.listener.direct.acknowledge-mode= # Acknowledge mode of container.
spring.rabbitmq.listener.direct.auto-startup=true # Whether to start the container automatically on
 startup.
spring.rabbitmq.listener.direct.consumers-per-queue= # Number of consumers per queue.
spring.rabbitmq.listener.direct.default-requeue-rejected= # Whether rejected deliveries are re-queued by
 default. Defaults to true.
spring.rabbitmq.listener.direct.idle-event-interval= # How often idle container events should be
 published.
spring.rabbitmq.listener.direct.prefetch= # Number of messages to be handled in a single request. It
 should be greater than or equal to the transaction size (if used).
spring.rabbitmq.listener.direct.retry.enabled=false # Whether publishing retries are enabled.
spring.rabbitmq.listener.direct.retry.initial-interval=1000ms # Interval between the first and second
 attempt to publish or deliver a message.
spring.rabbitmq.listener.direct.retry.max-attempts=3 # Maximum number of attempts to publish or deliver
 a message.
spring.rabbitmq.listener.direct.retry.max-interval=10000ms # Maximum interval between attempts.
spring.rabbitmq.listener.direct.retry.multiplier=1 # Multiplier to apply to the previous retry interval.
spring.rabbitmq.listener.direct.retry.stateless=true # Whether retries are stateless or stateful.
spring.rabbitmq.listener.simple.acknowledge-mode= # Acknowledge mode of container.
spring.rabbitmq.listener.simple.auto-startup=true # Whether to start the container automatically on
 startup.
spring.rabbitmq.listener.simple.concurrency= # Minimum number of listener invoker threads.
spring.rabbitmq.listener.simple.default-requeue-rejected= # Whether to re-queue delivery failures.
spring.rabbitmq.listener.simple.idle-event-interval= # How often idle container events should be
 published.
spring.rabbitmq.listener.simple.max-concurrency= # Maximum number of listener invoker.
spring.rabbitmq.listener.simple.prefetch= # Number of messages to be handled in a single request. It
 should be greater than or equal to the transaction size (if used).
spring.rabbitmq.listener.simple.retry.enabled=false # Whether publishing retries are enabled.
spring.rabbitmq.listener.simple.retry.initial-interval=1000 # Interval, in milliseconds, between the
 first and second attempt to deliver a message.
spring.rabbitmq.listener.simple.retry.max-attempts=3 # Maximum number of attempts to deliver a message.
spring.rabbitmq.listener.simple.retry.max-interval=10000 # Maximum interval, in milliseconds, between
 attempts.
spring.rabbitmq.listener.simple.retry.multiplier=1.0 # Multiplier to apply to the previous delivery
 retry interval.
spring.rabbitmq.listener.simple.retry.stateless=true # Whether or not retry is stateless or stateful.
spring.rabbitmq.listener.simple.transaction-size= # Number of messages to be processed in a transaction.
 That is, the number of messages between acks. For best results, it should be less than or equal to the
 prefetch count.
spring.rabbitmq.listener.type=simple # Listener container type.
spring.rabbitmq.password=guest # Login to authenticate against the broker.
spring.rabbitmq.port=5672 # RabbitMQ port.
spring.rabbitmq.publisher-confirms=false # Whether to enable publisher confirms.
spring.rabbitmq.publisher-returns=false # Whether to enable publisher returns.
spring.rabbitmq.requested-heartbeat= # Requested heartbeat timeout; zero for none. If a duration suffix
 is not specified, seconds will be used.
spring.rabbitmq.ssl.enabled=false # Whether to enable SSL support.
spring.rabbitmq.ssl.key-store= # Path to the key store that holds the SSL certificate.
spring.rabbitmq.ssl.key-store-password= # Password used to access the key store.
spring.rabbitmq.ssl.key-store-type=PKCS12 # Key store type.
spring.rabbitmq.ssl.trust-store= # Trust store that holds SSL certificates.
spring.rabbitmq.ssl.trust-store-password= # Password used to access the trust store.
spring.rabbitmq.ssl.trust-store-type=JKS # Trust store type.
spring.rabbitmq.ssl.algorithm= # SSL algorithm to use. By default, configured by the Rabbit client
 library.
spring.rabbitmq.template.exchange= # Name of the default exchange to use for send operations.
spring.rabbitmq.template.mandatory=false # Whether to enable mandatory messages.
spring.rabbitmq.template.receive-timeout=0 # Timeout for `receive()` methods.
spring.rabbitmq.template.reply-timeout=5000 # Timeout for `sendAndReceive()` methods.
spring.rabbitmq.template.retry.enabled=false # Whether to enable retries in the `RabbitTemplate`.
spring.rabbitmq.template.retry.initial-interval=1000 # Interval, in milliseconds, between the first and
 second attempt to publish a message.
spring.rabbitmq.template.retry.max-attempts=3 # Maximum number of attempts to publish a message.
spring.rabbitmq.template.retry.max-interval=10000 # Maximum number of attempts to publish a message.
spring.rabbitmq.template.retry.multiplier=1.0 # Multiplier to apply to the previous publishing retry
 interval.
spring.rabbitmq.template.routing-key= # Value of a default routing key to use for send operations.
spring.rabbitmq.username=guest # Login user to authenticate to the broker.
spring.rabbitmq.virtual-host= # Virtual host to use when connecting to the broker.


# ----------------------------------------
# ACTUATOR PROPERTIES
# ----------------------------------------

# MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.server.add-application-context-header=false # Add the "X-Application-Context" HTTP header in
 each response. Requires a custom management.server.port.
management.server.address= # Network address that to which the management endpoints should bind.
 Requires a custom management.server.port.
management.server.port= # Management endpoint HTTP port. Uses the same port as the application by
 default. Configure a different port to use management-specific SSL.
management.server.servlet.context-path= # Management endpoint context-path. For instance, `/management`.
 Requires a custom management.server.port.
management.server.ssl.ciphers= # Supported SSL ciphers. Requires a custom management.port.
management.server.ssl.client-auth= # Whether client authentication is wanted ("want") or needed
 ("need"). Requires a trust store. Requires a custom management.server.port.
management.server.ssl.enabled= # Whether to enable SSL support. Requires a custom
 management.server.port.
management.server.ssl.enabled-protocols= # Enabled SSL protocols. Requires a custom
 management.server.port.
management.server.ssl.key-alias= # Alias that identifies the key in the key store. Requires a custom
 management.server.port.
management.server.ssl.key-password= # Password used to access the key in the key store. Requires a
 custom management.server.port.
management.server.ssl.key-store= # Path to the key store that holds the SSL certificate (typically a jks
 file). Requires a custom management.server.port.
management.server.ssl.key-store-password= # Password used to access the key store. Requires a custom
 management.server.port.
management.server.ssl.key-store-provider= # Provider for the key store. Requires a custom
 management.server.port.
management.server.ssl.key-store-type= # Type of the key store. Requires a custom management.server.port.
management.server.ssl.protocol=TLS # SSL protocol to use. Requires a custom management.server.port.
management.server.ssl.trust-store= # Trust store that holds SSL certificates. Requires a custom
 management.server.port.
management.server.ssl.trust-store-password= # Password used to access the trust store. Requires a custom
 management.server.port.
management.server.ssl.trust-store-provider= # Provider for the trust store. Requires a custom
 management.server.port.
management.server.ssl.trust-store-type= # Type of the trust store. Requires a custom
 management.server.port.

# CLOUDFOUNDRY
management.cloudfoundry.enabled=true # Whether to enable extended Cloud Foundry actuator endpoints.
management.cloudfoundry.skip-ssl-validation=false # Whether to skip SSL verification for Cloud Foundry
 actuator endpoint security calls.

# ENDPOINTS GENERAL CONFIGURATION
management.endpoints.enabled-by-default= # Enable or disable all endpoints by default.

# ENDPOINTS JMX CONFIGURATION (JmxEndpointProperties)
management.endpoints.jmx.expose=* # Endpoint IDs that should be exposed or '*' for all.
management.endpoints.jmx.exclude= # Endpoint IDs that should be excluded.
management.endpoints.jmx.domain=org.springframework.boot # Endpoints JMX domain name. Fallback to
 'spring.jmx.default-domain' if set.
management.endpoints.jmx.static-names=false # Additional static properties to append to all ObjectNames
 of MBeans representing Endpoints.
management.endpoints.jmx.unique-names=false # Whether to ensure that ObjectNames are modified in case of
 conflict.

# ENDPOINTS WEB CONFIGURATION (WebEndpointProperties)
management.endpoints.web.expose=info,health # Endpoint IDs that should be exposed or '*' for all.
management.endpoints.web.exclude= # Endpoint IDs that should be excluded.
management.endpoints.web.base-path=/actuator # Base path for Web endpoints. Relative to
 server.servlet.context-path or management.server.servlet.context-path if management.server.port is
 configured.
management.endpoints.web.path-mapping= # Mapping between endpoint IDs and the path that should expose
 them.

# ENDPOINTS CORS CONFIGURATION (CorsEndpointProperties)
management.endpoints.web.cors.allow-credentials= # Whether credentials are supported. When not set,
 credentials are not supported.
management.endpoints.web.cors.allowed-headers= # Comma-separated list of headers to allow in a request.
 '*' allows all headers.
management.endpoints.web.cors.allowed-methods= # Comma-separated list of methods to allow. '*' allows
 all methods. When not set, defaults to GET.
management.endpoints.web.cors.allowed-origins= # Comma-separated list of origins to allow. '*' allows
 all origins. When not set, CORS support is disabled.
management.endpoints.web.cors.exposed-headers= # Comma-separated list of headers to include in a
 response.
management.endpoints.web.cors.max-age=1800 # How long the response from a pre-flight request can be
 cached by clients. If a duration suffix is not specified, seconds will be used.

# AUDIT EVENTS ENDPOINT (AuditEventsEndpoint)
management.endpoint.auditevents.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.auditevents.enabled= # Whether to enable the auditevents endpoint.

# BEANS ENDPOINT (BeansEndpoint)
management.endpoint.beans.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.beans.enabled= # Whether to enable the beans endpoint.
```

```
# CONDITIONS REPORT ENDPOINT (ConditionsReportEndpoint)
management.endpoint.conditions.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.conditions.enabled= # Whether to enable the conditions endpoint.

# CONFIGURATION PROPERTIES REPORT ENDPOINT
 (ConfigurationPropertiesReportEndpoint, ConfigurationPropertiesReportEndpointProperties)
management.endpoint.configprops.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.configprops.enabled= # Whether to enable the configprops endpoint.
management.endpoint.configprops.keys-to-
sanitize=password,secret,key,token,.*credentials.*,vcap_services # Keys that should be sanitized. Keys
 can be simple strings that the property ends with or regular expressions.

# ENVIRONMENT ENDPOINT (EnvironmentEndpoint, EnvironmentEndpointProperties)
management.endpoint.env.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.env.enabled= # Whether to enable the env endpoint.
management.endpoint.env.keys-to-sanitize=password,secret,key,token,.*credentials.*,vcap_services #
 Keys that should be sanitized. Keys can be simple strings that the property ends with or regular
 expressions.

# FLYWAY ENDPOINT (FlywayEndpoint)
management.endpoint.flyway.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.flyway.enabled= # Whether to enable the flyway endpoint.

# HEALTH ENDPOINT (HealthEndpoint, HealthEndpointProperties)
management.endpoint.health.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.health.enabled= # Whether to enable the health endpoint.
management.endpoint.health.show-details=false # Whether to show full health details instead of just the
 status when exposed over a potentially insecure connection.

# HEAP DUMP ENDPOINT (HeapDumpWebEndpoint)
management.endpoint.heapdump.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.heapdump.enabled= # Whether to enable the heapdump endpoint.

# HTTP TRACE ENDPOINT (HttpTraceEndpoint)
management.endpoint.httptrace.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.httptrace.enabled= # Whether to enable the HTTP trace endpoint.

# INFO ENDPOINT (InfoEndpoint)
info= # Arbitrary properties to add to the info endpoint.
management.endpoint.info.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.info.enabled=true # Whether to enable the info endpoint.

# JOLOKIA ENDPOINT (JolokiaProperties)
management.endpoint.jolokia.config.*= # Jolokia settings. See the Jolokia manual for details.
management.endpoint.jolokia.enabled=true # Whether to enable Jolokia.

# LIQUIBASE ENDPOINT (LiquibaseEndpoint)
management.endpoint.liquibase.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.liquibase.enabled= # Whether to enable the liquibase endpoint.

# LOG FILE ENDPOINT (LogFileWebEndpoint, LogFileWebEndpointProperties)
management.endpoint.logfile.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.logfile.enabled= # Whether to enable the logfile endpoint.
management.endpoint.logfile.external-file= # External Logfile to be accessed. Can be used if the logfile
 is written by output redirect and not by the logging system itself.

# LOGGERS ENDPOINT (LoggersEndpoint)
management.endpoint.loggers.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.loggers.enabled= # Whether to enable the loggers endpoint.

# REQUEST MAPPING ENDPOINT  (MappingsEndpoint)
management.endpoint.mappings.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.mappings.enabled= # Whether to enable the mappings endpoint.

# METRICS ENDPOINT (MetricsEndpoint)
management.endpoint.metrics.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.metrics.enabled= # Whether to enable the metrics endpoint.

# PROMETHEUS ENDPOINT (PrometheusScrapeEndpoint)
management.endpoint.prometheus.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.prometheus.enabled= # Whether to enable the metrics endpoint.

# SCHEDULED TASKS ENDPOINT (ScheduledTasksEndpoint)
management.endpoint.scheduledtasks.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.scheduledtasks.enabled= # Whether to enable the scheduled tasks endpoint.

# SESSIONS ENDPOINT (SessionsEndpoint)
management.endpoint.sessions.enabled= # Whether to enable the sessions endpoint.

# SHUTDOWN ENDPOINT (ShutdownEndpoint)
management.endpoint.shutdown.enabled=false # Whether to enable the shutdown endpoint.

# THREAD DUMP ENDPOINT (ThreadDumpEndpoint)
management.endpoint.threaddump.cache.time-to-live=0ms # Maximum time that a response can be cached.
management.endpoint.threaddump.enabled= # Whether to enable the threaddump endpoint.

# HEALTH INDICATORS
management.health.db.enabled=true # Whether to enable database health check.
management.health.cassandra.enabled=true # Whether to enable Cassandra health check.
management.health.couchbase.enabled=true # Whether to enable Couchbase health check.
management.health.defaults.enabled=true # Whether to enable default health indicators.
management.health.diskspace.enabled=true # Whether to enable disk space health check.
management.health.diskspace.path= # Path used to compute the available disk space.
management.health.diskspace.threshold=0 # Minimum disk space, in bytes, that should be available.
management.health.elasticsearch.enabled=true # Whether to enable Elasticsearch health check.
management.health.elasticsearch.indices= # Comma-separated index names.
management.health.elasticsearch.response-timeout=100ms # The time to wait for a response from the
 cluster.
management.health.influxdb.enabled=true # Whether to enable InfluxDB health check.
management.health.jms.enabled=true # Whether to enable JMS health check.
management.health.ldap.enabled=true # Whether to enable LDAP health check.
management.health.mail.enabled=true # Whether to enable Mail health check.
management.health.mongo.enabled=true # Whether to enable MongoDB health check.
management.health.neo4j.enabled=true # Whether to enable Neo4j health check.
management.health.rabbit.enabled=true # Whether to enable RabbitMQ health check.
management.health.redis.enabled=true # Whether to enable Redis health check.
management.health.solr.enabled=true # Whether to enable Solr health check.
management.health.status.http-mapping= # Mapping of health statuses to HTTP status codes. By default,
 registered health statuses map to sensible defaults (for example, UP maps to 200).
management.health.status.order=DOWN, OUT_OF_SERVICE, UP, UNKNOWN # Comma-separated list of health
 statuses in order of severity.

# HTTP TRACING (HttpTraceProperties)
management.httptrace.enabled=true # Whether to enable HTTP request-response tracing.
management.httptrace.include=request-headers,response-headers,cookies,errors # Items to be included in
 the trace.

# INFO CONTRIBUTORS (InfoContributorProperties)
management.info.build.enabled=true # Whether to enable build info.
management.info.defaults.enabled=true # Whether to enable default info contributors.
management.info.env.enabled=true # Whether to enable environment info.
management.info.git.enabled=true # Whether to enable git info.
management.info.git.mode=simple # Mode to use to expose git information.

# METRICS
management.metrics.binders.jvm.enabled=true # Whether to enable JVM metrics.
management.metrics.binders.logback.enabled=true # Whether to enable Logback metrics.
management.metrics.binders.processor.enabled=true # Whether to enable processor metrics.
management.metrics.binders.uptime.enabled=true # Whether to enable uptime metrics.
management.metrics.cache.metric-name=cache # Name of the metric for cache usage.
management.metrics.cache.instrument=true # Instrument all available caches.
management.metrics.export.atlas.batch-size= # Number of measurements per request to use for the backend.
 If more measurements are found, then multiple requests will be made.
management.metrics.export.atlas.config-refresh-frequency= # Frequency for refreshing config settings
 from the LWC service.
management.metrics.export.atlas.config-time-to-live= # Time to live for subscriptions from the LWC
 service.
management.metrics.export.atlas.config-uri= # URI for the Atlas LWC endpoint to retrieve current
 subscriptions.
management.metrics.export.atlas.connect-timeout= # Connection timeout for requests to the backend.
management.metrics.export.atlas.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.atlas.eval-uri= # URI for the Atlas LWC endpoint to evaluate the data for a
 subscription.
management.metrics.export.atlas.lwc-enabled= # Enable streaming to Atlas LWC.
management.metrics.export.atlas.meter-time-to-live= # Time to live for meters that do not have any
 activity. After this period the meter will be considered expired and will not get reported.
management.metrics.export.atlas.num-threads= # Number of threads to use with the metrics publishing
 scheduler.
management.metrics.export.atlas.read-timeout= # Read timeout for requests to the backend.
management.metrics.export.atlas.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.atlas.uri= # URI of the Atlas server.
management.metrics.export.datadog.api-key= # Datadog API key.
management.metrics.export.datadog.application-key= # Datadog application key.
management.metrics.export.datadog.batch-size= # Number of measurements per request to use for the
 backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.datadog.connect-timeout= # Connection timeout for requests to the backend.
management.metrics.export.datadog.descriptions= # Whether to publish descriptions metadata to Datadog.
 Turn this off to minimize the amount of metadata sent.
management.metrics.export.datadog.enabled=true # Whether exporting of metrics to this backend is
 enabled.
management.metrics.export.datadog.host-tag= # Tag that will be mapped to "host" when shipping metrics to
 Datadog. Can be omitted if host should be omitted on publishing.
management.metrics.export.datadog.num-threads= # Number of threads to use with the metrics publishing
 scheduler.
management.metrics.export.datadog.read-timeout= # Read timeout for requests to the backend.
management.metrics.export.datadog.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.datadog.uri= # URI to ship metrics to. If you need to publish metrics to an
 internal proxy en-route to Datadog, you can define the location of the proxy with this.
management.metrics.export.ganglia.addressing-mode= # UDP addressing mode, either unicast or multicast.
management.metrics.export.ganglia.duration-units= # Base time unit used to report durations.
management.metrics.export.ganglia.enabled=true # Whether exporting of metrics to Ganglia is enabled.
management.metrics.export.ganglia.host= # Host of the Ganglia server to receive exported metrics.
management.metrics.export.ganglia.port= # Port of the Ganglia server to receive exported metrics.
management.metrics.export.ganglia.protocol-version= # Ganglia protocol version. Must be either 3.1 or
 3.0.
management.metrics.export.ganglia.rate-units= # Base time unit used to report rates.
management.metrics.export.ganglia.step= # Step size (i.e. reporting frequency) to use.
management.metrics.export.ganglia.time-to-live= # Time to live for metrics on Ganglia.
management.metrics.export.graphite.duration-units= # Base time unit used to report durations.
management.metrics.export.graphite.enabled=true # Whether exporting of metrics to Graphite is enabled.
management.metrics.export.graphite.host= # Host of the Graphite server to receive exported metrics.
management.metrics.export.graphite.port= # Port of the Graphite server to receive exported metrics.
management.metrics.export.graphite.protocol= # Protocol to use while shipping data to Graphite.
management.metrics.export.graphite.rate-units= # Base time unit used to report rates.
management.metrics.export.graphite.step= # Step size (i.e. reporting frequency) to use.
management.metrics.export.graphite.tags-as-prefix= # For the default naming convention, turn the
 specified tag keys into part of the metric prefix.
management.metrics.export.influx.batch-size= # Number of measurements per request to use for the
 backend. If more measurements are found, then multiple requests will be made.
management.metrics.export.influx.compressed= # Enable GZIP compression of metrics batches published to
 Influx.
management.metrics.export.influx.connect-timeout= # Connection timeout for requests to the backend.
management.metrics.export.influx.consistency= # Write consistency for each point.
management.metrics.export.influx.db= # Tag that will be mapped to "host" when shipping metrics to
 Influx. Can be omitted if host should be omitted on publishing.
management.metrics.export.influx.enabled=true # Whether exporting of metrics to this backend is enabled.
management.metrics.export.influx.num-threads= # Number of threads to use with the metrics publishing
 scheduler.
management.metrics.export.influx.password= # Login password of the Influx server.
management.metrics.export.influx.read-timeout= # Read timeout for requests to the backend.
management.metrics.export.influx.retention-policy= # Retention policy to use (Influx writes to the
 DEFAULT retention policy if one is not specified).
management.metrics.export.influx.step=1m # Step size (i.e. reporting frequency) to use.
management.metrics.export.influx.uri= # URI of the Influx server.
management.metrics.export.influx.user-name= # Login user of the Influx server.
management.metrics.export.jmx.enabled=true # Whether exporting of metrics to JMX is enabled.
management.metrics.export.jmx.step= # Step size (i.e. reporting frequency) to use.
management.metrics.export.prometheus.descriptions= # Enable publishing descriptions as part of the
 scrape payload to Prometheus. Turn this off to minimize the amount of data sent on each scrape.
management.metrics.export.prometheus.enabled=true # Whether exporting of metrics to Prometheus is
 enabled.
management.metrics.export.prometheus.step= # Step size (i.e. reporting frequency) to use.
management.metrics.export.simple.enabled=true # Whether exporting of metrics to a simple in-memory store
 is enabled.
management.metrics.export.simple.mode=cumulative # Counting mode.
management.metrics.export.simple.step=10s # Step size (i.e. reporting frequency) to use.
management.metrics.export.statsd.enabled=true # Export metrics to StatsD.
management.metrics.export.statsd.flavor=datadog # StatsD line protocol to use.
management.metrics.export.statsd.host=localhost # Host of the StatsD server to receive exported metrics.
management.metrics.export.statsd.max-packet-length=1400 # Total length of a single payload should be
 kept within your network's MTU.
management.metrics.export.statsd.polling-frequency=10s # How often gauges will be polled. When a gauge
 is polled, its value is recalculated and if the value has changed, it is sent to the StatsD server.
management.metrics.export.statsd.port=8125 # Port of the StatsD server to receive exported metrics.
management.metrics.export.statsd.queue-size=2147483647 # Maximum size of the queue of items waiting to
 be sent to the StatsD server.
management.metrics.jdbc.instrument=true # Instrument all available data sources.
management.metrics.jdbc.metric-name=data.source # Name of the metric for data source usage.
management.metrics.rabbitmq.instrument=true # Instrument all available connection factories.
management.metrics.rabbitmq.metric-name=rabbitmq # Name of the metric for RabbitMQ usage.
management.metrics.use-global-registry=true # Whether auto-configured MeterRegistry implementations
 should be bound to the global static registry on Metrics.
management.metrics.web.client.record-request-percentiles=false # Whether instrumented requests record
 percentiles histogram buckets by default.
management.metrics.web.client.requests-metric-name=http.client.requests # Name of the metric for sent
 requests.
management.metrics.web.server.auto-time-requests=true # Whether requests handled by Spring MVC or
 WebFlux should be automatically timed.
management.metrics.web.server.record-request-percentiles=false # Whether instrumented requests record
 percentiles histogram buckets by default.
management.metrics.web.server.requests-metric-name=http.server.requests # Name of the metric for
 received requests.


# ----------------------------------------
# DEVTOOLS PROPERTIES
# ----------------------------------------

# DEVTOOLS (DevToolsProperties)
spring.devtools.livereload.enabled=true # Whether to enable a livereload.com-compatible server.
spring.devtools.livereload.port=35729 # Server port.
spring.devtools.restart.additional-exclude= # Additional patterns that should be excluded from
 triggering a full restart.
spring.devtools.restart.additional-paths= # Additional paths to watch for changes.
spring.devtools.restart.enabled=true # Enable automatic restart.
spring.devtools.restart.exclude=META-INF/maven/**,META-INF/resources/**,resources/**,static/**,public/
**,templates/**,**/*Test.class,**/*Tests.class,git.properties # Patterns that should be excluded from
 triggering a full restart.
spring.devtools.restart.log-condition-evaluation-delta=true # Whether to log the condition evaluation
 delta upon restart.
spring.devtools.restart.poll-interval=1s # Amount of time to wait between polling for classpath changes.
spring.devtools.restart.quiet-period=400ms # Amount of quiet time required without any classpath changes
 before a restart is triggered.
spring.devtools.restart.trigger-file= # Name of a specific file that, when changed, triggers the restart
 check. If not specified, any classpath file change triggers the restart.

# REMOTE DEVTOOLS (RemoteDevToolsProperties)
spring.devtools.remote.context-path=/.~~spring-boot!~ # Context path used to handle the remote
 connection.
spring.devtools.remote.proxy.host= # The host of the proxy to use to connect to the remote application.
spring.devtools.remote.proxy.port= # The port of the proxy to use to connect to the remote application.
spring.devtools.remote.restart.enabled=true # Whether to enable remote restart.
spring.devtools.remote.secret= # A shared secret required to establish a connection (required to enable
 remote support).
spring.devtools.remote.secret-header-name=X-AUTH-TOKEN # HTTP header used to transfer the shared secret.


# ----------------------------------------
# TESTING PROPERTIES
# ----------------------------------------

spring.test.database.replace=any # Type of existing DataSource to replace.
spring.test.mockmvc.print=default # MVC Print option.
```

Appendix B. Configuration Metadata

Spring Boot jars include metadata files that provide details of all supported configuration properties. The files are designed to let IDE developers offer contextual help and “code completion” as users are working with `application.properties` or `application.yml` files.

The majority of the metadata file is generated automatically at compile time by processing all items annotated with `@ConfigurationProperties`. However, it is possible to write part of the metadata manually for corner cases or more advanced use cases.

B.1 Metadata Format

Configuration metadata files are located inside jars under `META-INF/spring-configuration-metadata.json` They use a simple JSON format with items categorized under either “groups” or “properties” and additional values hints categorized under "hints", as shown in the following example:

```
{"groups": [
 {
  "name": "server",
  "type": "org.springframework.boot.autoconfigure.web.ServerProperties",
  "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
 },
 {
  "name": "spring.jpa.hibernate",
  "type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate",
  "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
  "sourceMethod": "getHibernate()"
 }
 ...
],"properties": [
 {
  "name": "server.port",
  "type": "java.lang.Integer",
  "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"
 },
 {
  "name": "server.servlet.path",
  "type": "java.lang.String",
  "sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties",
  "defaultValue": "/"
 },
 {
    "name": "spring.jpa.hibernate.ddl-auto",
    "type": "java.lang.String",
    "description": "DDL mode. This is actually a shortcut for the \"hibernate.hbm2ddl.auto\" property.",
    "sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"
 }
 ...
],"hints": [
 {
  "name": "spring.jpa.hibernate.ddl-auto",
  "values": [
   {
    "value": "none",
    "description": "Disable DDL handling."
   },
   {
    "value": "validate",
    "description": "Validate the schema, make no changes to the database."
   },
   {
    "value": "update",
    "description": "Update the schema if necessary."
   },
   {
    "value": "create",
    "description": "Create the schema and destroy previous data."
   },
   {
    "value": "create-drop",
    "description": "Create and then destroy the schema at the end of the session."
   }
  ]
 }
]}
```

Each “property” is a configuration item that the user specifies with a given value. For example, `server.port` and `server.servlet.path` might be specified in `application.properties`, as follows:

```
server.port=9090
server.servlet.path=/home
```

The “groups” are higher level items that do not themselves specify a value but instead provide a contextual grouping for properties. For example, the `server.port` and `server.servlet.path` properties are part of the `server` group.

> **Note**
>
> It is not required that every “property” has a “group”. Some properties might exist in their own right.

Finally, “hints” are additional information used to assist the user in configuring a given property. For example, when a developer is configuring the `spring.jpa.hibernate.ddl-auto` property, a tool can use the hints to offer some auto-completion help for the `none`, `validate`, `update`, `create`, and `create-drop` values.

Group Attributes

The JSON object contained in the `groups` array can contain the attributes shown in the following table:

Name	Type	Purpose
`name`	String	The full name of the group. This attribute is mandatory.
`type`	String	The class name of the data type of the group. For example, if the group were based on a class annotated with `@ConfigurationProperties`, the attribute would contain the fully qualified name of that class. If it were based on a `@Bean` method, it would be the return type of that method. If the type is not known, the attribute may be omitted.
`description`	String	A short description of the group that can be displayed to users. If not description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this group. For example, if the group were based on a `@Bean` method annotated

Name	Type	Purpose
		with `@ConfigurationProperties`, this attribute would contain the fully qualified name of the `@Configuration` class that contains the method. If the source type is not known, the attribute may be omitted.
`sourceMethod`	String	The full name of the method (include parenthesis and argument types) that contributed this group (for example, the name of a `@ConfigurationProperties` annotated `@Bean` method). If the source method is not known, it may be omitted.

Property Attributes

The JSON object contained in the `properties` array can contain the attributes described in the following table:

Name	Type	Purpose
`name`	String	The full name of the property. Names are in lower-case period-separated form (for example, `server.servlet.path`). This attribute is mandatory.
`type`	String	The full signature of the data type of the property (for example, `java.lang.String`) but also a full generic type (such as `java.util.Map<java.util.String,acme.MyEnum>`). You can use this attribute to guide the user as to the types of values that they can enter. For consistency, the type of a primitive is specified by using its wrapper counterpart (for example, `boolean` becomes `java.lang.Boolean`). Note that this class may be a complex type that gets converted from a `String` as values are bound. If the type is not known, it may be omitted.
`description`	String	A short description of the group that can be displayed to users. If no description is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`sourceType`	String	The class name of the source that contributed this property. For example, if the property were from a class annotated with `@ConfigurationProperties`, this attribute would contain the fully qualified name of that class. If the source type is unknown, it may be omitted.
`defaultValue`	Object	The default value, which is used if the property is not specified. If the type of the property is an array, it can be an array of value(s). If the default value is unknown, it may be omitted.
`deprecation`	Deprecation	Specify whether the property is deprecated. If the field is not deprecated or if that information is not known, it may be omitted. The next table offers more detail about the `deprecation` attribute.

The JSON object contained in the `deprecation` attribute of each `properties` element can contain the following attributes:

Name	Type	Purpose
`level`	String	The level of deprecation, which can be either `warning` (the default) or `error`. When a property has a `warning` deprecation level, it should still be bound in the environment. However, when it has an `error` deprecation level, the property is no longer managed and is not bound.
`reason`	String	A short description of the reason why the property was deprecated. If no reason is available, it may be omitted. It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).
`replacement`	String	The full name of the property that *replaces* this deprecated property. If there is no replacement for this property, it may be omitted.

Note

Prior to Spring Boot 1.3, a single `deprecated` boolean attribute can be used instead of the `deprecation` element. This is still supported in a deprecated fashion and should no longer be used. If no reason and replacement are available, an empty `deprecation` object should be set.

Deprecation can also be specified declaratively in code by adding the `@DeprecatedConfigurationProperty` annotation to the getter exposing the deprecated property. For instance, assume that the `app.acme.target` property was confusing and was renamed to `app.acme.name`. The following example shows how to handle that situation:

```java
@ConfigurationProperties("app.acme")
public class AcmeProperties {

 private String name;

 public String getName() { ... }

 public void setName(String name) { ... }

 @DeprecatedConfigurationProperty(replacement = "app.acme.name")
 @Deprecated
 public String getTarget() {
  return getName();
 }

 @Deprecated
 public void setTarget(String target) {
  setName(target);
 }
}
```

Note

There is no way to set a `level`. `warning` is always assumed, since code is still handling the property.

The preceding code makes sure that the deprecated property still works (delegating to the `name` property behind the scenes). Once the `getTarget` and `setTarget` methods can be removed from your public API, the automatic deprecation hint in the metadata goes away as well. If you want to keep a hint, adding manual metadata with an `error` deprecation level ensures that users are still informed about that property. Doing so is particularly useful when a `replacement` is provided.

Hint Attributes

The JSON object contained in the `hints` array can contain the attributes shown in the following table:

Name	**Type**	**Purpose**
`name`	String	The full name of the property to which this hint refers. Names are in lower-case period-separated form (such as `server.servlet.path`). If the property refers to a map (such as `system.contexts`), the hint either applies to the *keys* of the map (`system.context.keys`) or the *values* (`system.context.values`) of the map. This attribute is mandatory.
`values`	ValueHint[]	A list of valid values as defined by the `ValueHint` object (described in the next table). Each entry defines the value and may have a description.
`providers`	ValueProvider[]	A list of providers as defined by the `ValueProvider` object (described later in this document). Each entry defines the name of the provider and its parameters, if any.

The JSON object contained in the `values` attribute of each `hint` element can contain the attributes described in the following table:

Name	**Type**	**Purpose**
`value`	Object	A valid value for the element to which the hint refers. If the type of the property is an array, it can also be an array of value(s). This attribute is mandatory.
`description`	String	A short description of the value that can be displayed to users. If no description is available, it may be omitted . It is recommended that descriptions be short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (`.`).

The JSON object contained in the `providers` attribute of each `hint` element can contain the attributes described in the following table:

Name	**Type**	**Purpose**
`name`	String	The name of the provider to use to offer additional content assistance for the element to which the hint refers.

Name	Type	Purpose
`parameters`	JSON object	Any additional parameter that the provider supports (check the documentation of the provider for more details).

Repeated Metadata Items

Objects with the same “property” and “group” name can appear multiple times within a metadata file. For example, you could bind two separate classes to the same prefix, with each having potentially overlapping property names. While the same names appearing in the metadata multiple times should not be common, consumers of metadata should take care to ensure that they support it.

B.2 Providing Manual Hints

To improve the user experience and further assist the user in configuring a given property, you can provide additional metadata that:

- Describes the list of potential values for a property.
- Associates a provider, to attach a well defined semantic to a property, so that a tool can discover the list of potential values based on the project’s context.

Value Hint

The `name` attribute of each hint refers to the `name` of a property. In the initial example shown earlier, we provide five values for the `spring.jpa.hibernate.ddl-auto` property: `none`, `validate`, `update`, `create`, and `create-drop`. Each value may have a description as well.

If your property is of type `Map`, you can provide hints for both the keys and the values (but not for the map itself). The special `.keys` and `.values` suffixes must refer to the keys and the values, respectively.

Assume a `sample.contexts` maps magic `String` values to an integer, as shown in the following example:

```
@ConfigurationProperties("sample")
public class SampleProperties {

 private Map<String,Integer> contexts;
 // getters and setters
}
```

The magic values are (in this example) are `sample1` and `sample2`. In order to offer additional content assistance for the keys, you could add the following JSON to the manual metadata of the module:

```
{"hints": [
 {
  "name": "sample.contexts.keys",
  "values": [
   {
    "value": "sample1"
   },
   {
    "value": "sample2"
   }
  ]
 }
]}
```

Tip

We recommend that you use an `Enum` for those two values instead. If your IDE supports it, this is by far the most effective approach to auto-completion.

Value Providers

Providers are a powerful way to attach semantics to a property. In this section, we define the official providers that you can use for your own hints. However, your favorite IDE may implement some of these or none of them. Also, it could eventually provide its own.

Note

As this is a new feature, IDE vendors must catch up with how it works. Adoption times naturally vary.

The following table summarizes the list of supported providers:

Name	Description
`any`	Permits any additional value to be provided.
`class-reference`	Auto-completes the classes available in the project. Usually constrained by a base class that is specified by the `target` parameter.
`handle-as`	Handles the property as if it were defined by the type defined by the mandatory `target` parameter.
`logger-name`	Auto-completes valid logger names. Typically, package and class names available in the current project can be auto-completed.
`spring-bean-reference`	Auto-completes the available bean names in the current project. Usually constrained by a base class that is specified by the `target` parameter.
`spring-profile-name`	Auto-completes the available Spring profile names in the project.

Tip

Only one provider can be active for a given property, but you can specify several providers if they can all manage the property *in some way*. Make sure to place the most powerful provider first, as the IDE must use the first one in the JSON section that it can handle. If no provider for a given property is supported, no special content assistance is provided, either.

Any

The special **any** provider value permits any additional values to be provided. Regular value validation based on the property type should be applied if this is supported.

This provider is typically used if you have a list of values and any extra values should still be considered as valid.

The following example offers on and `off` as auto-completion values for `system.state`:

```
{"hints": [
 {
  "name": "system.state",
  "values": [
   {
    "value": "on"
   },
   {
    "value": "off"
   }
  ],
  "providers": [
   {
    "name": "any"
   }
  ]
 }
]}
```

Note that, in the preceding example, any other value is also allowed.

Class Reference

The **class-reference** provider auto-completes classes available in the project. This provider supports the following parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the class that should be assignable to the chosen value. Typically used to filter out-non candidate classes. Note that this information can be provided by the type itself by exposing a class with the appropriate upper bound.
`concrete`	`boolean`	true	Specify whether only concrete classes are to be considered as valid candidates.

The following metadata snippet corresponds to the standard `server.servlet.jsp.class-name` property that defines the `JspServlet` class name to use:

```
{"hints": [
 {
  "name": "server.servlet.jsp.class-name",
  "providers": [
   {
    "name": "class-reference",
    "parameters": {
     "target": "javax.servlet.http.HttpServlet"
    }
   }
  ]
 }
]}
```

Handle As

The **handle-as** provider lets you substitute the type of the property to a more high-level type. This typically happens when the property has a `java.lang.String` type, because you do not want your configuration classes to rely on classes that may not be on the classpath. This provider supports the following parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the type to consider for the property. This parameter is mandatory.

The following types can be used:

- Any `java.lang.Enum`: Lists the possible values for the property. (We recommend defining the property with the `Enum` type, as no further hint should be required for the IDE to auto-complete the values.)
- `java.nio.charset.Charset`: Supports auto-completion of charset/encoding values (such as `UTF-8`)
- `java.util.Locale`: auto-completion of locales (such as `en_US`)
- `org.springframework.util.MimeType`: Supports auto-completion of content type values (such as `text/plain`)
- `org.springframework.core.io.Resource`: Supports auto-completion of Spring's Resource abstraction to refer to a file on the filesystem or on the classpath. (such as `classpath:/sample.properties`)

Tip

If multiple values can be provided, use a `Collection` or *Array* type to teach the IDE about it.

The following metadata snippet corresponds to the standard `spring.liquibase.change-log` property that defines the path to the changelog to use. It is actually used internally as a `org.springframework.core.io.Resource` but cannot be exposed as such, because we need to keep the original String value to pass it to the Liquibase API.

```
{"hints": [
 {
  "name": "spring.liquibase.change-log",
  "providers": [
   {
    "name": "handle-as",
    "parameters": {
     "target": "org.springframework.core.io.Resource"
    }
   }
  ]
 }
]}
```

Logger Name

The **logger-name** provider auto-completes valid logger names. Typically, package and class names available in the current project can be auto-completed. Specific frameworks may have extra magic logger names that can be supported as well.

Since a logger name can be any arbitrary name, this provider should allow any value but could highlight valid package and class names that are not available in the project's classpath.

The following metadata snippet corresponds to the standard `logging.level` property. Keys are *logger names*, and values correspond to the standard log levels or any custom level.

```
{"hints": [
 {
  "name": "logging.level.keys",
  "values": [
   {
    "value": "root",
    "description": "Root logger used to assign the default logging level."
   }
  ],
  "providers": [
   {
    "name": "logger-name"
   }
  ]
 },
 {
  "name": "logging.level.values",
  "values": [
   {
    "value": "trace"
   },
   {
    "value": "debug"
   },
   {
    "value": "info"
   },
   {
    "value": "warn"
   },
   {
    "value": "error"
   },
   {
    "value": "fatal"
   },
   {
    "value": "off"
   }

  ],
  "providers": [
   {
    "name": "any"
   }
  ]
 }
]}
```

Spring Bean Reference

The **spring-bean-reference** provider auto-completes the beans that are defined in the configuration of the current project. This provider supports the following parameters:

Parameter	Type	Default value	Description
`target`	`String` (`Class`)	*none*	The fully qualified name of the bean class that should be assignable to the candidate. Typically used to filter out non-candidate beans.

The following metadata snippet corresponds to the standard `spring.jmx.server` property that defines the name of the `MBeanServer` bean to use:

```
{"hints": [
 {
  "name": "spring.jmx.server",
  "providers": [
   {
    "name": "spring-bean-reference",
    "parameters": {
     "target": "javax.management.MBeanServer"
    }
   }
  ]
 }
]}
```

> **Note**
>
> The binder is not aware of the metadata. If you provide that hint, you still need to transform the bean name into an actual Bean reference using by the `ApplicationContext`.

Spring Profile Name

The **spring-profile-name** provider auto-completes the Spring profiles that are defined in the configuration of the current project.

The following metadata snippet corresponds to the standard `spring.profiles.active` property that defines the name of the Spring profile(s) to enable:

```
{"hints": [
 {
  "name": "spring.profiles.active",
  "providers": [
   {
    "name": "spring-profile-name"
   }
  ]
 }
]}
```

B.3 Generating Your Own Metadata by Using the Annotation Processor

You can easily generate your own configuration metadata file from items annotated with `@ConfigurationProperties` by using the `spring-boot-configuration-processor` jar. The jar includes a Java annotation processor which is invoked as your project is compiled. To use the processor, include `spring-boot-configuration-processor` as an optional dependency. For example, with Maven, you can add:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-configuration-processor</artifactId>
 <optional>true</optional>
</dependency>
```

With Gradle, you can use the propdeps-plugin and specify the following dependency:

```
dependencies {
 optional "org.springframework.boot:spring-boot-configuration-processor"
}

compileJava.dependsOn(processResources)
```

> **Note**
>
> You need to add `compileJava.dependsOn(processResources)` to your build to ensure that resources are processed before code is compiled. Without this directive, any `additional-spring-configuration-metadata.json` files are not processed.

The processor picks up both classes and methods that are annotated with `@ConfigurationProperties`. The Javadoc for field values within configuration classes is used to populate the `description` attribute.

> **Note**
>
> You should only use simple text with `@ConfigurationProperties` field Javadoc, since they are not processed before being added to the JSON.

Properties are discovered through the presence of standard getters and setters with special handling for collection types (that is detected even if only a getter is present). The annotation processor also supports the use of the `@Data`, `@Getter`, and `@Setter` lombok annotations.

> **Note**
>
> If you are using AspectJ in your project, you need to make sure that the annotation processor runs only once. There are several ways to do this. With Maven, you can configure the `maven-apt-plugin` explicitly and add the dependency to the annotation processor only there. You could also let the AspectJ plugin run all the processing and disable annotation processing in the `maven-compiler-plugin` configuration, as follows:
>
> ```
> <plugin>
>  <groupId>org.apache.maven.plugins</groupId>
>  <artifactId>maven-compiler-plugin</artifactId>
>  <configuration>
>   <proc>none</proc>
>  </configuration>
> </plugin>
> ```

Nested Properties

The annotation processor automatically considers inner classes as nested properties. Consider the following class:

```
@ConfigurationProperties(prefix="server")
public class ServerProperties {

 private String name;

 private Host host;

 // ... getter and setters

 private static class Host {

  private String ip;

  private int port;

  // ... getter and setters

 }
```

```
}
```

The preceding example produces metadata information for `server.name`, `server.host.ip`, and `server.host.port` properties. You can use the `@NestedConfigurationProperty` annotation on a field to indicate that a regular (non-inner) class should be treated as if it were nested.

> **Tip**
>
> This has no effect on collections and maps, as those types are automatically identified, and a single metadata property is generated for each of them.

Adding Additional Metadata

Spring Boot’s configuration file handling is quite flexible, and it is often the case that properties may exist that are not bound to a `@ConfigurationProperties` bean. You may also need to tune some attributes of an existing key. To support such cases and let you provide custom "hints", the annotation processor automatically merges items from `META-INF/additional-spring-configuration-metadata.json` into the main metadata file.

If you refer to a property that has been detected automatically, the description, default value, and deprecation information are overridden, if specified. If the manual property declaration is not identified in the current module, it is added as a new property.

The format of the `additional-spring-configuration-metadata.json` file is exactly the same as the regular `spring-configuration-metadata.json`. The additional properties file is optional. If you do not have any additional properties, do not add the file.

Appendix C. Auto-configuration classes

Here is a list of all auto-configuration classes provided by Spring Boot, with links to documentation and source code. Remember to also look at the conditions report in your application for more details of which features are switched on. (To do so, start the app with `--debug` or `-Ddebug` or, in an Actuator application, use the `conditions` endpoint).

C.1 From the “spring-boot-autoconfigure” module

The following auto-configuration classes are from the `spring-boot-autoconfigure` module:

Configuration Class	Links
`ActiveMQAutoConfiguration`	javadoc
`AopAutoConfiguration`	javadoc
`ArtemisAutoConfiguration`	javadoc
`BatchAutoConfiguration`	javadoc
`CacheAutoConfiguration`	javadoc
`CassandraAutoConfiguration`	javadoc
`CassandraDataAutoConfiguration`	javadoc
`CassandraReactiveDataAutoConfiguration`	javadoc
`CassandraReactiveRepositoriesAutoConfiguration`	javadoc
`CassandraRepositoriesAutoConfiguration`	javadoc
`CloudAutoConfiguration`	javadoc
`CodecsAutoConfiguration`	javadoc
`ConfigurationPropertiesAutoConfiguration`	javadoc
`CouchbaseAutoConfiguration`	javadoc
`CouchbaseDataAutoConfiguration`	javadoc
`CouchbaseReactiveDataAutoConfiguration`	javadoc
`CouchbaseReactiveRepositoriesAutoConfiguration`	javadoc
`CouchbaseRepositoriesAutoConfiguration`	javadoc
`DataSourceAutoConfiguration`	javadoc
`DataSourceTransactionManagerAutoConfiguration`	javadoc
`DispatcherServletAutoConfiguration`	javadoc
`ElasticsearchAutoConfiguration`	javadoc

Configuration Class	Links
`ElasticsearchDataAutoConfiguration`	javadoc
`ElasticsearchRepositoriesAutoConfiguration`	javadoc
`EmbeddedLdapAutoConfiguration`	javadoc
`EmbeddedMongoAutoConfiguration`	javadoc
`ErrorMvcAutoConfiguration`	javadoc
`ErrorWebFluxAutoConfiguration`	javadoc
`FlywayAutoConfiguration`	javadoc
`FreeMarkerAutoConfiguration`	javadoc
`GroovyTemplateAutoConfiguration`	javadoc
`GsonAutoConfiguration`	javadoc
`H2ConsoleAutoConfiguration`	javadoc
`HazelcastAutoConfiguration`	javadoc
`HazelcastJpaDependencyAutoConfiguration`	javadoc
`HibernateJpaAutoConfiguration`	javadoc
`HttpEncodingAutoConfiguration`	javadoc
`HttpHandlerAutoConfiguration`	javadoc
`HttpMessageConvertersAutoConfiguration`	javadoc
`HypermediaAutoConfiguration`	javadoc
`InfluxDbAutoConfiguration`	javadoc
`IntegrationAutoConfiguration`	javadoc
`JacksonAutoConfiguration`	javadoc
`JdbcTemplateAutoConfiguration`	javadoc
`JerseyAutoConfiguration`	javadoc
`JestAutoConfiguration`	javadoc
`JmsAutoConfiguration`	javadoc
`JmxAutoConfiguration`	javadoc
`JndiConnectionFactoryAutoConfiguration`	javadoc
`JndiDataSourceAutoConfiguration`	javadoc
`JooqAutoConfiguration`	javadoc
`JpaRepositoriesAutoConfiguration`	javadoc

Configuration Class	Links
JsonbAutoConfiguration	javadoc
JtaAutoConfiguration	javadoc
KafkaAutoConfiguration	javadoc
LdapAutoConfiguration	javadoc
LdapDataAutoConfiguration	javadoc
LdapRepositoriesAutoConfiguration	javadoc
LiquibaseAutoConfiguration	javadoc
MailSenderAutoConfiguration	javadoc
MailSenderValidatorAutoConfiguration	javadoc
MessageSourceAutoConfiguration	javadoc
MongoAutoConfiguration	javadoc
MongoDataAutoConfiguration	javadoc
MongoReactiveAutoConfiguration	javadoc
MongoReactiveDataAutoConfiguration	javadoc
MongoReactiveRepositoriesAutoConfiguration	javadoc
MongoRepositoriesAutoConfiguration	javadoc
MultipartAutoConfiguration	javadoc
MustacheAutoConfiguration	javadoc
Neo4jDataAutoConfiguration	javadoc
Neo4jRepositoriesAutoConfiguration	javadoc
OAuth2ClientAutoConfiguration	javadoc
PersistenceExceptionTranslationAutoConfiguration	javadoc
ProjectInfoAutoConfiguration	javadoc
PropertyPlaceholderAutoConfiguration	javadoc
QuartzAutoConfiguration	javadoc
RabbitAutoConfiguration	javadoc
ReactiveSecurityAutoConfiguration	javadoc
ReactiveWebServerAutoConfiguration	javadoc
ReactorCoreAutoConfiguration	javadoc
RedisAutoConfiguration	javadoc

Configuration Class	Links
`RedisReactiveAutoConfiguration`	javadoc
`RedisRepositoriesAutoConfiguration`	javadoc
`RepositoryRestMvcAutoConfiguration`	javadoc
`RestTemplateAutoConfiguration`	javadoc
`SecurityAutoConfiguration`	javadoc
`SecurityFilterAutoConfiguration`	javadoc
`SendGridAutoConfiguration`	javadoc
`ServletWebServerFactoryAutoConfiguration`	javadoc
`SessionAutoConfiguration`	javadoc
`SolrAutoConfiguration`	javadoc
`SolrRepositoriesAutoConfiguration`	javadoc
`SpringApplicationAdminJmxAutoConfiguration`	javadoc
`SpringDataWebAutoConfiguration`	javadoc
`ThymeleafAutoConfiguration`	javadoc
`TransactionAutoConfiguration`	javadoc
`ValidationAutoConfiguration`	javadoc
`WebClientAutoConfiguration`	javadoc
`WebFluxAutoConfiguration`	javadoc
`WebMvcAutoConfiguration`	javadoc
`WebServicesAutoConfiguration`	javadoc
`WebSocketMessagingAutoConfiguration`	javadoc
`WebSocketReactiveAutoConfiguration`	javadoc
`WebSocketServletAutoConfiguration`	javadoc
`XADataSourceAutoConfiguration`	javadoc

C.2 From the “spring-boot-actuator-autoconfigure” module

The following auto-configuration classes are from the `spring-boot-actuator-autoconfigure` module:

Configuration Class	Links
`AuditAutoConfiguration`	javadoc
`AuditEventsEndpointAutoConfiguration`	javadoc

Configuration Class	Links
`BeansEndpointAutoConfiguration`	javadoc
`CassandraHealthIndicatorAutoConfiguration`	javadoc
`CloudFoundryActuatorAutoConfiguration`	javadoc
`ConditionsReportEndpointAutoConfiguration`	javadoc
`ConfigurationPropertiesReportEndpointAutoConfiguration`	javadoc
`CouchbaseHealthIndicatorAutoConfiguration`	javadoc
`DataSourceHealthIndicatorAutoConfiguration`	javadoc
`DiskSpaceHealthIndicatorAutoConfiguration`	javadoc
`ElasticsearchHealthIndicatorAutoConfiguration`	javadoc
`EndpointAutoConfiguration`	javadoc
`EnvironmentEndpointAutoConfiguration`	javadoc
`FlywayEndpointAutoConfiguration`	javadoc
`HealthEndpointAutoConfiguration`	javadoc
`HealthIndicatorAutoConfiguration`	javadoc
`HeapDumpWebEndpointAutoConfiguration`	javadoc
`HttpTraceAutoConfiguration`	javadoc
`HttpTraceEndpointAutoConfiguration`	javadoc
`InfluxDbHealthIndicatorAutoConfiguration`	javadoc
`InfoContributorAutoConfiguration`	javadoc
`InfoEndpointAutoConfiguration`	javadoc
`JmsHealthIndicatorAutoConfiguration`	javadoc
`JmxEndpointAutoConfiguration`	javadoc
`JolokiaEndpointAutoConfiguration`	javadoc
`LdapHealthIndicatorAutoConfiguration`	javadoc
`LiquibaseEndpointAutoConfiguration`	javadoc
`LogFileWebEndpointAutoConfiguration`	javadoc
`LoggersEndpointAutoConfiguration`	javadoc
`MailHealthIndicatorAutoConfiguration`	javadoc
`ManagementContextAutoConfiguration`	javadoc
`MappingsEndpointAutoConfiguration`	javadoc

Configuration Class	Links
`MetricsAutoConfiguration`	javadoc
`MongoHealthIndicatorAutoConfiguration`	javadoc
`Neo4jHealthIndicatorAutoConfiguration`	javadoc
`RabbitHealthIndicatorAutoConfiguration`	javadoc
`ReactiveCloudFoundryActuatorAutoConfiguration`	javadoc
`ReactiveManagementContextAutoConfiguration`	javadoc
`RedisHealthIndicatorAutoConfiguration`	javadoc
`ScheduledTasksEndpointAutoConfiguration`	javadoc
`ServletManagementContextAutoConfiguration`	javadoc
`SessionsEndpointAutoConfiguration`	javadoc
`ShutdownEndpointAutoConfiguration`	javadoc
`SolrHealthIndicatorAutoConfiguration`	javadoc
`ThreadDumpEndpointAutoConfiguration`	javadoc
`WebEndpointAutoConfiguration`	javadoc

Appendix D. Test auto-configuration annotations

The following table lists the various `@…Test` annotations that can be used to test slices of your application and the auto-configuration that they import by default:

Test slice	Imported auto-configuration
`@DataJpaTest`	`org.springframework.boot.autoconfigure.cache.Cach` `org.springframework.boot.autoconfigure.data.jpa.J` `org.springframework.boot.autoconfigure.flyway.Fly` `org.springframework.boot.autoconfigure.jdbc.DataS` `org.springframework.boot.autoconfigure.jdbc.DataS` `org.springframework.boot.autoconfigure.jdbc.JdbcT` `org.springframework.boot.autoconfigure.liquibase.` `org.springframework.boot.autoconfigure.orm.jpa.Hi` `org.springframework.boot.autoconfigure.transactio` `org.springframework.boot.test.autoconfigure.jdbc.` `org.springframework.boot.test.autoconfigure.orm.j`
`@DataLdapTest`	`org.springframework.boot.autoconfigure.cache.Cach` `org.springframework.boot.autoconfigure.data.ldap.` `org.springframework.boot.autoconfigure.data.ldap.` `org.springframework.boot.autoconfigure.ldap.LdapA` `org.springframework.boot.autoconfigure.ldap.embed`
`@DataMongoTest`	`org.springframework.boot.autoconfigure.cache.Cach` `org.springframework.boot.autoconfigure.data.mongo` `org.springframework.boot.autoconfigure.data.mongo` `org.springframework.boot.autoconfigure.data.mongo` `org.springframework.boot.autoconfigure.data.mongo` `org.springframework.boot.autoconfigure.mongo.Mong` `org.springframework.boot.autoconfigure.mongo.Mong` `org.springframework.boot.autoconfigure.mongo.embe`
`@DataNeo4jTest`	`org.springframework.boot.autoconfigure.cache.Cach` `org.springframework.boot.autoconfigure.data.neo4j` `org.springframework.boot.autoconfigure.data.neo4j` `org.springframework.boot.autoconfigure.transactio`
`@DataRedisTest`	`org.springframework.boot.autoconfigure.cache.Cach` `org.springframework.boot.autoconfigure.data.redis` `org.springframework.boot.autoconfigure.data.redis`
`@JdbcTest`	`org.springframework.boot.autoconfigure.cache.Cach` `org.springframework.boot.autoconfigure.flyway.Fly` `org.springframework.boot.autoconfigure.jdbc.DataS` `org.springframework.boot.autoconfigure.jdbc.DataS` `org.springframework.boot.autoconfigure.jdbc.JdbcT` `org.springframework.boot.autoconfigure.liquibase.`

Test slice	Imported auto-configuration
	org.springframework.boot.autoconfigure.transactio org.springframework.boot.test.autoconfigure.jdbc.
@JooqTest	org.springframework.boot.autoconfigure.flyway.Fly org.springframework.boot.autoconfigure.jdbc.DataS org.springframework.boot.autoconfigure.jdbc.DataS org.springframework.boot.autoconfigure.jooq.JooqA org.springframework.boot.autoconfigure.liquibase. org.springframework.boot.autoconfigure.transactio
@JsonTest	org.springframework.boot.autoconfigure.cache.Cach org.springframework.boot.autoconfigure.gson.GsonA org.springframework.boot.autoconfigure.jackson.Ja org.springframework.boot.autoconfigure.jsonb.Json org.springframework.boot.test.autoconfigure.json.
@RestClientTest	org.springframework.boot.autoconfigure.cache.Cach org.springframework.boot.autoconfigure.gson.GsonA org.springframework.boot.autoconfigure.http.HttpM org.springframework.boot.autoconfigure.http.codec org.springframework.boot.autoconfigure.jackson.Ja org.springframework.boot.autoconfigure.jsonb.Json org.springframework.boot.autoconfigure.web.client org.springframework.boot.autoconfigure.web.reacti org.springframework.boot.test.autoconfigure.web.c org.springframework.boot.test.autoconfigure.web.c
@WebFluxTest	org.springframework.boot.autoconfigure.cache.Cach org.springframework.boot.autoconfigure.context.Me org.springframework.boot.autoconfigure.validation org.springframework.boot.autoconfigure.web.reacti org.springframework.boot.test.autoconfigure.web.r
@WebMvcTest	org.springframework.boot.autoconfigure.cache.Cach org.springframework.boot.autoconfigure.context.Me org.springframework.boot.autoconfigure.freemarker org.springframework.boot.autoconfigure.groovy.tem org.springframework.boot.autoconfigure.gson.GsonA org.springframework.boot.autoconfigure.hateoas.Hy org.springframework.boot.autoconfigure.http.HttpM org.springframework.boot.autoconfigure.jackson.Ja org.springframework.boot.autoconfigure.jsonb.Json org.springframework.boot.autoconfigure.mustache.M org.springframework.boot.autoconfigure.thymeleaf. org.springframework.boot.autoconfigure.validation org.springframework.boot.autoconfigure.web.servle org.springframework.boot.autoconfigure.web.servle org.springframework.boot.test.autoconfigure.web.s org.springframework.boot.test.autoconfigure.web.s

Test slice	Imported auto-configuration
	org.springframework.boot.test.autoconfigure.web.s org.springframework.boot.test.autoconfigure.web.s

Appendix E. The Executable Jar Format

The `spring-boot-loader` modules lets Spring Boot support executable jar and war files. If you use the Maven plugin or the Gradle plugin, executable jars are automatically generated, and you generally do not need to know the details of how they work.

If you need to create executable jars from a different build system or if you are just curious about the underlying technology, this section provides some background.

E.1 Nested JARs

Java does not provide any standard way to load nested jar files (that is, jar files that are themselves contained within a jar). This can be problematic if you need to distribute a self-contained application that can be run from the command line without unpacking.

To solve this problem, many developers use “shaded” jars. A shaded jar packages all classes, from all jars, into a single “uber jar”. The problem with shaded jars is that it becomes hard to see which libraries are actually in your application. It can also be problematic if the same filename is used (but with different content) in multiple jars. Spring Boot takes a different approach and lets you actually nest jars directly.

The Executable Jar File Structure

Spring Boot Loader-compatible jar files should be structured in the following way:

```
example.jar
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-BOOT-INF
    +-classes
    |  +-mycompany
    |     +-project
    |        +-YourClasses.class
    +-lib
       +-dependency1.jar
       +-dependency2.jar
```

Application classes should be placed in a nested `BOOT-INF/classes` directory. Dependencies should be placed in a nested `BOOT-INF/lib` directory.

The Executable War File Structure

Spring Boot Loader-compatible war files should be structured in the following way:

```
example.war
 |
 +-META-INF
 |  +-MANIFEST.MF
 +-org
 |  +-springframework
 |     +-boot
 |        +-loader
 |           +-<spring boot loader classes>
 +-WEB-INF
    +-classes
    |  +-com
    |     +-mycompany
    |        +-project
    |           +-YourClasses.class
    +-lib
    |  +-dependency1.jar
    |  +-dependency2.jar
    +-lib-provided
       +-servlet-api.jar
       +-dependency3.jar
```

Dependencies should be placed in a nested `WEB-INF/lib` directory. Any dependencies that are required when running embedded but are not required when deploying to a traditional web container should be placed in `WEB-INF/lib-provided`.

E.2 Spring Boot's "JarFile" Class

The core class used to support loading nested jars is `org.springframework.boot.loader.jar.JarFile`. It lets you load jar content from a standard jar file or from nested child jar data. When first loaded, the location of each `JarEntry` is mapped to a physical file offset of the outer jar, as shown in the following example:

```
myapp.jar
+-------------------+-------------------------+
| /BOOT-INF/classes | /BOOT-INF/lib/mylib.jar |
|+-----------------+||+-----------+----------+|
||     A.class      |||  B.class  |  C.class ||
|+-----------------+||+-----------+----------+|
+-------------------+-------------------------+
 ^                    ^           ^
 0063                 3452        3980
```

The preceding example shows how `A.class` can be found in `/BOOT-INF/classes` in `myapp.jar` at position `0063`. `B.class` from the nested jar can actually be found in `myapp.jar` at position `3452`, and `C.class` is at position `3980`.

Armed with this information, we can load specific nested entries by seeking to the appropriate part of the outer jar. We do not need to unpack the archive, and we do not need to read all entry data into memory.

Compatibility with the Standard Java "JarFile"

Spring Boot Loader strives to remain compatible with existing code and libraries. `org.springframework.boot.loader.jar.JarFile` extends from `java.util.jar.JarFile` and should work as a drop-in replacement. The `getURL()` method returns a `URL` that opens a connection compatible with `java.net.JarURLConnection` and can be used with Java's `URLClassLoader`.

E.3 Launching Executable Jars

The `org.springframework.boot.loader.Launcher` class is a special bootstrap class that is used as an executable jar's main entry point. It is the actual `Main-Class` in your jar file, and it is used to setup an appropriate `URLClassLoader` and ultimately call your `main()` method.

There are three launcher subclasses (`JarLauncher`, `WarLauncher`, and `PropertiesLauncher`). Their purpose is to load resources (`.class` files and so on.) from nested jar files or war files in directories (as opposed to those explicitly on the classpath). In the case of `JarLauncher` and `WarLauncher`, the nested paths are fixed. `JarLauncher` looks in `BOOT-INF/lib/`, and `WarLauncher` looks in `WEB-INF/lib/` and `WEB-INF/lib-provided/`. You can add extra jars in those locations if you want more. The `PropertiesLauncher` looks in `BOOT-INF/lib/` in your application archive by default, but you can add additional locations by setting an environment variable called `LOADER_PATH` or `loader.path` in `loader.properties` (which is a comma-separated list of directories, archives, or directories within archives).

Launcher Manifest

You need to specify an appropriate `Launcher` as the `Main-Class` attribute of `META-INF/MANIFEST.MF`. The actual class that you want to launch (that is, the class that contains a `main` method) should be specified in the `Start-Class` attribute.

The following example shows a typical `MANIFEST.MF` for an executable jar file:

```
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.mycompany.project.MyApplication
```

For a war file, it would be as follows:

```
Main-Class: org.springframework.boot.loader.WarLauncher
Start-Class: com.mycompany.project.MyApplication
```

Note

You need not specify `Class-Path` entries in your manifest file. The classpath is deduced from the nested jars.

Exploded Archives

Certain PaaS implementations may choose to unpack archives before they run. For example, Cloud Foundry operates this way. You can run an unpacked archive by starting the appropriate launcher, as follows:

```
$ unzip -q myapp.jar
$ java org.springframework.boot.loader.JarLauncher
```

E.4 `PropertiesLauncher` Features

`PropertiesLauncher` has a few special features that can be enabled with external properties (System properties, environment variables, manifest entries, or `loader.properties`). The following table describes these properties:

Key	Purpose
`loader.path`	Comma-separated Classpath, such as `lib`, `${HOME}/app/lib`. Earlier entries take precedence, like a regular `-classpath` on the `javac` command line.

Key	Purpose
`loader.home`	Used to resolve relative paths in `loader.path`. For example, given `loader.path=lib`, then `${loader.home}/lib` is a classpath location (along with all jar files in that directory). This property is also used to locate a `loader.properties` file, as in the following example `/opt/app` It defaults to `${user.dir}`.
`loader.args`	Default arguments for the main method (space separated).
`loader.main`	Name of main class to launch (for example, `com.app.Application`).
`loader.config.name`	Name of properties file (for example, `launcher`) It defaults to `loader`.
`loader.config.location`	Path to properties file (for example, `classpath:loader.properties`). It defaults to `loader.properties`.
`loader.system`	Boolean flag to indicate that all properties should be added to System properties It defaults to `false`.

When specified as environment variables or manifest entries, the following names should be used:

Key	Manifest entry	Environment variable
`loader.path`	`Loader-Path`	`LOADER_PATH`
`loader.home`	`Loader-Home`	`LOADER_HOME`
`loader.args`	`Loader-Args`	`LOADER_ARGS`
`loader.main`	`Start-Class`	`LOADER_MAIN`
`loader.config.location`	`Loader-Config-Location`	`LOADER_CONFIG_LOCATION`
`loader.system`	`Loader-System`	`LOADER_SYSTEM`

Tip

Build plugins automatically move the `Main-Class` attribute to `Start-Class` when the fat jar is built. If you use that, specify the name of the class to launch by using the `Main-Class` attribute and leaving out `Start-Class`.

The following rules apply to working with `PropertiesLauncher`:

- `loader.properties` is searched for in `loader.home`, then in the root of the classpath, and then in `classpath:/BOOT-INF/classes`. The first location where a file with that name exists is used.
- `loader.home` is the directory location of an additional properties file (overriding the default) only when `loader.config.location` is not specified.

- `loader.path` can contain directories (which are scanned recursively for jar and zip files), archive paths, a directory within an archive that is scanned for jar files (for example, `dependencies.jar!/lib`), or wildcard patterns (for the default JVM behavior). Archive paths can be relative to `loader.home` or anywhere in the file system with a `jar:file:` prefix.
- `loader.path` (if empty) defaults to `BOOT-INF/lib` (meaning a local directory or a nested one if running from an archive). Because of this, `PropertiesLauncher` behaves the same as `JarLauncher` when no additional configuration is provided.
- `loader.path` can not be used to configure the location of `loader.properties` (the classpath used to search for the latter is the JVM classpath when `PropertiesLauncher` is launched).
- Placeholder replacement is done from System and environment variables plus the properties file itself on all values before use.
- The search order for properties (where it makes sense to look in more than one place) is environment variables, system properties, `loader.properties`, the exploded archive manifest, and the archive manifest.

E.5 Executable Jar Restrictions

You need to consider the following restrictions when working with a Spring Boot Loader packaged application:

- Zip entry compression: The `ZipEntry` for a nested jar must be saved by using the `ZipEntry.STORED` method. This is required so that we can seek directly to individual content within the nested jar. The content of the nested jar file itself can still be compressed, as can any other entries in the outer jar.
- System classLoader: Launched applications should use `Thread.getContextClassLoader()` when loading classes (most libraries and frameworks do so by default). Trying to load nested jar classes with `ClassLoader.getSystemClassLoader()` fails. `java.util.Logging` always uses the system classloader. For this reason, you should consider a different logging implementation.

E.6 Alternative Single Jar Solutions

If the preceding restrictions mean that you cannot use Spring Boot Loader, consider the following alternatives:

- Maven Shade Plugin
- JarClassLoader
- OneJar

Appendix F. Dependency versions

The following table provides details of all of the dependency versions that are provided by Spring Boot in its CLI (Command Line Interface), Maven dependency management, and Gradle plugin. When you declare a dependency on one of these artifacts without declaring a version, the version listed in the table is used.

Group ID	Artifact ID	Version
`antlr`	`antlr`	2.7.7
`ch.qos.logback`	`logback-access`	1.2.3
`ch.qos.logback`	`logback-classic`	1.2.3
`ch.qos.logback`	`logback-core`	1.2.3
`com.atomikos`	`transactions-jdbc`	4.0.6
`com.atomikos`	`transactions-jms`	4.0.6
`com.atomikos`	`transactions-jta`	4.0.6
`com.couchbase.client`	`couchbase-spring-cache`	2.1.0
`com.couchbase.client`	`java-client`	2.5.4
`com.datastax.cassandra`	`cassandra-driver-core`	3.4.0
`com.datastax.cassandra`	`cassandra-driver-mapping`	3.4.0
`com.fasterxml`	`classmate`	1.3.4
`com.fasterxml.jackson.core`	`jackson-annotations`	2.9.0
`com.fasterxml.jackson.core`	`jackson-core`	2.9.2
`com.fasterxml.jackson.core`	`jackson-databind`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-avro`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-cbor`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-csv`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-ion`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-properties`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-protobuf`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-smile`	2.9.2
`com.fasterxml.jackson.dataformat`	`jackson-dataformat-xml`	2.9.2

Group ID	Artifact ID	Version
com.fasterxml.jackson.dataformat	jackson-dataformat-yaml	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-guava	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-hibernate3	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-hibernate4	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-hibernate5	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-hppc	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-jaxrs	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-jdk8	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-joda	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-json-org	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-jsr310	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-jsr353	2.9.2
com.fasterxml.jackson.datatype	jackson-datatype-pcollections	2.9.2
com.fasterxml.jackson.jaxrs	jackson-jaxrs-base	2.9.2
com.fasterxml.jackson.jaxrs	jackson-jaxrs-cbor-provider	2.9.2
com.fasterxml.jackson.jaxrs	jackson-jaxrs-json-provider	2.9.2
com.fasterxml.jackson.jaxrs	jackson-jaxrs-smile-provider	2.9.2
com.fasterxml.jackson.jaxrs	jackson-jaxrs-xml-provider	2.9.2
com.fasterxml.jackson.jaxrs	jackson-jaxrs-yaml-provider	2.9.2
com.fasterxml.jackson.jr	jackson-jr-all	2.9.2
com.fasterxml.jackson.jr	jackson-jr-objects	2.9.2
com.fasterxml.jackson.jr	jackson-jr-retrofit2	2.9.2
com.fasterxml.jackson.jr	jackson-jr-stree	2.9.2

Group ID	Artifact ID	Version
com.fasterxml.jackson.module	jackson-module-afterburner	2.9.2
com.fasterxml.jackson.module	jackson-module-guice	2.9.2
com.fasterxml.jackson.module	jackson-module-jaxb-annotations	2.9.2
com.fasterxml.jackson.module	jackson-module-jsonSchema	2.9.2
com.fasterxml.jackson.module	jackson-module-kotlin	2.9.2
com.fasterxml.jackson.module	jackson-module-mrbean	2.9.2
com.fasterxml.jackson.module	jackson-module-osgi	2.9.2
com.fasterxml.jackson.module	jackson-module-parameter-names	2.9.2
com.fasterxml.jackson.module	jackson-module-paranamer	2.9.2
com.fasterxml.jackson.module	jackson-module-scala_2.10	2.9.2
com.fasterxml.jackson.module	jackson-module-scala_2.11	2.9.2
com.fasterxml.jackson.module	jackson-module-scala_2.12	2.9.2
com.github.ben-manes.caffeine	caffeine	2.6.1
com.github.mxab.thymeleaf.extras	thymeleaf-extras-data-attribute	2.0.1
com.google.appengine	appengine-api-1.0-sdk	1.9.60
com.google.code.gson	gson	2.8.2
com.googlecode.json-simple	json-simple	1.1.1
com.h2database	h2	1.4.196
com.hazelcast	hazelcast	3.9.2
com.hazelcast	hazelcast-client	3.9.2
com.hazelcast	hazelcast-hibernate52	1.2.2
com.hazelcast	hazelcast-spring	3.9.2
com.jayway.jsonpath	json-path	2.4.0

Group ID	Artifact ID	Version
com.jayway.jsonpath	json-path-assert	2.4.0
com.microsoft.sqlserver	mssql-jdbc	6.2.2.jre8
com.querydsl	querydsl-apt	4.1.4
com.querydsl	querydsl-collections	4.1.4
com.querydsl	querydsl-core	4.1.4
com.querydsl	querydsl-jpa	4.1.4
com.querydsl	querydsl-mongodb	4.1.4
com.rabbitmq	amqp-client	5.1.2
com.samskivert	jmustache	1.14
com.sendgrid	sendgrid-java	4.1.2
com.sun.mail	javax.mail	1.6.0
com.timgroup	java-statsd-client	3.1.0
com.unboundid	unboundid-ldapsdk	4.0.4
com.zaxxer	HikariCP	2.7.6
commons-codec	commons-codec	1.11
commons-pool	commons-pool	1.6
de.flapdoodle.embed	de.flapdoodle.embed.mongo	2.0.1
dom4j	dom4j	1.6.1
io.dropwizard.metrics	metrics-annotation	3.2.6
io.dropwizard.metrics	metrics-core	3.2.6
io.dropwizard.metrics	metrics-ehcache	3.2.6
io.dropwizard.metrics	metrics-ganglia	3.2.6
io.dropwizard.metrics	metrics-graphite	3.2.6
io.dropwizard.metrics	metrics-healthchecks	3.2.6
io.dropwizard.metrics	metrics-httpasyncclient	3.2.6
io.dropwizard.metrics	metrics-jdbi	3.2.6
io.dropwizard.metrics	metrics-jersey	3.2.6
io.dropwizard.metrics	metrics-jersey2	3.2.6
io.dropwizard.metrics	metrics-jetty8	3.2.6
io.dropwizard.metrics	metrics-jetty9	3.2.6

Group ID	Artifact ID	Version
`io.dropwizard.metrics`	`metrics-jetty9-legacy`	3.2.6
`io.dropwizard.metrics`	`metrics-json`	3.2.6
`io.dropwizard.metrics`	`metrics-jvm`	3.2.6
`io.dropwizard.metrics`	`metrics-log4j`	3.2.6
`io.dropwizard.metrics`	`metrics-log4j2`	3.2.6
`io.dropwizard.metrics`	`metrics-logback`	3.2.6
`io.dropwizard.metrics`	`metrics-servlet`	3.2.6
`io.dropwizard.metrics`	`metrics-servlets`	3.2.6
`io.lettuce`	`lettuce-core`	5.0.1.RELEASE
`io.micrometer`	`micrometer-core`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-atlas`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-datadog`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-ganglia`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-graphite`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-influx`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-jmx`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-new-relic`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-prometheus`	1.0.0-rc.8
`io.micrometer`	`micrometer-registry-statsd`	1.0.0-rc.8
`io.netty`	`netty-all`	4.1.20.Final
`io.netty`	`netty-buffer`	4.1.20.Final
`io.netty`	`netty-codec`	4.1.20.Final
`io.netty`	`netty-codec-dns`	4.1.20.Final
`io.netty`	`netty-codec-haproxy`	4.1.20.Final
`io.netty`	`netty-codec-http`	4.1.20.Final

Group ID	Artifact ID	Version
io.netty	netty-codec-http2	4.1.20.Final
io.netty	netty-codec-memcache	4.1.20.Final
io.netty	netty-codec-mqtt	4.1.20.Final
io.netty	netty-codec-redis	4.1.20.Final
io.netty	netty-codec-smtp	4.1.20.Final
io.netty	netty-codec-socks	4.1.20.Final
io.netty	netty-codec-stomp	4.1.20.Final
io.netty	netty-codec-xml	4.1.20.Final
io.netty	netty-common	4.1.20.Final
io.netty	netty-dev-tools	4.1.20.Final
io.netty	netty-example	4.1.20.Final
io.netty	netty-handler	4.1.20.Final
io.netty	netty-handler-proxy	4.1.20.Final
io.netty	netty-resolver	4.1.20.Final
io.netty	netty-resolver-dns	4.1.20.Final
io.netty	netty-transport	4.1.20.Final
io.netty	netty-transport-native-epoll	4.1.20.Final
io.netty	netty-transport-native-kqueue	4.1.20.Final
io.netty	netty-transport-native-unix-common	4.1.20.Final
io.netty	netty-transport-rxtx	4.1.20.Final
io.netty	netty-transport-sctp	4.1.20.Final
io.netty	netty-transport-udt	4.1.20.Final
io.projectreactor	reactor-core	3.1.3.RELEASE
io.projectreactor	reactor-test	3.1.3.RELEASE
io.projectreactor.addons	reactor-adapter	3.1.4.RELEASE
io.projectreactor.addons	reactor-extra	3.1.4.RELEASE
io.projectreactor.addons	reactor-logback	3.1.4.RELEASE
io.projectreactor.ipc	reactor-netty	0.7.3.RELEASE

Group ID	Artifact ID	Version
`io.projectreactor.kafka`	`reactor-kafka`	1.0.0.RELEASE
`io.reactivex`	`rxjava`	1.3.4
`io.reactivex`	`rxjava-reactive-streams`	1.2.1
`io.reactivex.rxjava2`	`rxjava`	2.1.8
`io.rest-assured`	`rest-assured`	3.0.6
`io.searchbox`	`jest`	5.3.3
`io.undertow`	`undertow-core`	1.4.22.Final
`io.undertow`	`undertow-servlet`	1.4.22.Final
`io.undertow`	`undertow-websockets-jsr`	1.4.22.Final
`javax.annotation`	`javax.annotation-api`	1.3.1
`javax.cache`	`cache-api`	1.1.0
`javax.jms`	`javax.jms-api`	2.0.1
`javax.json`	`javax.json-api`	1.1.2
`javax.json.bind`	`javax.json.bind-api`	1.0
`javax.mail`	`javax.mail-api`	1.6.0
`javax.money`	`money-api`	1.0.1
`javax.servlet`	`javax.servlet-api`	3.1.0
`javax.servlet`	`jstl`	1.2
`javax.transaction`	`javax.transaction-api`	1.2
`javax.validation`	`validation-api`	2.0.1.Final
`jaxen`	`jaxen`	1.1.6
`joda-time`	`joda-time`	2.9.9
`junit`	`junit`	4.12
`mysql`	`mysql-connector-java`	5.1.45
`net.bytebuddy`	`byte-buddy`	1.7.9
`net.bytebuddy`	`byte-buddy-agent`	1.7.9
`net.java.dev.jna`	`jna`	4.5.1
`net.java.dev.jna`	`jna-platform`	4.5.1
`net.sf.ehcache`	`ehcache`	2.10.4
`net.sourceforge.htmlunit`	`htmlunit`	2.29

Group ID	Artifact ID	Version
`net.sourceforge.jtds`	`jtds`	1.3.1
`net.sourceforge.nekohtml`	`nekohtml`	1.9.22
`nz.net.ultraq.thymeleaf`	`thymeleaf-layout-dialect`	2.2.2
`org.apache.activemq`	`activemq-amqp`	5.15.2
`org.apache.activemq`	`activemq-blueprint`	5.15.2
`org.apache.activemq`	`activemq-broker`	5.15.2
`org.apache.activemq`	`activemq-camel`	5.15.2
`org.apache.activemq`	`activemq-client`	5.15.2
`org.apache.activemq`	`activemq-console`	5.15.2
`org.apache.activemq`	`activemq-http`	5.15.2
`org.apache.activemq`	`activemq-jaas`	5.15.2
`org.apache.activemq`	`activemq-jdbc-store`	5.15.2
`org.apache.activemq`	`activemq-jms-pool`	5.15.2
`org.apache.activemq`	`activemq-kahadb-store`	5.15.2
`org.apache.activemq`	`activemq-karaf`	5.15.2
`org.apache.activemq`	`activemq-leveldb-store`	5.15.2
`org.apache.activemq`	`activemq-log4j-appender`	5.15.2
`org.apache.activemq`	`activemq-mqtt`	5.15.2
`org.apache.activemq`	`activemq-openwire-generator`	5.15.2
`org.apache.activemq`	`activemq-openwire-legacy`	5.15.2
`org.apache.activemq`	`activemq-osgi`	5.15.2
`org.apache.activemq`	`activemq-partition`	5.15.2
`org.apache.activemq`	`activemq-pool`	5.15.2
`org.apache.activemq`	`activemq-ra`	5.15.2
`org.apache.activemq`	`activemq-run`	5.15.2
`org.apache.activemq`	`activemq-runtime-config`	5.15.2
`org.apache.activemq`	`activemq-shiro`	5.15.2
`org.apache.activemq`	`activemq-spring`	5.15.2

Group ID	Artifact ID	Version
`org.apache.activemq`	`activemq-stomp`	5.15.2
`org.apache.activemq`	`activemq-web`	5.15.2
`org.apache.activemq`	`artemis-amqp-protocol`	2.4.0
`org.apache.activemq`	`artemis-commons`	2.4.0
`org.apache.activemq`	`artemis-core-client`	2.4.0
`org.apache.activemq`	`artemis-jms-client`	2.4.0
`org.apache.activemq`	`artemis-jms-server`	2.4.0
`org.apache.activemq`	`artemis-journal`	2.4.0
`org.apache.activemq`	`artemis-native`	2.4.0
`org.apache.activemq`	`artemis-selector`	2.4.0
`org.apache.activemq`	`artemis-server`	2.4.0
`org.apache.activemq`	`artemis-service-extensions`	2.4.0
`org.apache.commons`	`commons-dbcp2`	2.2.0
`org.apache.commons`	`commons-lang3`	3.7
`org.apache.commons`	`commons-pool2`	2.5.0
`org.apache.derby`	`derby`	10.14.1.0
`org.apache.httpcomponents`	`httpasyncclient`	4.1.3
`org.apache.httpcomponents`	`httpclient`	4.5.5
`org.apache.httpcomponents`	`httpcore`	4.4.9
`org.apache.httpcomponents`	`httpcore-nio`	4.4.9
`org.apache.httpcomponents`	`httpmime`	4.5.5
`org.apache.johnzon`	`johnzon-jsonb`	1.1.5
`org.apache.logging.log4j`	`log4j-1.2-api`	2.10.0
`org.apache.logging.log4j`	`log4j-api`	2.10.0
`org.apache.logging.log4j`	`log4j-cassandra`	2.10.0
`org.apache.logging.log4j`	`log4j-core`	2.10.0
`org.apache.logging.log4j`	`log4j-couchdb`	2.10.0
`org.apache.logging.log4j`	`log4j-flume-ng`	2.10.0
`org.apache.logging.log4j`	`log4j-iostreams`	2.10.0
`org.apache.logging.log4j`	`log4j-jcl`	2.10.0

Group ID	Artifact ID	Version
org.apache.logging.log4j	log4j-jmx-gui	2.10.0
org.apache.logging.log4j	log4j-jul	2.10.0
org.apache.logging.log4j	log4j-liquibase	2.10.0
org.apache.logging.log4j	log4j-mongodb	2.10.0
org.apache.logging.log4j	log4j-slf4j-impl	2.10.0
org.apache.logging.log4j	log4j-taglib	2.10.0
org.apache.logging.log4j	log4j-to-slf4j	2.10.0
org.apache.logging.log4j	log4j-web	2.10.0
org.apache.solr	solr-analysis-extras	6.6.2
org.apache.solr	solr-analytics	6.6.2
org.apache.solr	solr-cell	6.6.2
org.apache.solr	solr-clustering	6.6.2
org.apache.solr	solr-core	6.6.2
org.apache.solr	solr-dataimporthandler	6.6.2
org.apache.solr	solr-dataimporthandler-extras	6.6.2
org.apache.solr	solr-langid	6.6.2
org.apache.solr	solr-solrj	6.6.2
org.apache.solr	solr-test-framework	6.6.2
org.apache.solr	solr-uima	6.6.2
org.apache.solr	solr-velocity	6.6.2
org.apache.tomcat	tomcat-annotations-api	8.5.27
org.apache.tomcat	tomcat-catalina-jmx-remote	8.5.27
org.apache.tomcat	tomcat-jdbc	8.5.27
org.apache.tomcat	tomcat-jsp-api	8.5.27
org.apache.tomcat.embed	tomcat-embed-core	8.5.27
org.apache.tomcat.embed	tomcat-embed-el	8.5.27
org.apache.tomcat.embed	tomcat-embed-jasper	8.5.27
org.apache.tomcat.embed	tomcat-embed-websocket	8.5.27
org.aspectj	aspectjrt	1.8.13

Group ID	Artifact ID	Version
`org.aspectj`	`aspectjtools`	1.8.13
`org.aspectj`	`aspectjweaver`	1.8.13
`org.assertj`	`assertj-core`	3.9.0
`org.codehaus.btm`	`btm`	2.1.4
`org.codehaus.groovy`	`groovy`	2.4.13
`org.codehaus.groovy`	`groovy-all`	2.4.13
`org.codehaus.groovy`	`groovy-ant`	2.4.13
`org.codehaus.groovy`	`groovy-bsf`	2.4.13
`org.codehaus.groovy`	`groovy-console`	2.4.13
`org.codehaus.groovy`	`groovy-docgenerator`	2.4.13
`org.codehaus.groovy`	`groovy-groovydoc`	2.4.13
`org.codehaus.groovy`	`groovy-groovysh`	2.4.13
`org.codehaus.groovy`	`groovy-jmx`	2.4.13
`org.codehaus.groovy`	`groovy-json`	2.4.13
`org.codehaus.groovy`	`groovy-jsr223`	2.4.13
`org.codehaus.groovy`	`groovy-nio`	2.4.13
`org.codehaus.groovy`	`groovy-servlet`	2.4.13
`org.codehaus.groovy`	`groovy-sql`	2.4.13
`org.codehaus.groovy`	`groovy-swing`	2.4.13
`org.codehaus.groovy`	`groovy-templates`	2.4.13
`org.codehaus.groovy`	`groovy-test`	2.4.13
`org.codehaus.groovy`	`groovy-testng`	2.4.13
`org.codehaus.groovy`	`groovy-xml`	2.4.13
`org.codehaus.janino`	`janino`	3.0.8
`org.eclipse.jetty`	`apache-jsp`	9.4.8.v20171121
`org.eclipse.jetty`	`apache-jstl`	9.4.8.v20171121
`org.eclipse.jetty`	`jetty-alpn-client`	9.4.8.v20171121
`org.eclipse.jetty`	`jetty-alpn-conscrypt-client`	9.4.8.v20171121
`org.eclipse.jetty`	`jetty-alpn-conscrypt-server`	9.4.8.v20171121

Group ID	Artifact ID	Version
org.eclipse.jetty	jetty-alpn-java-client	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-java-server	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-openjdk8-client	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-openjdk8-server	9.4.8.v20171121
org.eclipse.jetty	jetty-alpn-server	9.4.8.v20171121
org.eclipse.jetty	jetty-annotations	9.4.8.v20171121
org.eclipse.jetty	jetty-ant	9.4.8.v20171121
org.eclipse.jetty	jetty-client	9.4.8.v20171121
org.eclipse.jetty	jetty-continuation	9.4.8.v20171121
org.eclipse.jetty	jetty-deploy	9.4.8.v20171121
org.eclipse.jetty	jetty-distribution	9.4.8.v20171121
org.eclipse.jetty	jetty-hazelcast	9.4.8.v20171121
org.eclipse.jetty	jetty-home	9.4.8.v20171121
org.eclipse.jetty	jetty-http	9.4.8.v20171121
org.eclipse.jetty	jetty-http-spi	9.4.8.v20171121
org.eclipse.jetty	jetty-infinispan	9.4.8.v20171121
org.eclipse.jetty	jetty-io	9.4.8.v20171121
org.eclipse.jetty	jetty-jaas	9.4.8.v20171121
org.eclipse.jetty	jetty-jaspi	9.4.8.v20171121
org.eclipse.jetty	jetty-jmx	9.4.8.v20171121
org.eclipse.jetty	jetty-jndi	9.4.8.v20171121
org.eclipse.jetty	jetty-nosql	9.4.8.v20171121
org.eclipse.jetty	jetty-plus	9.4.8.v20171121
org.eclipse.jetty	jetty-proxy	9.4.8.v20171121
org.eclipse.jetty	jetty-quickstart	9.4.8.v20171121
org.eclipse.jetty	jetty-rewrite	9.4.8.v20171121
org.eclipse.jetty	jetty-security	9.4.8.v20171121
org.eclipse.jetty	jetty-server	9.4.8.v20171121
org.eclipse.jetty	jetty-servlet	9.4.8.v20171121

Group ID	Artifact ID	Version
org.eclipse.jetty	jetty-servlets	9.4.8.v20171121
org.eclipse.jetty	jetty-spring	9.4.8.v20171121
org.eclipse.jetty	jetty-unixsocket	9.4.8.v20171121
org.eclipse.jetty	jetty-util	9.4.8.v20171121
org.eclipse.jetty	jetty-util-ajax	9.4.8.v20171121
org.eclipse.jetty	jetty-webapp	9.4.8.v20171121
org.eclipse.jetty	jetty-xml	9.4.8.v20171121
org.eclipse.jetty.cdi	cdi-core	9.4.8.v20171121
org.eclipse.jetty.cdi	cdi-full-servlet	9.4.8.v20171121
org.eclipse.jetty.cdi	cdi-servlet	9.4.8.v20171121
org.eclipse.jetty.fcgi	fcgi-client	9.4.8.v20171121
org.eclipse.jetty.fcgi	fcgi-server	9.4.8.v20171121
org.eclipse.jetty.gcloud	jetty-gcloud-session-manager	9.4.8.v20171121
org.eclipse.jetty.http2	http2-client	9.4.8.v20171121
org.eclipse.jetty.http2	http2-common	9.4.8.v20171121
org.eclipse.jetty.http2	http2-hpack	9.4.8.v20171121
org.eclipse.jetty.http2	http2-http-client-transport	9.4.8.v20171121
org.eclipse.jetty.http2	http2-server	9.4.8.v20171121
org.eclipse.jetty.memcached	jetty-memcached-sessions	9.4.8.v20171121
org.eclipse.jetty.orbit	javax.servlet.jsp	2.2.0.v201112011158
org.eclipse.jetty.osgi	jetty-httpservice	9.4.8.v20171121
org.eclipse.jetty.osgi	jetty-osgi-boot	9.4.8.v20171121
org.eclipse.jetty.osgi	jetty-osgi-boot-jsp	9.4.8.v20171121
org.eclipse.jetty.osgi	jetty-osgi-boot-warurl	9.4.8.v20171121
org.eclipse.jetty.websocket	javax-websocket-client-impl	9.4.8.v20171121
org.eclipse.jetty.websocket	javax-websocket-server-impl	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-api	9.4.8.v20171121

Group ID	Artifact ID	Version
org.eclipse.jetty.websocket	websocket-client	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-common	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-server	9.4.8.v20171121
org.eclipse.jetty.websocket	websocket-servlet	9.4.8.v20171121
org.ehcache	ehcache	3.4.0
org.ehcache	ehcache-clustered	3.4.0
org.ehcache	ehcache-transactions	3.4.0
org.elasticsearch	elasticsearch	5.6.6
org.elasticsearch.client	transport	5.6.6
org.elasticsearch.plugin	transport-netty4-client	5.6.6
org.firebirdsql.jdbc	jaybird-jdk17	3.0.3
org.firebirdsql.jdbc	jaybird-jdk18	3.0.3
org.flywaydb	flyway-core	5.0.6
org.freemarker	freemarker	2.3.27-incubating
org.glassfish	javax.el	3.0.0
org.glassfish.jersey.containers	jersey-container-servlet	2.26
org.glassfish.jersey.containers	jersey-container-servlet-core	2.26
org.glassfish.jersey.core	jersey-client	2.26
org.glassfish.jersey.core	jersey-common	2.26
org.glassfish.jersey.core	jersey-server	2.26
org.glassfish.jersey.ext	jersey-bean-validation	2.26
org.glassfish.jersey.ext	jersey-entity-filtering	2.26
org.glassfish.jersey.ext	jersey-spring4	2.26
org.glassfish.jersey.media	jersey-media-jaxb	2.26
org.glassfish.jersey.media	jersey-media-json-jackson	2.26
org.glassfish.jersey.media	jersey-media-multipart	2.26
org.hamcrest	hamcrest-core	1.3
org.hamcrest	hamcrest-library	1.3

Group ID	Artifact ID	Version
org.hibernate	hibernate-c3p0	5.2.12.Final
org.hibernate	hibernate-core	5.2.12.Final
org.hibernate	hibernate-ehcache	5.2.12.Final
org.hibernate	hibernate-entitymanager	5.2.12.Final
org.hibernate	hibernate-envers	5.2.12.Final
org.hibernate	hibernate-hikaricp	5.2.12.Final
org.hibernate	hibernate-infinispan	5.2.12.Final
org.hibernate	hibernate-java8	5.2.12.Final
org.hibernate	hibernate-jcache	5.2.12.Final
org.hibernate	hibernate-jpamodelgen	5.2.12.Final
org.hibernate	hibernate-proxool	5.2.12.Final
org.hibernate	hibernate-spatial	5.2.12.Final
org.hibernate	hibernate-testing	5.2.12.Final
org.hibernate	hibernate-validator-annotation-processor	6.0.7.Final
org.hibernate.validator	hibernate-validator	6.0.7.Final
org.hsqldb	hsqldb	2.4.0
org.infinispan	infinispan-jcache	9.1.4.Final
org.infinispan	infinispan-spring4-common	9.1.4.Final
org.infinispan	infinispan-spring4-embedded	9.1.4.Final
org.influxdb	influxdb-java	2.8
org.javassist	javassist	3.22.0-CR2
org.jboss	jboss-transaction-spi	7.6.0.Final
org.jboss.logging	jboss-logging	3.3.1.Final
org.jboss.narayana.jta	jdbc	5.7.2.Final
org.jboss.narayana.jta	jms	5.7.2.Final
org.jboss.narayana.jta	jta	5.7.2.Final
org.jboss.narayana.jts	narayana-jts-integration	5.7.2.Final
org.jdom	jdom2	2.0.6

Group ID	Artifact ID	Version
org.jetbrains.kotlin	kotlin-reflect	1.2.20
org.jetbrains.kotlin	kotlin-runtime	1.2.20
org.jetbrains.kotlin	kotlin-stdlib	1.2.20
org.jetbrains.kotlin	kotlin-stdlib-jdk7	1.2.20
org.jetbrains.kotlin	kotlin-stdlib-jdk8	1.2.20
org.jetbrains.kotlin	kotlin-stdlib-jre7	1.2.20
org.jetbrains.kotlin	kotlin-stdlib-jre8	1.2.20
org.jolokia	jolokia-core	1.4.0
org.jooq	jooq	3.10.4
org.jooq	jooq-codegen	3.10.4
org.jooq	jooq-meta	3.10.4
org.junit.jupiter	junit-jupiter-api	5.0.3
org.junit.jupiter	junit-jupiter-engine	5.0.3
org.liquibase	liquibase-core	3.5.3
org.mariadb.jdbc	mariadb-java-client	2.2.1
org.mockito	mockito-core	2.13.0
org.mockito	mockito-inline	2.13.0
org.mongodb	bson	3.6.1
org.mongodb	mongodb-driver	3.6.1
org.mongodb	mongodb-driver-async	3.6.1
org.mongodb	mongodb-driver-core	3.6.1
org.mongodb	mongodb-driver-reactivestreams	1.7.0
org.mongodb	mongo-java-driver	3.6.1
org.mortbay.jasper	apache-el	8.5.24.1
org.neo4j	neo4j-ogm-api	3.0.3
org.neo4j	neo4j-ogm-bolt-driver	3.0.3
org.neo4j	neo4j-ogm-core	3.0.3
org.neo4j	neo4j-ogm-http-driver	3.0.3
org.postgresql	postgresql	42.2.1
org.projectlombok	lombok	1.16.20

Group ID	Artifact ID	Version
org.quartz-scheduler	quartz	2.3.0
org.reactivestreams	reactive-streams	1.0.2
org.seleniumhq.selenium	htmlunit-driver	2.29.0
org.seleniumhq.selenium	selenium-api	3.8.1
org.seleniumhq.selenium	selenium-chrome-driver	3.8.1
org.seleniumhq.selenium	selenium-edge-driver	3.8.1
org.seleniumhq.selenium	selenium-firefox-driver	3.8.1
org.seleniumhq.selenium	selenium-ie-driver	3.8.1
org.seleniumhq.selenium	selenium-java	3.8.1
org.seleniumhq.selenium	selenium-opera-driver	3.8.1
org.seleniumhq.selenium	selenium-remote-driver	3.8.1
org.seleniumhq.selenium	selenium-safari-driver	3.8.1
org.seleniumhq.selenium	selenium-support	3.8.1
org.skyscreamer	jsonassert	1.5.0
org.slf4j	jcl-over-slf4j	1.7.25
org.slf4j	jul-to-slf4j	1.7.25
org.slf4j	log4j-over-slf4j	1.7.25
org.slf4j	slf4j-api	1.7.25
org.slf4j	slf4j-ext	1.7.25
org.slf4j	slf4j-jcl	1.7.25
org.slf4j	slf4j-jdk14	1.7.25
org.slf4j	slf4j-log4j12	1.7.25
org.slf4j	slf4j-nop	1.7.25
org.slf4j	slf4j-simple	1.7.25
org.springframework	spring-aop	5.0.3.RELEASE
org.springframework	spring-aspects	5.0.3.RELEASE
org.springframework	spring-beans	5.0.3.RELEASE
org.springframework	spring-context	5.0.3.RELEASE
org.springframework	spring-context-indexer	5.0.3.RELEASE
org.springframework	spring-context-support	5.0.3.RELEASE

Group ID	Artifact ID	Version
org.springframework	spring-core	5.0.3.RELEASE
org.springframework	spring-expression	5.0.3.RELEASE
org.springframework	spring-instrument	5.0.3.RELEASE
org.springframework	spring-jcl	5.0.3.RELEASE
org.springframework	spring-jdbc	5.0.3.RELEASE
org.springframework	spring-jms	5.0.3.RELEASE
org.springframework	spring-messaging	5.0.3.RELEASE
org.springframework	spring-orm	5.0.3.RELEASE
org.springframework	spring-oxm	5.0.3.RELEASE
org.springframework	spring-test	5.0.3.RELEASE
org.springframework	spring-tx	5.0.3.RELEASE
org.springframework	spring-web	5.0.3.RELEASE
org.springframework	spring-webflux	5.0.3.RELEASE
org.springframework	spring-webmvc	5.0.3.RELEASE
org.springframework	spring-websocket	5.0.3.RELEASE
org.springframework.amqp	spring-amqp	2.0.2.RELEASE
org.springframework.amqp	spring-rabbit	2.0.2.RELEASE
org.springframework.batch	spring-batch-core	4.0.0.RELEASE
org.springframework.batch	spring-batch-infrastructure	4.0.0.RELEASE
org.springframework.batch	spring-batch-integration	4.0.0.RELEASE
org.springframework.batch	spring-batch-test	4.0.0.RELEASE
org.springframework.boot	spring-boot	2.0.0.RC1
org.springframework.boot	spring-boot-actuator	2.0.0.RC1
org.springframework.boot	spring-boot-actuator-autoconfigure	2.0.0.RC1
org.springframework.boot	spring-boot-autoconfigure	2.0.0.RC1
org.springframework.boot	spring-boot-autoconfigure-processor	2.0.0.RC1

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-configuration-metadata	2.0.0.RC1
org.springframework.boot	spring-boot-configuration-processor	2.0.0.RC1
org.springframework.boot	spring-boot-devtools	2.0.0.RC1
org.springframework.boot	spring-boot-loader	2.0.0.RC1
org.springframework.boot	spring-boot-loader-tools	2.0.0.RC1
org.springframework.boot	spring-boot-properties-migrator	2.0.0.RC1
org.springframework.boot	spring-boot-starter	2.0.0.RC1
org.springframework.boot	spring-boot-starter-activemq	2.0.0.RC1
org.springframework.boot	spring-boot-starter-actuator	2.0.0.RC1
org.springframework.boot	spring-boot-starter-amqp	2.0.0.RC1
org.springframework.boot	spring-boot-starter-aop	2.0.0.RC1
org.springframework.boot	spring-boot-starter-artemis	2.0.0.RC1
org.springframework.boot	spring-boot-starter-batch	2.0.0.RC1
org.springframework.boot	spring-boot-starter-cache	2.0.0.RC1
org.springframework.boot	spring-boot-starter-cloud-connectors	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-cassandra	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-cassandra-reactive	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-couchbase	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-couchbase-reactive	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-elasticsearch	2.0.0.RC1

Group ID	Artifact ID	Version
org.springframework.boot	spring-boot-starter-data-jpa	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-ldap	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-mongodb	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-mongodb-reactive	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-neo4j	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-redis	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-redis-reactive	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-rest	2.0.0.RC1
org.springframework.boot	spring-boot-starter-data-solr	2.0.0.RC1
org.springframework.boot	spring-boot-starter-freemarker	2.0.0.RC1
org.springframework.boot	spring-boot-starter-groovy-templates	2.0.0.RC1
org.springframework.boot	spring-boot-starter-hateoas	2.0.0.RC1
org.springframework.boot	spring-boot-starter-integration	2.0.0.RC1
org.springframework.boot	spring-boot-starter-jdbc	2.0.0.RC1
org.springframework.boot	spring-boot-starter-jersey	2.0.0.RC1
org.springframework.boot	spring-boot-starter-jetty	2.0.0.RC1
org.springframework.boot	spring-boot-starter-jooq	2.0.0.RC1
org.springframework.boot	spring-boot-starter-json	2.0.0.RC1

Group ID	Artifact ID	Version
`org.springframework.boot`	`spring-boot-starter-jta-atomikos`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-jta-bitronix`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-jta-narayana`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-log4j2`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-logging`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-mail`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-mustache`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-quartz`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-reactor-netty`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-security`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-test`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-thymeleaf`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-tomcat`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-undertow`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-validation`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-web`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-webflux`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-web-services`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-starter-websocket`	2.0.0.RC1

Group ID	Artifact ID	Version
`org.springframework.boot`	`spring-boot-test`	2.0.0.RC1
`org.springframework.boot`	`spring-boot-test-autoconfigure`	2.0.0.RC1
`org.springframework.cloud`	`spring-cloud-cloudfoundry-connector`	2.0.1.RELEASE
`org.springframework.cloud`	`spring-cloud-connectors-core`	2.0.1.RELEASE
`org.springframework.cloud`	`spring-cloud-heroku-connector`	2.0.1.RELEASE
`org.springframework.cloud`	`spring-cloud-localconfig-connector`	2.0.1.RELEASE
`org.springframework.cloud`	`spring-cloud-spring-service-connector`	2.0.1.RELEASE
`org.springframework.data`	`spring-data-cassandra`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-commons`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-couchbase`	3.0.3.RELEASE
`org.springframework.data`	`spring-data-elasticsearch`	3.0.3.RELEASE
`org.springframework.data`	`spring-data-envers`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-gemfire`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-geode`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-jpa`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-keyvalue`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-ldap`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-mongodb`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-mongodb-cross-store`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-neo4j`	5.0.3.RELEASE
`org.springframework.data`	`spring-data-redis`	2.0.3.RELEASE
`org.springframework.data`	`spring-data-rest-core`	3.0.3.RELEASE
`org.springframework.data`	`spring-data-rest-hal-browser`	3.0.3.RELEASE
`org.springframework.data`	`spring-data-rest-webmvc`	3.0.3.RELEASE

Group ID	Artifact ID	Version
`org.springframework.data`	`spring-data-solr`	3.0.3.RELEASE
`org.springframework.hateoas`	`spring-hateoas`	0.24.0.RELEASE
`org.springframework.integration`	`spring-integration-amqp`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-core`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-event`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-feed`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-file`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-ftp`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-gemfire`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-groovy`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-http`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-ip`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-jdbc`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-jms`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-jmx`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-jpa`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-mail`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-mongodb`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-mqtt`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-redis`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-rmi`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-scripting`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-security`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-sftp`	5.0.1.RELEASE
`org.springframework.integration`	`spring-integration-stomp`	5.0.1.RELEASE

Group ID	Artifact ID	Version
org.springframework.integration	spring-integration-stream	5.0.1.RELEASE
org.springframework.integration	spring-integration-syslog	5.0.1.RELEASE
org.springframework.integration	spring-integration-test	5.0.1.RELEASE
org.springframework.integration	spring-integration-test-support	5.0.1.RELEASE
org.springframework.integration	spring-integration-twitter	5.0.1.RELEASE
org.springframework.integration	spring-integration-webflux	5.0.1.RELEASE
org.springframework.integration	spring-integration-websocket	5.0.1.RELEASE
org.springframework.integration	spring-integration-ws	5.0.1.RELEASE
org.springframework.integration	spring-integration-xml	5.0.1.RELEASE
org.springframework.integration	spring-integration-xmpp	5.0.1.RELEASE
org.springframework.integration	spring-integration-zookeeper	5.0.1.RELEASE
org.springframework.kafka	spring-kafka	2.1.2.RELEASE
org.springframework.kafka	spring-kafka-test	2.1.2.RELEASE
org.springframework.ldap	spring-ldap-core	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-core-tiger	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-ldif-batch	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-ldif-core	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-odm	2.3.2.RELEASE
org.springframework.ldap	spring-ldap-test	2.3.2.RELEASE
org.springframework.plugin	spring-plugin-core	1.2.0.RELEASE
org.springframework.plugin	spring-plugin-metadata	1.2.0.RELEASE
org.springframework.restdocs	spring-restdocs-asciidoctor	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-core	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-mockmvc	2.0.0.RELEASE

Group ID	Artifact ID	Version
org.springframework.restdocs	spring-restdocs-restassured	2.0.0.RELEASE
org.springframework.restdocs	spring-restdocs-webtestclient	2.0.0.RELEASE
org.springframework.retry	spring-retry	1.2.2.RELEASE
org.springframework.security	spring-security-acl	5.0.1.RELEASE
org.springframework.security	spring-security-aspects	5.0.1.RELEASE
org.springframework.security	spring-security-cas	5.0.1.RELEASE
org.springframework.security	spring-security-config	5.0.1.RELEASE
org.springframework.security	spring-security-core	5.0.1.RELEASE
org.springframework.security	spring-security-crypto	5.0.1.RELEASE
org.springframework.security	spring-security-data	5.0.1.RELEASE
org.springframework.security	spring-security-ldap	5.0.1.RELEASE
org.springframework.security	spring-security-messaging	5.0.1.RELEASE
org.springframework.security	spring-security-oauth2-client	5.0.1.RELEASE
org.springframework.security	spring-security-oauth2-core	5.0.1.RELEASE
org.springframework.security	spring-security-oauth2-jose	5.0.1.RELEASE
org.springframework.security	spring-security-openid	5.0.1.RELEASE
org.springframework.security	spring-security-remoting	5.0.1.RELEASE
org.springframework.security	spring-security-taglibs	5.0.1.RELEASE
org.springframework.security	spring-security-test	5.0.1.RELEASE
org.springframework.security	spring-security-web	5.0.1.RELEASE
org.springframework.session	spring-session-core	2.0.1.RELEASE
org.springframework.session	spring-session-data-mongodb	2.0.0.RELEASE
org.springframework.session	spring-session-data-redis	2.0.1.RELEASE
org.springframework.session	spring-session-hazelcast	2.0.1.RELEASE

Group ID	Artifact ID	Version
`org.springframework.session`	`spring-session-jdbc`	2.0.1.RELEASE
`org.springframework.ws`	`spring-ws-core`	3.0.0.RELEASE
`org.springframework.ws`	`spring-ws-security`	3.0.0.RELEASE
`org.springframework.ws`	`spring-ws-support`	3.0.0.RELEASE
`org.springframework.ws`	`spring-ws-test`	3.0.0.RELEASE
`org.synchronoss.cloud`	`nio-multipart-parser`	1.1.0
`org.thymeleaf`	`thymeleaf`	3.0.9.RELEASE
`org.thymeleaf`	`thymeleaf-spring5`	3.0.9.RELEASE
`org.thymeleaf.extras`	`thymeleaf-extras-java8time`	3.0.1.RELEASE
`org.thymeleaf.extras`	`thymeleaf-extras-springsecurity4`	3.0.2.RELEASE
`org.webjars`	`hal-browser`	3325375
`org.webjars`	`webjars-locator`	0.32-1
`org.xerial`	`sqlite-jdbc`	3.21.0.1
`org.xmlunit`	`xmlunit-core`	2.5.1
`org.xmlunit`	`xmlunit-legacy`	2.5.1
`org.xmlunit`	`xmlunit-matchers`	2.5.1
`org.yaml`	`snakeyaml`	1.19
`redis.clients`	`jedis`	2.9.0
`wsdl4j`	`wsdl4j`	1.6.3
`xml-apis`	`xml-apis`	1.4.01