Spring JavaConfig Reference Guide

Rod Johnson
Costin Leau
Chris Beams

Version 1.0.0.M4

Published November 2008

Table of Contents

Preface

Spring Java Configuration (JavaConfig) provides a pure-Java means of configuring the Spring IoC container. By taking advantage of Java 5.0 language features such as annotations and generics, JavaConfig allows users to express configuration logic and metadata directly in code, alleviating any need for XML.

By relying only on basic Java syntax and language features, JavaConfig offers several distinct advantages:

- JavaConfig provides a truly object-oriented mechanism for dependency injection, meaning you can take full advantage of reuse, inheritance and polymorphism in your configuration code.
- You are given complete control over instantiation and dependency injection, meaning that even the most complex objects can be dealt with gracefully.
- Because only Java is required, you are left with fully refactorable configuration logic that requires no special tooling beyond your IDE.

To get a sense of how to use JavaConfig, let's configure an application consisting of two beans:

```
@Configuration
public class AppConfig {
    @Bean
    public Service service() {
        return new ServiceImpl(repository());
    }

    @Bean
    public Repository repository() {
        return new JdbcRepository();
    }
}
```

Bootstrapping this application would then be as simple as the following:

```
public class AppBootstrap {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx = new JavaConfigApplicationContext(AppConfig.class);
        Service service = ctx.getBean(Service.class);
        service.doSomething();
    }
}
```

While this example is a trivial one, JavaConfig can flex to meet the needs of the most complex and sophisticated enterprise applications. This guide will show you how.

1. Introduction

1.1. What this guide covers

This guide covers all aspects of Spring JavaConfig. It covers implementing and using `@Configuration` classes to configure the Spring IoC container and working with the feature set. It also covers extending and customizing JavaConfig.

Topics not covered

For the purposes of this document, a general familiarity with the core concepts of the Spring IoC container are assumed. It is beyond the scope of this document to discuss such matters at length. If the following concepts are not familiar, it is recommended that you first read Chapter 3. IoC from the core Spring Framework documentation (or that you reference its relevant sections on an as-needed basis while reading this document): inversion of control (IoC) and/or dependency injection (DI), spring-managed beans, bean scoping, autowiring, `BeanFactory`, `ApplicationContext`, `BeanFactoryPostProcessor` , `BeanPostProcessor`, Spring's initialization, destruction, and other bean lifecycle callback mechanisms.

If general aspect oriented programming (AOP) concepts are unfamiliar, or AOP with Spring and AspectJ 5's `@Aspect` style are unfamiliar, it is recommended that you first read Chapter 6. AOP from the Core Spring Framework documentation.

1.2. What JavaConfig requires to run

- Java 5.0 or higher
- Spring 2.5.6 or higher
- AspectJ 1.6.2 or higher
- CGLIB 2.1.3

1.3. Where to get support

Professional from-the-source support for Spring JavaConfig is available from SpringSource, the company behind Spring.

1.4. Where to follow development

You can provide feedback and help make JavaConfig best serve the needs of the Spring community by interacting with the developers at the Spring JavaConfig Community Forum.

Report bugs and influence the JavaConfig project roadmap using Spring's JIRA issue tracker.

Browse JavaConfig sources and subscribe to RSS commit feeds using the Spring's FishEye service.

Stay tuned to Continuous Integration and nightly snapshot build status using Spring's Bamboo service.

Subscribe to the Spring Community Portal and SpringSource Team Blog for the latest Spring news and announcements, including information on Spring JavaConfig releases.

Follow JavaConfig development, get release notifications and provide feedback to the JavaConfig team via Twitter .

1.5. How to retrieve and build JavaConfig from source

You must have Java 5.0 (or better) and Ant 1.7.0 (or better) installed to build JavaConfig from source.

```
svn co https://src.springframework.org/svn/spring-javaconfig/trunk/ spring-javaconfig
cd spring-javaconfig/org.springframework.config.java
ant clean test
```

Publishing JavaConfig artifacts to a local Maven repository

```
ant jar publish-maven-local
```

1.6. How to import JavaConfig as an Eclipse project

The JavaConfig source tree contains Eclipse project metadata (.classpath and .project files), making it convienient to import. Also, as Ivy is used to resolve dependencies, it is necessary to set up an IVY_CACHE variable as detailed below.

```
From within Eclipse:
1. Preferences->Java->Build Path->Classpath Variables->New...
   Name:  IVY_CACHE
   Value: [your-javaconfig-root]>/ivy-cache/repository
2. File->Import->Existing Projects into Workspace
3. Select [your-javaconfig-root]/org.springframework.config.java/
4. org.springframework.config.java should show up with a checkbox
```

The project should now be imported, error-free and ready for development.

1.7. How to obtain milestone builds

Via manual download

Milestone builds are available from Spring's milestone build area.

Via Maven

To access milestone builds using Maven, add the following repositories to your Maven pom:

```
<repository>
    <id>com.springsource.repository.bundles.milestone</id>
    <name>SpringSource Enterprise Bundle Repository - SpringSource Milestone Releases</name>
    <url>http://repository.springsource.com/maven/bundles/milestone</url>
</repository>

<repository>
    <id>com.springsource.repository.bundles.external</id>
    <name>SpringSource Enterprise Bundle Repository - External Releases</name>
    <url>http://repository.springsource.com/maven/bundles/external</url>
</repository>
```

Then add the following dependency:

```
<dependency>
    <groupId>org.springframework.javaconfig</groupId>
    <artifactId>org.springframework.config.java</artifactId>
    <version>1.0.0.M4</version>
</dependency>
```

Via Ivy

To access milestone builds using Ivy, add the following repositories to your Ivy config:

```
<url name="com.springsource.repository.bundles.milestone">
    <ivy pattern="http://repository.springsource.com/ivy/bundles/milestone/
                  [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
    <artifact pattern="http://repository.springsource.com/ivy/bundles/milestone/
                       [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
</url>

<url name="com.springsource.repository.bundles.external">
    <ivy pattern="http://repository.springsource.com/ivy/bundles/external/
                  [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
    <artifact pattern="http://repository.springsource.com/ivy/bundles/external/
                       [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
</url>
```

Then declare the following dependency:

```
<dependency org="org.springframework.javaconfig" name="org.springframework.config.java"
            rev="1.0.0.M4" conf="compile->runtime"/>
```

1.8. How to obtain nightly (aka 'snapshot') builds

Via manual download

Nightly builds are available from Spring's snapshot build area.

Via Maven

To access nightly builds using Maven, add the following repositories to your Maven pom:

```
<repository>
    <id>com.springsource.repository.bundles.snapshot</id>
    <name>SpringSource Enterprise Bundle Repository - SpringSource Snapshot Builds</name>
    <url>http://repository.springsource.com/maven/bundles/snapshot</url>
</repository>

<repository>
    <id>com.springsource.repository.bundles.external</id>
    <name>SpringSource Enterprise Bundle Repository - External Releases</name>
    <url>http://repository.springsource.com/maven/bundles/external</url>
</repository>
```

Then add the following dependency:

```
<dependency>
    <groupId>org.springframework.javaconfig</groupId>
    <artifactId>org.springframework.config.java</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
</dependency>
```

Via Ivy

To access nightly builds using Ivy, add the following repositories to your Ivy config:

```
<url name="com.springsource.repository.bundles.snapshot">
    <ivy pattern="http://repository.springsource.com/ivy/bundles/snapshot/
                  [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
    <artifact pattern="http://repository.springsource.com/ivy/bundles/snapshot/
                       [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
</url>
```

```
<url name="com.springsource.repository.bundles.external">
    <ivy pattern="http://repository.springsource.com/ivy/bundles/external/
                  [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
    <artifact pattern="http://repository.springsource.com/ivy/bundles/external/
                       [organisation]/[module]/[revision]/[artifact]-[revision].[ext]"/>
</url>
```

Then declare the following dependency:

```
<dependency org="org.springframework.javaconfig" name="org.springframework.config.java"
            rev="1.0.0.BUILD-SNAPSHOT" conf="compile->runtime"/>
```

2. Authoring @Configuration classes

The central artifact in Spring JavaConfig is the `@Configuration`-annotated class. These classes consist principally of `@Bean`-annotated methods that define instantiation, configuration, and initialization logic for objects that will be managed by the Spring IoC container.

2.1. @Configuration

Annotating a class with the `@Configuration` indicates that the class may be used by JavaConfig as a source of bean definitions. The simplest possible `@Configuration` class would read as follows:

```
@Configuration
public class ApplicationConfig {
}
```

An application may make use of just one `@Configuration`-annotated class, or many. `@Configuration` can be considered the equivalent of XML's `<beans/>` element. Like `<beans/>`, it provides an opportunity to explicitly set defaults for all enclosed bean definitions.

```
@Configuration(defaultAutowire = Autowire.BY_TYPE, defaultLazy = Lazy.FALSE)
public class ApplicationConfig {
    // bean definitions follow
}
```

Because the semantics of the attributes to the `@Configuration` annotation are 1:1 with the attributes to the `<beans/>` element, this documentation defers to the beans-definition section of Chapter 3, IoC from the Core Spring documentation. See also the Javadoc for `@Configuration` for details on each of the available annotation attributes.

Tip

Jump to Section 3.1, " Bootstrapping applications with JavaConfigApplicationContext " to see how `@Configuration` classes are used to create a Spring Application Context.

2.2. @Bean

`@Bean` is a method-level annotation and a direct analog of the XML `<bean/>` element. The annotation supports most of the attributes offered by `<bean/>`, such as: `init-method`, `destroy-method`, `autowiring`, `lazy-init`, `dependency-check`, `depends-on` and `scope`.

Declaring a bean

To declare a bean, simply annotate a method with the `@Bean` annotation. When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a `BeanFactory`. By default, the bean name will be the same as the method name (see bean naming for details on how to customize this behavior). The following is a simple example of a `@Bean` method declaration:

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

For comparison sake, the configuration above is exactly equivalent to the following Spring XML:

```
<beans>
    <bean name="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

Both will result in a bean named `transferService` being available in the `BeanFactory` / `ApplicationContext`, bound to an object instance of type `TransferServiceImpl`:

```
transferService -> com.acme.TransferServiceImpl
```

Injecting dependencies

When `@Bean`s have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new Foo(bar());
    }

    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```

In the example above, the `foo` bean recevies a reference to `bar` via constructor injection.

Receiving lifecycle callbacks

Using JSR-250 annotations

JavaConfig, like the core Spring Framework, supports use of JSR-250 "Common Annotations". For example:

```
public class FooService {
    @PostConstruct
    public void init() {
        // custom initialization logic
    }
}

@Configuration
@AnnotationDrivenConfig
public class ApplicationConfig {
    @Bean
    public FooService fooService() {
        return new FooService();
    }
}
```

In the above example, `FooService` declares `@PostConstruct` . By declaring JavaConfig's `@AnnotationDrivenConfig` on The `@Configuration` class, this annotation will be respected by the container and called immediately after construction. See The core framework documentation on support for JSR-250 annotations for further details.

Using Spring interfaces

Spring's lifecycle callbacks are fully supported. If a bean implements `InitializingBean`, `DisposableBean`, or `Lifecycle`, their respective methods will be called by the container in accordance with their Javadoc.

Using `@Bean` `initMethodName` / `destroyMethodName` attributes

The `@Bean` annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's `init-method` and `destroy-method` attributes to the `bean` element:

```
public class Foo {
    public void init() {
        // initialization logic
    }
}
public class Bar {
    public void cleanup() {
        // destruction logic
    }
}
@Configuration
public class AppConfig {
    @Bean(initMethodName="init")
```

```
    public Foo foo() {
        return new Foo();
    }
    @Bean(destroyMethodName="cleanup")
    public Bar bar() {
        return new Bar();
    }
}
```

Of course, in the case of `Foo` above, it would be equally as valid to call the `init()` method directly during construction:

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        Foo foo = new Foo();
        foo.init();
        return foo;
    }

    // ...
}
```

Tip

Remember that because you are working directly in Java, you can do anything you like with your objects, and do not always need to rely on the container!

Using `*Aware` interfaces

The standard set of `*Aware` interfaces such as `BeanFactoryAware`, `BeanNameAware`, `MessageSourceAware`, `ApplicationContextAware`, etc. are fully supported. Consider an example class that implements `BeanFactoryAware`:

```
public class AwareBean implements BeanFactoryAware {

    private BeanFactory factory;

    // BeanFactoryAware setter (called by Spring during bean instantiation)
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
        this.factory = beanFactory;
    }

    public void close(){
        // do clean-up
    }
}
```

If the class above were declared as a bean as follows:

```
@Configuration
```

```
public class AppConfig {
    @Bean
    public AwareBean awareBean() {
        return new AwareBean();
    }
}
```

its `setBeanFactory` method will be called during initialization, providing the bean with access to its enclosing `BeanFactory`.

Specifying bean scope

Using @Bean's scope attribute

JavaConfig makes available each of the four standard scopes specified in Section 3.4, "Bean Scopes" of the Spring reference documentation.

The `DefaultScopes` class provides string constants for each of these four scopes. SINGLETON is the default, and can be overridden by supplying the `scope` attribute to `@Bean` annotation:

```
@Configuration
public class MyConfiguration {
    @Bean(scope=DefaultScopes.PROTOTYPE)
    public Encryptor encryptor() {
        // ...
    }
}
```

`@ScopedProxy`

Spring offers a convenient way of working with scoped dependencies through scoped proxies. The easiest way to create such a proxy when using the XML configuration is the `<aop:scoped-proxy/>` element. JavaConfig offers equivalent support with the `@ScopedProxy` annotation, which provides the same semantics and configuration options.

If we were to port the the XML reference documentation scoped proxy example (see link above) to JavaConfig, it would look like the following:

```
// a HTTP Session-scoped bean exposed as a proxy
@Bean(scope = DefaultScopes.SESSION)
@ScopedProxy
public UserPreferences userPreferences() {
   return new UserPreferences();
}

@Bean
public Service userService() {
   UserService service = new SimpleUserService();
   // a reference to the proxied 'userPreferences' bean
   service.seUserPreferences(userPreferences());
```

```
    return service;
}
```

Lookup method injection

As noted in the core documentation, lookup method injection is an advanced feature that should be comparatively rarely used. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. JavaConfig provides a natural means for implementing this pattern. *Note that the example below is adapted from the example classes and configuration in the core documentation linked above.*

```
package fiona.apple;

public abstract class CommandManager {
    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();

        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

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

JavaConfig can easily create a subclass of `CommandManager` where the abstract `createCommand()` is overridden in such a way that it 'looks up' a brand new (prototype) command object:

```
@Bean(scope=DefaultScopes.PROTOTYPE)
public AsyncCommand asyncCommand() {
    AsyncCommand command = new AsyncCommand();
    // inject dependencies here as required
    return command;
}

@Bean
public CommandManager commandManager() {
    // return new anonymous implementation of CommandManager with command() overridden
    // to return a new prototype Command object
    return new CommandManager() {
        protected Command command() {
            return asyncCommand();
        }
    }
}
```

Customizing bean naming

By default, JavaConfig uses a `@Bean` method's name as the name of the resulting bean. This functionality can be overridden, however, using the `BeanNamingStrategy` extension point.

```
public class Main {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx = new JavaConfigApplicationContext();
        ctx.setBeanNamingStrategy(new CustomBeanNamingStrategy());
        ctx.addConfigClass(MyConfig.class);
        ctx.refresh();
        ctx.getBean("customBeanName");
    }
}
```

Note

`JavaConfigApplicationContext` will be covered in detail in Chapter 3, *Using @Configuration classes*

For more details, see the API documentation for `BeanNamingStrategy`.

Working with Spring `FactoryBean` implementations

Spring provides many implementations of the `FactoryBean` interface. Usually these classes are used to support integrations with other frameworks. Take for example `org.springframework.orm.hibernate3.LocalSessionFactoryBean`. This class is used to create a Hibernate SessionFactory and requires as dependencies the location of Hibernate mapping files and a DataSource. Here's how it is commonly used in XML:

```
<beans>
    <bean id="sessionFactory"
          class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="mappingResources">
            <list>
                <value>com/acme/Bank.hbm.xml</value>
                <value>com/acme/Account.hbm.xml</value>
                <value>com/acme/Customer.hbm.xml</value>
            </list>
        </property>
    </bean>

    <bean id="dataSource" class="...">
        <!-- ... -->
    </bean>
</beans>
```

The Spring container recognizes that `LocalSessionFactoryBean` implements the `FactoryBean` interface, and thus treats this bean specially: An instance of `LocalSessionFactoryBean` is instantiated, but instead of being directly returned, instead the `getObject()` method is invoked. It is the object returned from this call `getObject()` that is ultimately registered as the `sessionFactory` bean.

How then would we use `LocalSessionFactoryBean` in JavaConfig? The best approach is to extend the `ConfigurationSupport` base class and use the `getObject()` method:

```
@Configuration
public class DataAccessConfig extends ConfigurationSupport {
    @Bean
    public SessionFactory sessionFactory() {
        LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();
        factoryBean.setDataSource(dataSource());
        ArrayList<String> mappingFiles = new ArrayList<String>();
        mappingFiles.add("com/acme/Bank.hbm.xml");
        mappingFiles.add("com/acme/Account.hbm.xml");
        mappingFiles.add("com/acme/Customer.hbm.xml");
        factoryBean.setMappingResources(mappingFiles);
        return this.getObject(SessionFactory.class, factoryBean);
    }
    // ... other beans, including dataSource() ...
}
```

Notice the call to `this.getObject(Class, FactoryBean)`? This call ensures that any container callbacks are invoked on the `FactoryBean` object, and then returns the value from the FactoryBean's `getObject()` in a type-safe fashion.

3. Using `@Configuration` classes

3.1. Bootstrapping applications with `JavaConfigApplicationContext`

`JavaConfigApplicationContext` provides direct access to the beans defined by `@Configuration`-annotated classes. For more information on the `ApplicationContext` API in general, please refer to the Core Spring documentation.

Construction Options

Instantiating the `JavaConfigApplicationContext` can be done by supplying `@Configuration` class literals to the constructor, and/or strings representing packages to scan for `@Configuration` classes.

Construction by class literal

Each of the class literals supplied to the constructor will be processed, and for each `@Bean` method encountered, JavaConfig will create a bean definition and ultimately instantiate and initialize the bean.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(AppConfig.class);
Service service = context.getBean(Service.class);
```

Passing multiple `@Configuration` classes:

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfig.class, DataConfig.class);
Service service = context.getBean(Service.class);
```

Construction by base package

Base packages will be scanned for the existence of any `@Configuration` classes. Any candidate classes will then be processed much as if they had been supplied directly as class literals to the constructor.

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("com.acme.app.configuration");
Service service = context.getBean(Service.class);
```

Passing multiple base packages:

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("com.acme.configuration", "com.acme.other");
Service service = context.getBean(Service.class);
```

Matching packages and classes by wildcard:

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext("**/configuration/**/*.class", "**/other/*Config.class");
Service service = context.getBean(Service.class);
```

Note

The wildcard syntax for matching packages and classes above is based on Ant Patterns

Post-construction configuration

When one or more classes/packages are supplied as constructor arguments, a `JavaConfigApplicationContext` instance cannot be further configured. If post-construction configuration is preferred or required, use either the no-arg constructor, configure by calling setters, then manually refresh the context. After the call to `refresh()`, the context will be 'closed for configuration'.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext();
context.setParent(otherConfig);
context.setConfigClasses(AppConfig.class, DataConfig.class);
context.setBasePackages("com.acme.configuration");
context.refresh();
Service service = (Service) context.getBean("serviceA");
```

Note

Whenever multiple packages and/or classes are used to instantiate a `JavaConfigApplicationContext`, *order matters*. This is important when considering what happens if two configuration classes define a bean with the same name. The last-specified class wins.

Accessing beans with `getBean()`

`JavaConfigApplicationContext` provides several variants of the `getBean()` method for accessing beans.

Type-safe access

The preferred method for accessing beans is with the type-safe `getBean()` method.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class);
```

Disambuguation options

If more than one bean of type `Service` had been defined in the example above, the call to `getBean()` would have thrown an exception indicating an ambiguity that the container could not resolve. In these cases, the user has a number of options for disambiguation:

Indicating a `@Bean` as `primary`

Like Spring's XML configuration, JavaConfig allows for specifying a given `@Bean` as `primary`:

```
@Configuration
public class MyConfig {
    @Bean(primary=Primary.TRUE)
    public Service myService() {
        return new Service();
    }

    @Bean
    public Service backupService() {
        return new Service();
    }
}
```

After this modification, all calls to `getBean(Service.class)` will return the `primary` bean

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class); // returns the myService() primary bean
```

Disambiguation by bean name

JavaConfig provides a `getBean()` variant that accepts both a class and a bean name for cases just such as these.

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = context.getBean(Service.class, "myService");
```

Because bean ids must be unique, this call guarantees that the ambiguity cannot occur.

Retrieve all beans of a given type

It is also reasonable to call the `getBeansOfType()` method in order to return all beans that

implement a given interface:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Map matchingBeans = context.getBeansOfType(Service.class);
```

Note that this latter approach is actually a feature of the Core Spring Framework's `AbstractApplicationContext` (which `JavaConfigApplicationContext` extends) and is not type-safe, in that the returned `Map` is not parameterized.

String-based access

Beans may be accessed via the traditional string-based `getBean()` API as well. Of course this is not type-safe and requires casting, but avoids any potential ambiguity entirely:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(...);
Service service = (Service) context.getBean("myService");
```

3.2. Bootstrapping web applications with `JavaConfigWebApplicationContext`

`JavaConfigWebApplicationContext` allows for seamlessly bootstrapping `@Configuration` classes within your servlet container's `web.xml` deployment descriptor. This process requires no Spring XML whatsoever:

```
<web-app>
    <!-- Configure ContextLoaderListener to use JavaConfigWebApplicationContext
         instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>
            org.springframework.config.java.context.JavaConfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-delimited
         fully-qualified @Configuration classes -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>example.ApplicationConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using ContextLoaderListener -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
```

```
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <!-- Configure DispatcherServlet to use JavaConfigWebApplicationContext
             instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>
                org.springframework.config.java.context.JavaConfigWebApplicationContext
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more comma- or space-delimited
             and fully-qualified @Configuration classes -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>example.web.ServletConfig</param-value>
        </init-param>
    </servlet>

    <!-- map all requests for /main/* to the dispatcher servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/main/*</url-pattern>
    </servlet-mapping>
</web-app>
```

For basic information regarding initialization parameters to `DispatcherServlet` and use of `ContextLoaderListener`, see Chapter 13. Web MVC framework of the Core Spring Framework documentation.

4. Modularizing configurations

While the simplest configuration may be expressed as a single class that exposes several beans, it is often desirable to *modularize* configurations for reuse and clarity.

4.1. Partitioning bean definitions into multiple `@Configuration` classes

The simplest technique for modularizing configurations is to split up a single `@Configuration` class into multiple smaller classes:

```
// monolithic configuration
@Configuration
public class AppConfig {
    @Bean
    public ServiceA serviceA() {
       // ...
    }

    @Bean
    public ServiceB serviceB() {
       // ...
    }

   // assume many bean definitions follow
}
```

The above configuration class might be supplied as a parameter to `JavaConfigApplicationContext`:

```
JavaConfigApplicationContext context = new JavaConfigApplicationContext(AppConfig.class);
ServiceA serviceA = context.getBean(ServiceA.class);
ServiceB serviceB = context.getBean(ServiceB.class);
```

We can easily partition this configuration such that bean definitions are spread across two classes, instead of one:

```
// partitioned configuration
@Configuration
public class AppConfigA {
    @Bean
    public ServiceA serviceA() {
       // ...
    }
}

@Configuration
public class AppConfigB {
    @Bean
```

```
    public ServiceB serviceB() {
        // ...
    }
}
```

Now simply supply both configuration classes to the constructor of `JavaConfigApplicationContext`:

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(AppConfigA.class, AppConfigB.class);
// both beans are still available in the resulting application context
ServiceA serviceA = context.getBean(ServiceA.class);
ServiceB serviceB = context.getBean(ServiceB.class);
```

4.2. Referencing beans across `@Configuration` classes

One configuration class may need to reference a bean defined in another configuration class (or in XML, for that matter). The preferred mechanism for doing this is using Spring's `@Autowired` annotation:

Direct bean references with `@Autowired`

One `@Configuration` class may directly reference bean instances registered from another using Spring's `@Autowired` annotation.

```
@Configuration
public class ConfigOne {
    @Bean
    public AccountRepository accountRepository() {
        // create and return an AccountRepository object
    }
}

@Configuration
@AnnotationDrivenConfig
public class ConfigTwo {
    @Autowired AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }
}
```

Given that both these configuration classes are supplied to the application context at runtime, the `AccountRepository` bean declared in `ConfigOne` will be autowired (injected) into the `AccountRepository` field in `ConfigTwo`.

```
JavaConfigApplicationContext context =
    new JavaConfigApplicationContext(ConfigOne.class, ConfigTwo.class);
```

Fully-qualified bean references with `@Autowired`

In addition to being able to reference any particular bean definition as seen above, one `@Configuration` class may reference the instance of any other `@Configuration` class using `@Autowired`. This works because the `@Configuration` classes themselves are instantiated and managed as individual Spring beans.

```
@Configuration
public class ConfigOne {
    @Bean
    public AccountRepository accountRepository() {
        // create and return an AccountRepository object
    }
}

@Configuration
@AnnotationDrivenConfig
public class ConfigTwo {
    @Autowired ConfigOne configOne;

    @Bean
    public TransferService transferService() {
         // transferService references accountRepository in a 'fully-qualified' fashion:
        return new TransferServiceImpl(configOne.accountRepository());
    }
}
```

Tip

The 'fully-qualified' approach is generally preferred as it provides a the significant advantage of being able to easily navigate within an IDE to the source of the referenced bean.

Open issue: Should `@AnnotationDrivenConfig` be enabled by default? Rationale: given that `@Autowired` is the preferred method for referencing external beans, it is likely to need to be enabled in all but the most trivial configurations. See SJC-219.

4.3. Aggregating `@Configuration` classes with `@Import`

Thus far, we've seen how to break up bean definitions into multiple `@Configuration` classes and how to reference those beans across `@Configuration` boundaries. These scenarios have required providing all `@Configuration` classes to the constructor of a `JavaConfigApplicationContext`, and this is not always ideal. Often it is preferable to use an *aggregation* approach, where one `@Configuration` class logically *imports* the bean definitions defined by another.

The `@Import` annotation provides just this kind of support, and it is the direct equivalent of the `<import/>` element found in Spring beans XML files.

```
@Configuration
public class DataSourceConfig {
    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(...);
    }
}

@Configuration
@AnnotationDrivenConfig
@Import(DataSourceConfig.class) // <-- AppConfig imports DataSourceConfig
public class AppConfig extends ConfigurationSupport {
    @Autowired DataSourceConfig dataSourceConfig;

    @Bean
    public void TransferService transferService() {
        return new TransferServiceImpl(dataSourceConfig.dataSource());
    }
}
```

The bootstrapping of this application is simplified, as it only needs to supply `AppConfig` when instantiating a `JavaConfigApplicationContext`.

```
public class Main {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx =
            new JavaConfigApplicationContext(AppConfig.class); // specifies single class
        // ...
    }
}
```

Multiple configurations may be imported by supplying an array of classes to the `@Import` annotation

```
@Configuration
@Import({ DataSourceConfig.class, TransactionConfig.class })
public class AppConfig extends ConfigurationSupport {
    // @Bean methods here can reference @Bean methods in DataSourceConfig or TransactionConfig
}
```

4.4. `ConfigurationSupport`

As a convenience, `@Configuration` classses may extend `ConfigurationSupport`, primarily in order to facilitate easy lookup of beans from the enclosing `ApplicationContext`.

```
@Configuration
public class AppConfig extends ConfigurationSupport {
    @Bean
    public Service serviceA() {
         // referencing a bean of type DataSource declared elsewhere
        DataSource dataSource = this.getBean(DataSource.class);
        return new ServiceImpl(dataSource);
    }
```

```
}
```

5. Working with externalized values

5.1. `@ExternalValue`

What about `PropertyOverrideConfigurer`?

Those familiar with XML configuration will notice that there is not a direct equivalent for the popular `PropertyOverrideConfigurer` or the more recent `<context:property-placeholder/>`. However, the combined use of JavaConfig's `@ExternalValue` and various `ValueSource` annotations provide similar functionality.

Externally defined values such as usernames, passwords, file paths, and the like may be accessed using `@ExternalValue` and one or more of JavaConfig's `ValueSource` annotations.

`@ExternalValue` fields

```
@Configuration
@PropertiesValueSource("classpath:com/acme/db.properties")
public class AppConfig {
    @ExternalValue("datasource.username") String username;

    @Bean
    public TestBean testBean() {
        return new TestBean(username);
    }
}
```

`com/acme/db.properties` will be read from the classpath and the value associated with key `datasource.username` will be injected into the `username` field. The contents of `db.properties` might be as follows:

```
datasource.username=scott
datasource.password=tiger
...
```

Note

An array of properties file locations may be supplied to `@PropertiesValueSource`, and along with `classpath:`, all of the standard Spring resource-loading prefixes are supported, such as `file:` and `http:`.

```
@Configuration
```

```
@PropertiesValueSource({"classpath:com/acme/a.properties", "file:/opt/acme/b.properties"})
public class AppConfig {
    // ...
}
```

`@ExternalValue` methods

`@ExternalValue` may also be used as a method-level annotation

```
@Configuration
@PropertiesValueSource("classpath:com/acme/db.properties")
public abstract class AppConfig {
    @ExternalValue("datasource.username")
    abstract String username();

    @Bean
    public TestBean testBean() {
        return new TestBean(username());
    }
}
```

The primary advantage to using `@ExternalValue` methods is that rather than injecting the external value just once (as is done in the case of `@ExternalValue` fields), `@ExternalValue` methods are evaluated every time they're referenced. As this is not usually required, `@ExternalValue` fields are the preferred method. A downside of `@ExternalValue` methods is that they should be abstract, requiring you to declare the entire `@Configuration` class abstract, and this is not in alignment with the semantics users typically associate with using the `abstract` keyword.

5.2. Available `ValueSource` annotations

- `@PropertiesValueSource`
- `@EnvironmentValueSource`
- `@SystemPropertiesValueSource`

`ValueSource` annotations may be used in conjunction:

```
@Configuration
@EnvironmentValueSource
@SystemPropertiesValueSource
@PropertiesValueSource("classpath:com/acme/db.properties")
public class AppConfig {
    @ExternalValue("datasource.username") String username;

    @Bean
```

```
    public TestBean testBean() {
        return new TestBean(username);
    }
}
```

In this example, `datasource.username` will be looked for in `db.properties`, in the set of environment variables present at runtime and in the system properties.

Note

Explicit ordering of `ValueSource` annotations is not yet supported but will be soon. See SJC-170 and SJC-171 for details.

6. Combining configuration styles

JavaConfig can be used in conjunction with any or all of Spring's other container configuration approaches.

6.1. JavaConfig and XML

Bootstrapping JavaConfig from XML with `ConfigurationPostProcessor`

You may desire or be required to use XML as the primary mechanism for configuring the container, but wish to selectively use `@Configuration` classes to define certain beans. For such cases, JavaConfig provides `ConfigurationPostProcessor`, a Spring `BeanPostProcessor` capable of processing `@Configuration` classes.

```
<beans>
    <!-- first, define your individual @Configuration classes as beans -->
    <bean class="com.myapp.config.AppConfig"/>
    <bean class="com.myapp.config.DataConfig"/>

    <!-- be sure to include the JavaConfig bean post-processor -->
    <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>
</beans>
```

Then, bootstrap an XML ApplicationContext:

```
ApplicationContext context = new ClassPathXmlApplicationContext("application-config.xml");
```

The beans defined in `AppConfig` and `DataConfig` will be available via `context`.

Configuring configurations

An added benefit that comes along with bootstrapping JavaConfig from XML is that the configuration bean instances are eligible, just as any other bean, for dependency injection:

```
<beans>
    <!-- a possible "configurable configuration" -->
    <bean class="org.my.company.config.AppConfiguration">
        <property name="env" value="TESTING"/>
        <property name="monitoring" value="true"/>
        <property name="certificates" value="classpath:/META-INF/config/MyCompany.certs"/>
    </bean>
    <!-- JavaConfig post-processor -->
    <bean class="org.springframework.config.java.process.ConfigurationPostProcessor"/>
</beans>
```

Bootstrapping XML from JavaConfig with `@ImportXml`

The `@ImportXml` annotation is provided to support importing beans defined in XML into `@Configuration` classes.
`datasource-config.xml`:

```
<beans>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
        <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
        <property name="username" value="sa"/>
        <property name="password" value=""/>
    </bean>
</beans>
```

```
@Configuration
@AnnotationDrivenConfig // enable the @Autowired annotation
@ImportXml("classpath:com/company/app/datasource-config.xml")
public class Config {
    // autowire the DataSource bean declared in datasource-config.xml
    @Autowired DataSource dataSource;

    @Bean
    public FooRepository fooRepository() {
        // inject the autowired-from-XML dataSource
        return new JdbcFooRepository(dataSource);
    }

    @Bean
    public FooService fooService() {
        return new FooServiceImpl(fooRepository());
    }
}
```

Tip

Regardless of the bootstrapping mechanism used - `ConfigurationPostProcessor` or `@ImportXml` - bean references may always be bi-directional. XML-defined beans may reference `@Configuration`-defined beans and vice-versa.

6.2. JavaConfig and Annotation-Driven Configuration

`@AnnotationDrivenConfig`

Spring 2.5 introduced a new style of dependency injection with *Annotation-Driven Injection*. In Spring XML, Annotation-Driven Injection is enabled in the container by declaring

```
<context:annotation-config/>
```

In JavaConfig, this same functionality is enabled with the `@AnnotationDrivenConfig` annotation

```
@Configuration
@AnnotationDrivenConfig
public class Config {
    // may now use @Autowired to reference beans from other @Configuration classes, XML, etc
}
```

`@ComponentScan`

An equivalent for Spring XML's `<context:component-scan/>` is provided with the `@ComponentScan` annotation.

```
package com.company.foo;

@Service
public class FooServiceImpl implements FooService {
    private final FooRepository fooRepository;

    @Autowired
    public FooService(FooRepository fooRepository) {
        this.fooRepository = fooRepository;
    }

    // ...
}
```

```
package com.company.foo;

@Repository
public class JdbcFooRepository implements FooRepository {
    private final DataSource dataSource;

    @Autowired
    public FooRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    // ...
}
```

```
@Configuration
@ComponentScan("com.company") // search the com.company package for @Component classes
@ImportXml("classpath:com/company/data-access-config.xml") // XML with DataSource bean
public class Config {
}
```

Because Spring's `@Service` and `@Repository` stereotype annotations are each meta-annotated with `@Component`, they are candidates for component scanning. Because `FooServiceImpl` and `JdbcFooRepository` both reside underneath the `com.company` package, they will be discovered during component scanning and will be autowired together. `@ImportXml` pulls in the `DataSource` bean, ensuring it will be available for autowiring into `JdbcFooRepository`.

With the above very minimal configuration in the `Config` class, we can bootstrap and use the application as follows:

```
public class Main {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx = new JavaConfigApplicationContext(Config.class);
        FooService fooService = ctx.getBean(FooService.class);
        fooService.doStuff();
    }
}
```

Please see "Chapter 3, IoC" of the core spring documentation for additional detail on Annotation-Driven Injection support.

7. Transaction-management support

7.1. @AnnotationDrivenTx

JavaConfig provides full support for the annotation-driven declarative transaction management features provided by the core Spring Framework with the `@AnnotationDrivenTx` annotation:

```
public class FooServiceImpl implements FooService {
    @Transactional
    public void doStuff() {
        // invoke multiple calls to data access layer
    }
}
```

```
@Configuration
@AnnotationDrivenTx
public class Config {
    @Bean
    public FooService fooService() {
        return new FooServiceImpl(fooRepository());
    }

    @Bean
    public FooRepository fooRepository() {
        return new JdbcFooRepository(dataSource());
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManeger(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        // create and return a new JDBC DataSource ...
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx = new JavaConfigApplicationContext(Config.class);

        // The FooService retrieved from the container will be proxied for tx management
        FooService fooService = ctx.getBean(FooService.class);

        // call the @Transactional method on the proxy - transactional behavior is guaranteed
        fooService.doStuff();
    }
}
```

Like Spring XML's `<tx:annotation-driven>` element, `@AnnotationDrivenTx` expects the presence of a bean named `transactionManager` of type `PlatformTransactionManager` as in the example above. Should you wish to forego this convention and name a transaction manager bean another name, you may do as follows:

```
@Configuration
@AnnotationDrivenTx(transactionManager="txManager") // specify explicitly the bean to use
public class Config {
    @Bean
    public PlatformTransactionManager txManager() {
        return new DataSourceTransactionManeger(dataSource());
    }

    // other beans...
}
```

The other attributes available to the `@AnnotationDrivenTx` are similar to the attributes to the `<tx:annotation-driven>` element. See the related documentation and the JavaDoc for `@AnnotationDrivenTx` for details.

8. AOP support

JavaConfig focuses its AOP support on AspectJ 5 `@Aspect`-style aspects.

8.1. @AspectJAutoProxy

Fashioned after Spring XML's `<aop:aspectj-autoproxy>`, `@AspectJAutoProxy` detects any `@Aspect` beans and generates proxies as appropriate to weave the advice methods in those aspects against other beans in the container.

```
/**
 * An aspect that logs a message before any property (javabeans setter method) is invoked
 */
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(PropertyChangeTracker.class);

    @Before("execution(void set*(*))")
    public void trackChange() {
        logger.info("property about to change");
    }
}
```

```
/**
 * A class with setter methods
 */
public class SimpleCache implements Cache {
    private int size = 100; // default value;
    private DataSource dataSource;

    public void setCacheSize(int size) {
        this.size = size;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

```
/**
 * A @Configuration class that wires up TransferService and applies
 * the PropertyChangeTracker aspect.
 */
@Configuration
@AspectJAutoProxy
public class Config {

    // declare the aspect itself as a bean
    @Bean
    public PropertyChangeTracker propertyChangeTracker() {
        return new PropertyChangeTracker();
    }
```

```
    @Bean
    public Cache cache() {
        return new SimpleCache();
    }
}
```

```
/**
 * A main method to bootstrap the application
 */
public class Main {
    public static void main(String[] args) {
        JavaConfigApplicationContext ctx = new JavaConfigApplicationContext(Config.class);
        Cache cache = ctx.getBean(Cache.class);
        // should see "property about to change" message in the log when calling this line..
        cache.setCacheSize(1000);
    }
}
```

Note

The attributes to the `@AspectJAutoProxy` annotation are very similar to the attributes to the `<aop:aspectj-autoproxy>` element. See the related documentation and the JavaDoc for `@AspectJAutoProxy` for details.

`@Aspect`-annotated `@Configuration` classes

As a convenience, `@Configuration` classes may themselves be annotated with `@Aspect`, allowing for inline advice methods. Modifying the example above, the `Config` class could be rewritten as follows, and `PropertyChangeTracker` could be eliminated entirely:

```
@Aspect
@Configuration
@AspectJAutoProxy
public class Config {
    private Logger logger = Logger.getLogger(Config.class);

    // declare the advice method locally, eliminating the need for PropertyChangeTracker
    @Before("execution(void set*(*))")
    public void trackChange() {
        logger.info("property about to change");
    }

    @Bean
    public Cache cache() {
        return new SimpleCache();
    }
}
```

9. JMX support

9.1. `@MBeanExport`

As an equivalent to Spring XML's `<context:mbean-export/>` element, JavaConfig provides the `@MBeanExport` annotation.

```
/**
 * A performance monitor that implements a JMX MBean interface
 */
public class PerformanceMonitor implements PerformanceMonitorMBean {
    public int getHitCount() {
        return hitCount;
    }
    // ...
}

public interface PerformanceMonitorMBean {
    int getHitCount();
    // ...
}
```

```
/**
 * A @Configuration class that wires up a PerformanceMonitor bean and ensures
 * it is registered with the local MBean server using @MBeanExport
 */
@Configuration
@MBeanExport
public class Config {
    // declare the MBean class as a spring bean
    @Bean
    public PerformanceMonitor performanceMonitor() {
        return new PerformanceMonitor();
    }
}
```

Note

Like with `<context:mbean-export/>`, MBean classes can be defined in a number of ways - using traditional MBean interfaces as shown above, or with Spring's `@ManagedResource` and associated annotations, etc. See the core Spring Framework documentation on JMX support for details.

10. Testing support

10.1. `JavaConfigContextLoader`

The `TestContext framework`, first released with Spring 2.5 provides comprehensive support for system testing using the JUnit and TestNG testing frameworks. JavaConfig provides integration with the TestContext framework via `JavaConfigContextLoader`.

```
package com.bank;

/**
 * Configures the TransferService application
 */
@Configuration
public class TransferAppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository());
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        // create and return a new DataSource
    }
}
```

```
/**
 * System tests for the TransferService application
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="com.bank.TransferAppConfig",
                      loader=JavaConfigContextLoader.class)
public class TransferServiceTests {
    // autowire in the beans to test from the TransferServiceApp application context
    @Autowired TransferService transferService;

    SimpleJdbcTemplate jdbcTemplate;

    // DataSource parameter gets autowired from the TransferServiceApp context
    @Autowired
    void initJdbcTemplate(DataSource dataSource) {
        this.jdbcTemplate = new SimpleJdbcTemplate(dataSource);
    }

    @Test
    public void testTransferFunds() {
        String balQuery = "select balance from accounts where id=?";

        // execute assertions against the jdbcTemplate that
        // prove initial conditions are correct
        assertThat(jdbcTemplate.queryForObject(balQuery, double.class, "2", equalTo(100.00d);
```

```
        // transfer 300 dollars from account with id 1 to account with id 2
        transferService.transfer(300.00d, "1", "2");

        // execute assertions against the jdbcTemplate that
        // the database has been updated properly.
        assertThat(jdbcTemplate.queryForObject(balQuery, double.class, "2", equalTo(400.00d);
    }

    // additional @Test methods...
}
```

Note

Unfortunately, due to compatibility constraints with the TestContext framework's `@ContextConfiguration` *annotation, the fully-qualified classname for* `com.bank.TransferAppConfig` *must be expressed as a string. This has a negative effect on refactorability, and will be improved in the future if possible. Vote for SJC-238 if this improvement is important to you.*

11. Extending JavaConfig

11.1. @Plugin

Similar to Spring 2.0's support for XML namespaces, JavaConfig provides an extensibility mechanism with the `@Plugin` annotation. The general intent is to allow for a more expressive and declarative way to register commonly used bean definitions.

To get a sense of what `@Plugin` can do, consider each of the following annotations already discussed in this document: `@AnnotationDrivenConfig`, `@AnnotationDrivenTx`, `@MBeanExport`, `@PropertiesValueSource`. These annotations are all "`@Plugin` annotations". Take `AnnotationDrivenConfig` for example:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Plugin(handler=AnnotationDrivenConfigHandler.class)
public @interface AnnotationDrivenConfig { }
```

`@AnnotationDrivenConfig` is 'meta-annotated' as `@Plugin`. Notice the `handler` attribute to `@Plugin` accepts `AnnotationDrivenConfigHandler.class`:

```
class AnnotationDrivenConfigHandler implements ConfigurationPlugin<AnnotationDrivenConfig> {

    public void handle(AnnotationDrivenConfig annotation, BeanDefinitionRegistry registry) {
        AnnotationConfigUtils.registerAnnotationConfigProcessors(registry);
    }

}
```

The `handle` method of any `ConfigurationPlugin` implementation is provided with the annotation (`AnnotationDrivenConfig` in this case), as well as a `BeanDefinitionRegistry` - this registry is the same registry that JavaConfig is using to register all objects created from `@Bean` methods. Therefore, `ConfigurationPlugin` implementations have direct control over the container and can do with it as they please. Any number and type of bean definitions may be registered, offloading tedious or repetitive work from the `@Configuration` class author.

Note

This documentation is preliminary and the `@Plugin`/ `ConfigurationPlugin` API may change before JavaConfig's GA release. To find out more about programming `@Plugin` annotations, study the existing set of annotations mentioned above, and don't hesitate to interact with us via the JavaConfig forum.