SPRING

@Configuration is placed on configuration classes, classes that declare beans

@Bean is used on methods to declare beans

In a Spring application an application context loads bean definitions and wires them together

@Aspect is used on a class to declare it as an aspect

@Pointcut is used on a method to specify the target method

@Before specifies what method gets invoked before the target method – before advice

@After specifies what method gets invoked after the target method – after advice

@EnableAspectJAutoProxy needs to be applied on the configuration class in order for aspects to work

Application context :

Spring comes with several flavors of application context. Here are a few that you’ll

most likely encounter:

AnnotationConfigApplicationContext—Loads a Spring application context

from one or more Java-based configuration classes

AnnotationConfigWebApplicationContext—Loads a Spring web application

context from one or more Java-based configuration classes

ClassPathXmlApplicationContext—Loads a context definition from one or

more XML files located in the classpath, treating context-definition files as classpath

resources

FileSystemXmlApplicationContext—Loads a context definition from one or

more XML files in the filesystem

XmlWebApplicationContext—Loads context definitions from one or more

XML files contained in a web application

http://projects.spring.io/spring-webflow/.

<http://docs.spring.io/spring-ws/site/>

projects.spring.io/spring-security/

<http://projects.spring.io/spring-integration>

[www.manning.com/fisher/](http://www.manning.com/fisher/)

[www.manning.com/templier/](http://www.manning.com/templier/)

<http://projects.spring.io/spring-batch/>

<https://spring.io/guides/gs/accessing-facebook/>

<https://spring.io/guides/gs/accessing-twitter/>

http://projects.spring.io/spring-android/

Page 84

Aspect oriented programming: *cross-cutting concerns*

Dhanji R. Prasanna’s *Dependency Injection*

Wiring :

Three ways of wiring :

* Explicit configuration in XML
* Explicit configuration in Java
* Implicit bean discovery and automatic wiring

Automatic wiring :

Component scanning – spring automatically discovers beans to be created in the application context

Autowiring – spring automatically satisfies bean dependencies

@Component – placed on a class, annotation that identifies the class as a component class and serves as a clue to Spring that a bean should be created for this class, thus eliminating the need to explicitly declare a Spring bean. All beans in a spring application context are given an id, if an id is not explicitly provided spring derives one from the class name

@Component(“somename”)

@Bean(“somename”) / @Bean(name = “somename”)

@Configuration – placed on class, designates a class as a configuration class, bean declarations can be made inside it

@ComponentScan –placed on class, on a configuration class, enables component scanning in Spring. By default it will only scan the same package as the configuration class. It scans the package for classes annotated with @Component and creates a bean. @ComponentScan(“packagename”) or @ComponentScan(basePackages=”packagename”) for explicitly declaring a package, or when you need multiple packages @ComponentScan(basePackages = {“packagename1”, “packagename2”}) , the more typesafeoption is specifying the packages by classes that they contain @ComponentScan(basePackageClasses={Class1.class, Class2.class})

@RunWith(SpringJUnit4ClassRunner.class) – placed on test class, used to have a Spring application context automatically created when the test starts and @ContextConfiguration tells it to load its configuration from a specific configuration class.

Autowiring is a means of letting spring automatically satisfy a bean’s dependencies by finding other beans in the application context that are a match to the bean’s needs.

@Autowired can be placed on constructor, setter methods– indicates that autowiring should be performed

If no match if found an error is thrown, to avoid this: @Autowired(required = false)

@Bean

public CDPlayer cdPlayer() {

return new CDPlayer(sgtPeppers());

}

The cdPlayer() method, like the sgtPeppers() method, is annotated with @Bean to

indicate that it will produce an instance of a bean to be registered in the Spring application

context. The ID of the bean will be cdPlayer, the same as the method’s name.

The body of the cdPlayer() method differs subtly from that of the sgtPeppers()

method. Rather than construct an instance via its default method, the CDPlayer

instance is created by calling its constructor that takes a CompactDisc.

It appears that the CompactDisc is provided by calling sgtPeppers, but that’s not

exactly true. Because the sgtPeppers() method is annotated with @Bean, Spring will

intercept any calls to it and ensure that the bean produced by that method is returned

rather than allowing it to be invoked again.

If the call to sgtPeppers() was treated like any other call to a Java method, then each

CDPlayer would be given its own instance of SgtPeppers. That would make sense if we

were talking about real CD players and compact discs. If you have two CD players,

there’s no physical way for a single compact disc to simultaneously be inserted into

two CD players.

In software, however, there’s no reason you couldn’t inject the same instance of

SgtPeppers into as many other beans as you want. By default, all beans in Spring are

singletons, and there’s no reason you need to create a duplicate instance for the second

CDPlayer bean. So Spring intercepts the call to sgtPeppers() and makes sure

that what is returned is the Spring bean that was created when Spring itself called

sgtPeppers() to create the CompactDisc bean. Therefore, both CDPlayer beans will

be given the same instance of SgtPeppers.

In any event, it’s important to recognize that although you’re performing DI via

the CDPlayer’s constructor, there’s no reason you couldn’t apply other styles of DI

here. For example, if you wanted to inject a CompactDisc via a setter method, it might

look like this:

@Bean

public CDPlayer cdPlayer(CompactDisc compactDisc) {

CDPlayer cdPlayer = new CDPlayer(compactDisc);

cdPlayer.setCompactDisc(compactDisc);

return cdPlayer;

}

In Java configuration, you can use the @Profile annotation to specify which profile

a bean belongs to. For example, the embedded database DataSource bean might

be configured in a configuration class like this:

package com.myapp;

import javax.activation.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import

org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;

import

www.it-ebooks.info

***Environments and profiles* 67**

org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration

@Profile("dev")

public class DevelopmentProfileConfig {

@Bean(destroyMethod="shutdown")

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

.addScript("classpath:test-data.sql")

.build();

}

}

The main thing I want to draw your attention to is the @Profile annotation applied at

the class level. It tells Spring that the beans in this configuration class should be created

only if the dev profile is active. If the dev profile isn’t active, then the @Bean methods

will be ignored.

Meanwhile, you may have another configuration class for production that looks

like this:

package com.myapp;

import javax.activation.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration

@Profile("prod")

public class ProductionProfileConfig {

@Bean

public DataSource dataSource() {

JndiObjectFactoryBean jndiObjectFactoryBean =

new JndiObjectFactoryBean();

jndiObjectFactoryBean.setJndiName("jdbc/myDS");

jndiObjectFactoryBean.setResourceRef(true);

jndiObjectFactoryBean.setProxyInterface(

javax.sql.DataSource.class);

return (DataSource) jndiObjectFactoryBean.getObject();

}

}

In this case, the bean won’t be created unless the prod profile is active.

In Spring 3.1, you could only use the @Profile annotation at the class level. Starting

with Spring 3.2, however, you can use @Profile at the method level, alongside the

@Bean annotation. This makes it possible to combine both bean declarations into a

single configuration class, as shown in the following listing.

www.it-ebooks.info

**68** CHAPTER 3 ***Advanced wiring***

package com.myapp;

import javax.activation.DataSource;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Profile;

import

org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;

import

org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration

public class DataSourceConfig {

@Bean(destroyMethod="shutdown")

@Profile("dev")

public DataSource embeddedDataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScript("classpath:schema.sql")

.addScript("classpath:test-data.sql")

.build();

}

@Bean

@Profile("prod")

public DataSource jndiDataSource() {

JndiObjectFactoryBean jndiObjectFactoryBean =

new JndiObjectFactoryBean();

jndiObjectFactoryBean.setJndiName("jdbc/myDS");

jndiObjectFactoryBean.setResourceRef(true);

jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);

return (DataSource) jndiObjectFactoryBean.getObject();

}

}

What’s not apparent here is that although each of the DataSource beans is in a profile

and will only be created if the prescribed profile is active, there are probably other

beans that aren’t defined in the scope of a given profile. Any bean that isn’t given a

profile will always be created, regardless of what profile is active.

***Activating profiles***

Spring honors two separate properties when determining which profiles are active:

spring.profiles.active and spring.profiles.default. If spring.profiles.active

is set, then its value determines which profiles are active. But if spring

.profiles.active isn’t set, then Spring looks to spring.profiles.default. If neither

spring.profiles.active nor spring.profiles.default is set, then there are no

active profiles, and only those beans that aren’t defined as being in a profile are created.

There are several ways to set these properties:

 As initialization parameters on DispatcherServlet

 As context parameters of a web application

 As JNDI entries

 As environment variables

 As JVM system properties

 Using the @ActiveProfiles annotation on an integration test class

I’ll leave it to you to choose the best combination of spring.profiles.active and

spring.profiles.default to suit your needs.

One approach that I like is to set spring.profiles.default to the development

profile using parameters on DispatcherServlet and in the servlet context (for the

sake of ContextLoaderListener). For example, a web application’s web.xml file

might set spring.profiles.default as shown in the next listing.

<?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">

Listing 3.3 Setting default profiles in a web application’s web.xml file

**“prod” profile beans**

www.it-ebooks.info

***Environments and profiles* 71**

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<context-param>

<param-name>spring.profiles.default</param-name>

<param-value>dev</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>

org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>spring.profiles.default</param-name>

<param-value>dev</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>

With spring.profiles.default set this way, any developer can retrieve the application

code from source control and run it using development settings (such as an

embedded database) without any additional configuration.

Then, when the application is deployed in a QA, production, or other environment,

the person responsible for deploying it can set spring.profiles.active using

system properties, environment variables, or JNDI as appropriate. When spring

.profiles.active is set, it doesn’t matter what spring.profiles.default is set to;

the profiles set in spring.profiles.active take precedence.

You’ve probably noticed that the word *profiles* is plural in spring.profiles.active

and spring.profiles.default. This means you can activate multiple profiles at the

same time by listing the profile names, separated by commas. Of course, it probably

doesn’t make much sense to enable both dev and prod profiles at the same time, but

you could enable multiple orthogonal profiles simultaneously.

TESTING WITH PROFILES

When running an integration test, you’ll often want to test using the same configuration

(or some subset thereof) you’d use in production. But if your configuration

**Set default profile**

**for context**

**Set default profile**

**for servlet**

www.it-ebooks.info

**72** CHAPTER 3 ***Advanced wiring***

references beans that are in profiles, you need a way to enable the appropriate profile

when running those tests.

Spring offers the @ActiveProfiles annotation to let you specify which profile(s)

should be active when a test is run. Often it’s the development profile that you’ll want

to activate during an integration test. For example, here’s a snippet of a test class that

uses @ActiveProfiles to activate the dev profile:

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes={PersistenceTestConfig.class})

@ActiveProfiles("dev")

public class PersistenceTest {

...

}

Spring profiles are a great way to conditionally define beans where the condition is

based on which profile is active. But Spring 4 offers a more general-purpose mechanism

for conditional bean definitions where the condition is up to you. Let’s see how

to define conditional beans using Spring 4 and the @Conditional annotation.

Suppose you want one or more beans to be configured if and only if some library is

available in the application’s classpath. Or let’s say you want a bean to be created only

if a certain other bean is also declared. Maybe you want a bean to be created if and

only if a specific environment variable is set.

Until Spring 4, it was difficult to achieve this level of conditional configuration, but

Spring 4 introduced a new @Conditional annotation that can be applied to @Bean

methods. If the prescribed condition evaluates to true, then the bean is created. Otherwise

the bean is ignored.

For example, suppose you have a class named MagicBean that you only want Spring

to instantiate if a magic environment property has been set. If the environment has no

such property, then the MagicBean should be ignored. The following listing shows a

configuration that conditionally configures the MagicBean using @Conditional.

@Bean

@Conditional(MagicExistsCondition.class)

public MagicBean magicBean() {

return new MagicBean();

}

As you can see, @Conditional is given a Class that specifies the condition—in this

case, MagicExistsCondition. @Conditional comes paired with a Condition interface:

public interface Condition {

boolean matches(ConditionContext ctxt,

AnnotatedTypeMetadata metadata);

}

Listing 3.4 Conditionally configuring a bean

**Conditionally create bean**

www.it-ebooks.info

***Conditional beans* 73**

The class given to @Conditional can be any type that implements the Condition

interface. As you can see, it’s a straightforward interface to implement, requiring only

that you provide an implementation for the matches() method. If the matches()

method returns true, then the @Conditional-annotated beans are created. If

matches() returns false, then those beans aren’t created.

For this example, you need to create an implementation of Condition that hinges

its decision on the presence of a magic property in the environment. The next listing

shows MagicExistsCondition, an implementation of Condition that does the trick.

package com.habuma.restfun;

import org.springframework.context.annotation.Condition;

import org.springframework.context.annotation.ConditionContext;

import org.springframework.core.type.AnnotatedTypeMetadata;

import org.springframework.util.ClassUtils;

public class MagicExistsCondition implements Condition {

public boolean matches(

ConditionContext context, AnnotatedTypeMetadata metadata) {

Environment env = context.getEnvironment();

return env.containsProperty("magic");

}

}

The matches() method in this listing is simple but powerful. It uses the Environment

obtained from the given ConditionContext object to check for the presence of an

environment property named magic. For this example, the value of the property is

irrelevant; it only needs to exist. This results in true being returned from matches().

Consequently, the condition is met, and any beans whose @Conditional annotation

refers to MagicExistsCondition will be created.

On the other hand, if the property doesn’t exist, the condition will fail, false will

be returned from matches(), and none of those beans will be created.

MagicExistsCondition only uses the Environment from the ConditionContext,

but there’s much more that a Condition implementation can consider. The

matches() method is given a ConditionContext and an AnnotatedTypeMetadata to

use in making its decision.

AMBIGUITY IN AUTOWIRING

For those times when ambiguity does happen, however, Spring offers a couple of

options. You can declare one of the candidate beans as the primary choice, or you can

use qualifiers to help Spring narrow its choices to a single candidate.

When declaring beans, you can avoid autowiring ambiguity by designating one of

the candidate beans as a primary bean. In the event of any ambiguity, Spring will

choose the primary bean over any other candidate beans. Essentially, you’re declaring

your “favorite” bean.

Let’s say that ice cream is your favorite dessert. You can express that favorite choice

in Spring using the @Primary annotation. @Primary can be used either alongside

@Component for beans that are component-scanned or alongside @Bean for beans

declared in Java configuration. For example, here’s how you might declare the

@Component-annotated IceCream bean as the primary choice:

@Component

@Primary

public class IceCream implements Dessert { ... }

Or, if you’re declaring the IceCream bean explicitly in Java configuration, the @Bean

method might look like this:

@Bean

@Primary

public Dessert iceCream() {

return new IceCream();

}

In contrast, Spring’s qualifiers apply a narrowing operation to all candidate beans,

ultimately arriving at the single bean that meets the prescribed qualifications. If ambiguity

still exists after applying all qualifiers, you can always apply more qualifiers to

narrow the choices further.

The @Qualifier annotation is the main way to work with qualifiers. It can be

applied alongside @Autowired or @Inject at the point of injection to specify which

bean you want to be injected. For example, let’s say you want to ensure that the

IceCream bean is injected into setDessert():

@Autowired

@Qualifier("iceCream")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

This is a prime example of qualifiers in their simplest form. The parameter given to

@Qualifier is the ID of the bean that you want to inject. All @Component-annotated

classes will be created as beans whose ID is the uncapitalized class name. Therefore,

@Qualifier("iceCream") refers to the bean created when component-scanning created

an instance of the IceCream class.

Actually, there’s a bit more to the story than that. To be more precise, @Qualifier

("iceCream") refers to the bean that has the String “iceCream” as a qualifier. For

www.it-ebooks.info

**78** CHAPTER 3 ***Advanced wiring***

lack of having specified any other qualifiers, all beans are given a default qualifier

that’s the same as their bean ID. Therefore, the setDessert() method will be injected

with the bean that has “iceCream” as a qualifier. That just happens to be the bean

whose ID is iceCream, created when the IceCream class was component-scanned.

Basing qualification on the default bean ID qualifier is simple but can pose some

problems. What do you suppose would happen if you refactored the IceCream class,

renaming it Gelato? In that case, the bean’s ID and default qualifier would be gelato,

which doesn’t match the qualifier on setDessert(). Autowiring would fail.

The problem is that you specified a qualifier on setDessert() that is tightly coupled

to the class name of the bean being injected. Any change to that class name will

render the qualifier ineffective.

CREATING CUSTOM QUALIFIERS

Instead of relying on the bean ID as the qualifier, you can assign your own qualifier to

a bean. All you need to do is place the @Qualifier annotation on the bean declaration.

For example, it can be applied alongside @Component like this:

@Component

@Qualifier("cold")

public class IceCream implements Dessert { ... }

In this case, a qualifier of cold is assigned to the IceCream bean. Because it’s not coupled

to the class name, you can refactor the name of the IceCream class all you want

without worrying about breaking autowiring. It will work as long as you refer to the

cold qualifier at the injection point:

@Autowired

@Qualifier("cold")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

It’s worth noting that @Qualifier can also be used alongside the @Bean annotation

when explicitly defining beans with Java configuration:

@Bean

@Qualifier("cold")

public Dessert iceCream() {

return new IceCream();

}

solution is to tack on another @Qualifier at both the injection point

and at the bean definition. Maybe the IceCream class could look like this:

@Component

@Qualifier("cold")

@Qualifier("creamy")

public class IceCream implements Dessert { ... }

Perhaps the Popsicle class could also use another @Qualifier:

@Component

@Qualifier("cold")

@Qualifier("fruity")

public class Popsicle implements Dessert { ... }

And at the injection point, you could narrow it down to IceCream like this:

@Autowired

@Qualifier("cold")

@Qualifier("creamy")

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

There’s only one small problem: Java doesn’t allow multiple annotations of the same

type to be repeated on the same item.1 The compiler will complain with errors if you

try this. There’s no way you can use @Qualifier (at least not directly) to narrow the

list of autowiring candidates to a single choice.

What you can do, however, is create custom qualifier annotations to represent the

traits you want your beans to be qualified with. All you have to do is create an annotation

that is itself annotated with @Qualifier. Rather than use @Qualifier("cold"),

you can use a custom @Cold annotation that’s defined like this:

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,

ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Cold { }

1 Java 8 allows repeated annotations, as long as the annotation is annotated with @Repeatable. Even so,

Spring’s @Qualifier annotation isn’t annotated with @Repeatable.

www.it-ebooks.info

**80** CHAPTER 3 ***Advanced wiring***

Likewise, you can create a new @Creamy annotation as a replacement for @Qualifier

("creamy"):

@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,

ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Creamy { }

And, similarly, you can create @Soft, @Crispy, and @Fruity annotations to use wherever

you’d otherwise use the @Qualifier annotation. By annotating these annotations

with @Qualifier, they take on the characteristics of @Qualifier. They are, in fact,

qualifier annotations in their own right.

Now you can revisit IceCream and annotate it with @Cold and @Creamy, like this:

@Component

@Cold

@Creamy

public class IceCream implements Dessert { ... }

Similarly, the Popsicle class can be annotated with @Cold and @Fruity:

@Component

@Cold

@Fruity

public class Popsicle implements Dessert { ... }

Finally, at the injection point, you can use any combination of qualifier annotations

necessary to narrow the selection to the one bean that meets your specifications. To

arrive at the IceCream bean, the setDessert() method can be annotated like this:

@Autowired

@Cold

@Creamy

public void setDessert(Dessert dessert) {

this.dessert = dessert;

}

Spring defines several scopes under which a bean can be created, including the

following:

 *Singleton*—One instance of the bean is created for the entire application. This is the default

 *Prototype*—One instance of the bean is created every time the bean is injected

into or retrieved from the Spring application context.

 *Session*—In a web application, one instance of the bean is created for each session.

 *Request*—In a web application, one instance of the bean is created for each

request.

Singleton scope is the default scope, but as we’ve discussed, it isn’t ideal for mutable

types. To select an alternative type, you can use the @Scope annotation, either in conjunction

with the @Component annotation or with the @Bean annotation.

For example, if you’re relying on component-scanning to discover and declare a

bean, then you can annotate the bean class with @Scope to make it a prototype bean:

@Component

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public class Notepad { ... }

Here, you specify prototype scope by using the SCOPE\_PROTOTYPE constant from the

ConfigurableBeanFactory class. You could also use @Scope("prototype"), but using

the SCOPE\_PROTOTYPE constant is safer and less prone to mistakes.

Alternatively, if you’re configuring the Notepad bean as a prototype in Java configuration,

you can use @Scope along with @Bean to specify the desired scoping:

@Bean

@Scope(ConfigurableBeanFactory.SCOPE\_PROTOTYPE)

public Notepad notepad() {

return new Notepad();

}