A **use case** represents the reason a person uses the app. For example, in a ridesharing app, a use case is “requesting a car.” For an app managing food delivery, a use case is “ordering a pizza.”

We have **Spring Data Access**, which is a module of Spring Core, and we also have an independent project in the Spring ecosystem named Spring Data. Spring Data Access contains fundamental data access implementations like the transaction mechanism and JDBC tools. **Spring Data** enhances access to databases and offers a broader set of tools, which makes development more accessible and enables your app to connect to different kinds of data sources.

**Alternatives:**

For example, let’s take the **Spring IoC container**. Years ago, the Java EE specification was a solution very much appreciated by the developers. With a slightly different philosophy, Java EE (which in 2017 was open sourced and remade in Jakarta EE, <https://jakarta.ee> /) offered specifications like Context and Dependency Injection (CDI) or Enterprise Java Beans (EJB). You could use CDI or EJB to manage a context of object instances and implement aspects (named “interceptors” in the EE terminology). Also, Google Guice (<https://github.com/google/guice> ) was an appreciated framework for the management of object instances in a container.

For some of the projects taken individually, you could find one or more alternatives. For example, you could choose to use Apache Shiro (<https://shiro.apache.org/> ) instead of **Spring Security**. Or you could decide to implement your web app using the Play framework (<https://www.playframework.com> /) instead of **Spring MVC** and Spring-related technologies.

**When to use Spring in real-world scenarios**

1. The development of a backend app
2. The development of an automation testing framework
3. The development of a desktop app
4. The development of a mobile app

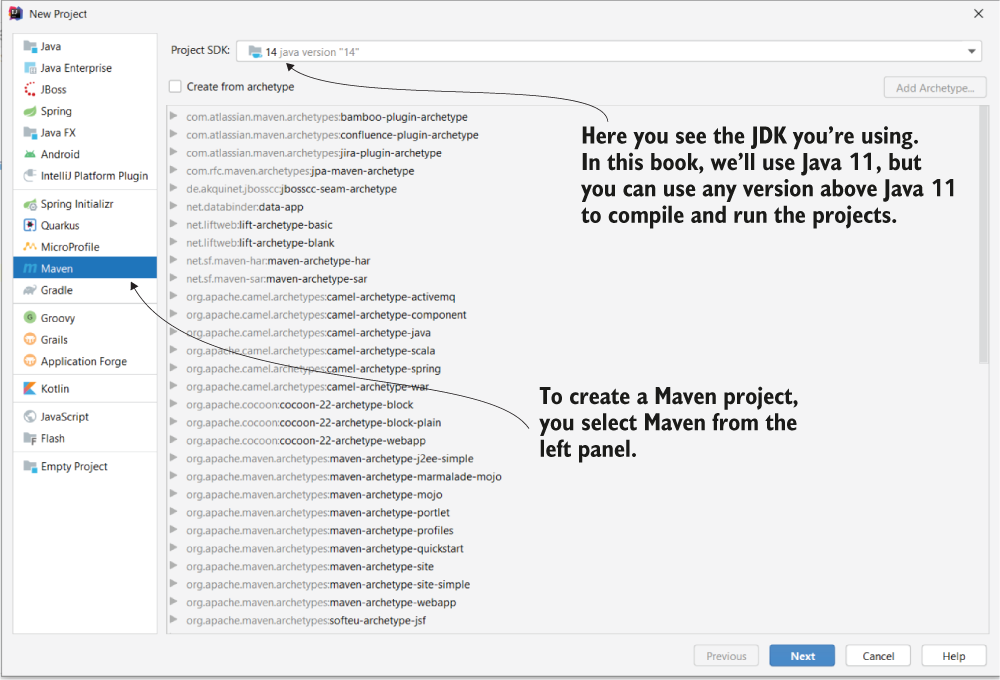
**When not to use framework**

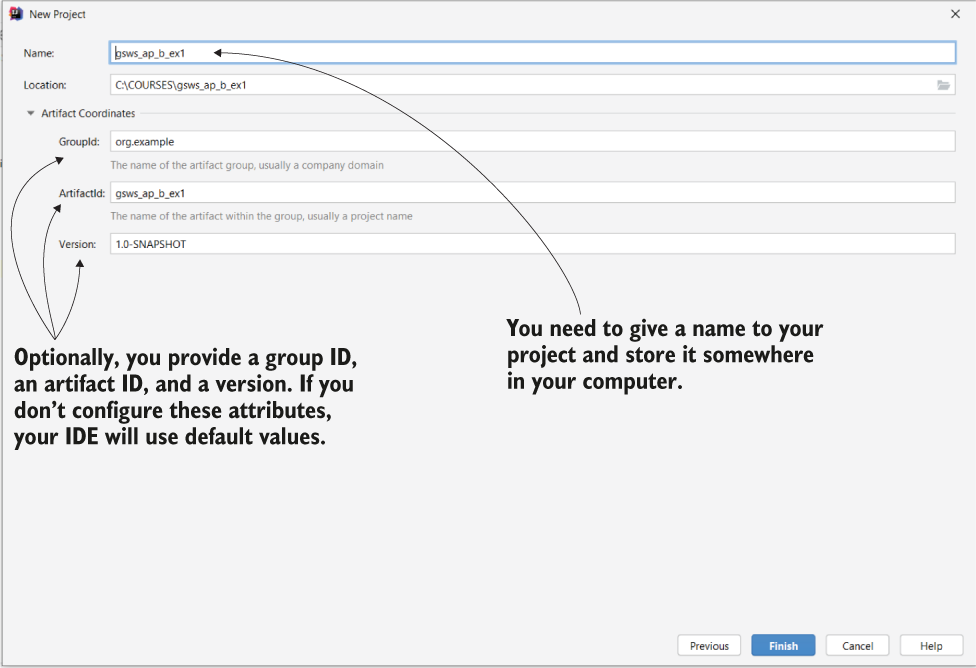
1. You need to implement a particular functionality with a footprint as small as possible. By footprint, I mean the storage memory occupied by the app’s files.
2. Specific security requirements force you to implement only custom code in your app without making use of any open source framework.
3. You’d have to make so many customizations over the framework that you’d write more code than if you’d simply not used it at all.
4. You already have a functional app, and by changing it to use a framework you don’t gain any benefit.

# Chapter 2 Spring Context

## **PROJECT 01 – Add bean to Spring context(v.1 - @Bean)**

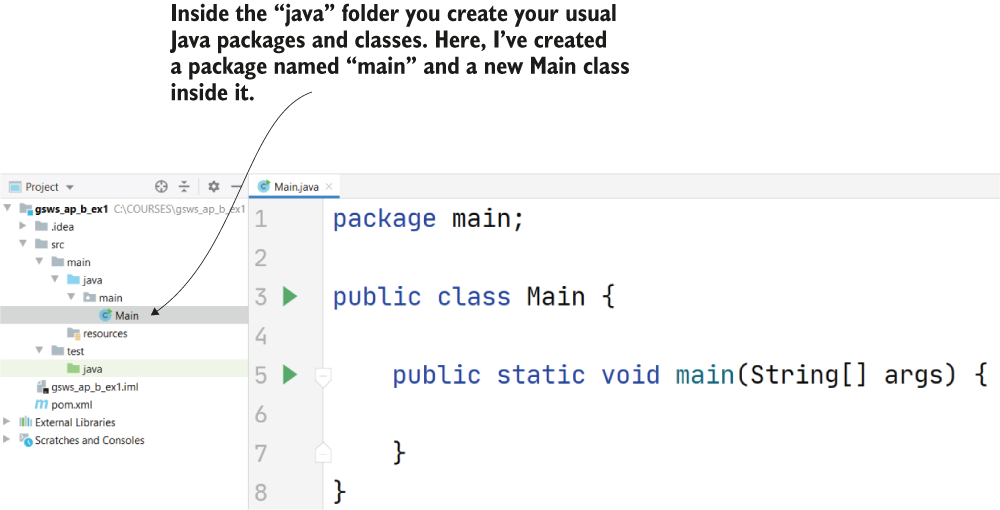
Let’s start by creating a new project. You create a new project in IntelliJ from File > New > Project. This will get you to a window like the one in figure 2.1.

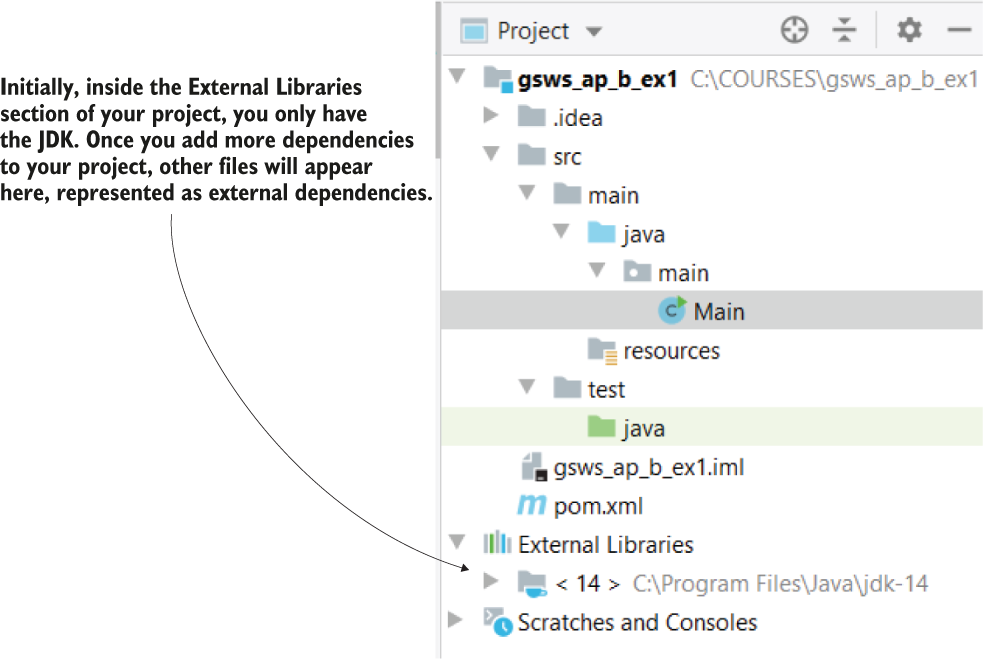




Once you’ve selected the type of your project, in the next window (figure 2.2) you need to give it a name. In addition to the project name and choosing the location in which to store it, for a Maven project you can also specify the following:

* A **group ID**, which we use to group multiple related projects
* An **artifact ID**, which is the name of the current application
* A **version**, which is an identifier of the current implementation state





Listing 2.2 Adding a new dependency in the 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>org.example</groupId>

<artifactId>sq\_ch2\_ex1</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies> ❶

<dependency> ❷

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.2.6.RELEASE</version>

</dependency>

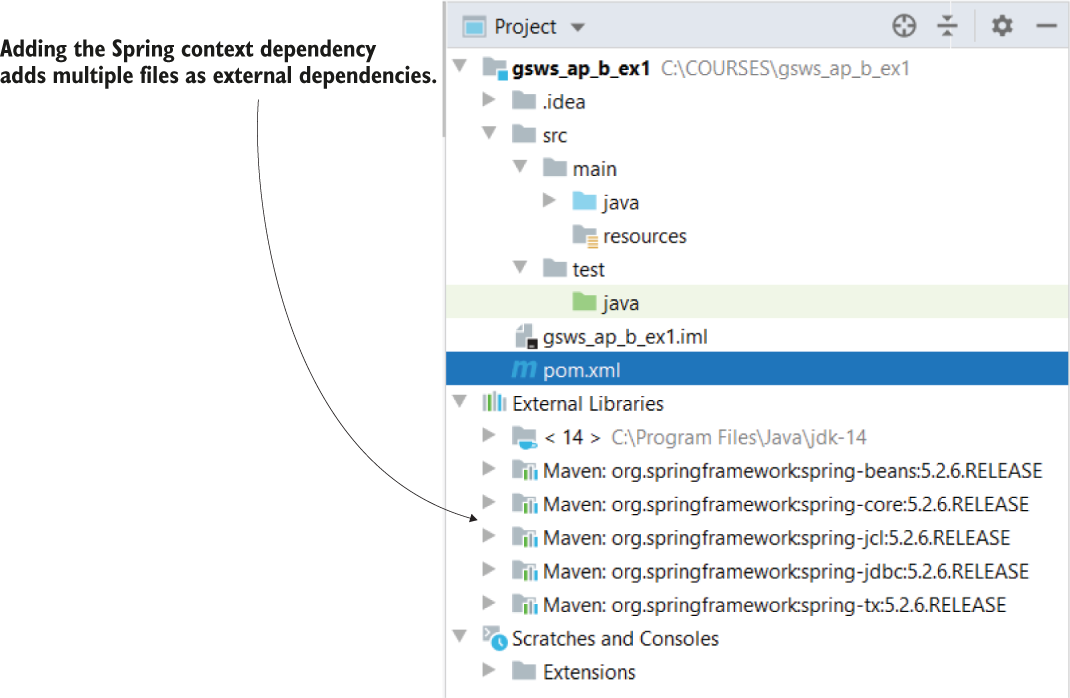
</dependencies>

</project>

❶ You need to write the dependencies for the project between the <dependencies> and </dependecies> tags.

❷ A dependency is represented by a group of <dependency> </dependency> tags.

Once you’ve added the dependency in the pom.xml file, as presented in the previous listing, the IDE downloads them, and you’ll now find these dependencies in the “External Libraries” folder (figure 2.6).



When you add a new dependency to the pom.xml file, Maven downloads the jar files representing thatdependency. You find these jar files in the External Libraries folder of the project.

2.2 Adding new beans to the Spring context

You have multiple ways to add beans in the Spring context such that Spring can manage them and plug features it provides into your app. Depending on the action, you’ll choose a specific way to add the bean. You can add beans in the context in the following ways:

* Using the @Bean annotation
* Using stereotype annotations
* Programmatically

Let’s first create a project with a reference to no framework—not even Spring. We’ll then add the dependencies needed to use the Spring context and create it.

Listing 2.3 The Parrot class

public class Parrot {

private String name;

// Omitted getters and setters

}

You can now define a class containing the main method and create an instance of the class Parrot, as presented in the following listing. I usually name this class Main.

Listing 2.4 Creating an instance of the Parrot class

public class Main {

public static void main(String[] args) {

Parrot p = new Parrot();

}

}

It’s now time to add the needed dependencies to our project. Because we’re using Maven, I’ll add the dependencies in the pom.xml file, as presented in the following listing.

Listing 2.5 Adding the dependency for Spring context

<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>org.example</groupId>

<artifactId>sq-ch2-ex1</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.2.6.RELEASE</version>

</dependency>

</dependencies>

</project>

Listing 2.6 Creating the instance of the Spring context

public class Main {

public static void main(String[] args) {

var context =

new AnnotationConfigApplicationContext(); ❶

Parrot p = new Parrot();

}

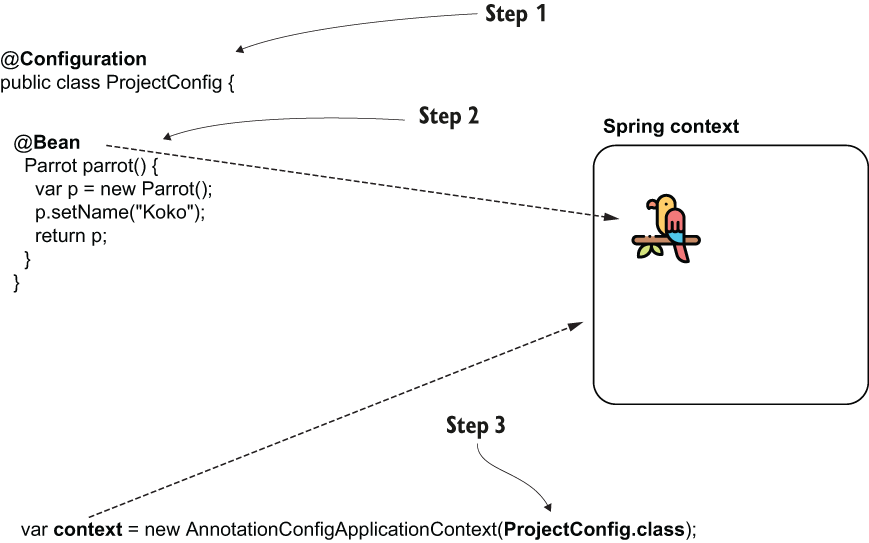
}

❶ Creates an instance of the Spring context

We use the AnnotationConfigApplicationContext class to create the Spring context instance. Spring offers multiple implementations. In most cases you’ll use the AnnotationConfigApplicationContext class - the implementation that uses the most used today’s approach: annotations.

The steps you need to follow to add a bean to the Spring context using the @Bean annotation are as follows :

1. Define a configuration class (annotated with @Configuration) for your project, which, as we’ll discuss later, we use to configure the context of Spring.
2. Add a method to the configuration class that returns the object instance you want to add to the context and annotate the method with the @Bean annotation.
3. Make Spring use the configuration class defined in step 1. As you’ll learn later, we use configuration classes to write different configurations for the framework.



A **configuration** class is a special class in Spring applications that we use to instruct Spring to do specific actions. For example, we can tell Spring to create beans or to enable certain functionalities.

I separate the classes into **different packages** to make the code easier to understand. For example, I create the configuration classes in a package named **config**, and the Main class in a package named **main**. Organizing the classes into packages is a good practice.

Listing 2.8 Defining the @Bean method

@Configuration

public class ProjectConfig {

@Bean ❶

Parrot parrot() {

var p = new Parrot();

p.setName("Koko"); ❷

return p; ❸

}

}

❶ By adding the @Bean annotation, we instruct Spring to call this method when at context initialization and add the returned value to the context.

❷ Set a name for the parrot we’ll use later when we test the app.

❸ Spring adds to its context the Parrot instance returned by the method.

Observe that the name I used for the method doesn’t contain a verb. You probably learned that a Java best practice is to put verbs in method names because the methods generally represent actions. But for methods we use to add beans in the Spring context, we don’t follow this convention. Such methods represent the object instances they return and that will now be part of the Spring context. The method’s name also becomes the bean’s name (as in listing 2.8, the bean’s name is now “parrot”). By convention, you can use nouns, and most often they have the same name as the class.

Listing 2.9 Initializing the Spring context based on the defined configuration class

public class Main {

public static void main(String[] args) {

var context =

new AnnotationConfigApplicationContext(

ProjectConfig.class); ❶

}

}

❶ When creating the Spring context instance, send the configuration class as a parameter to instruct Spring to use it.

To verify the Parrot instance is indeed part of the context now, you can refer to the instance and print its name in the console, as presented in the following listing.

Listing 2.10 Referring to the Parrot instance from the context

public class Main {

public static void main(String[] args) {

var context =

new AnnotationConfigApplicationContext(

ProjectConfig.class);

Parrot p = context.getBean(Parrot.class); ❶

System.out.println(p.getName());

}

}

❶ Gets a reference of a bean of type Parrot from the Spring context

Now you’ll see the name you gave to the parrot you added in the context in the console, in my case *Koko*.

The next listing shows you how I changed the configuration class to also add a bean of type String and a bean of type Integer.

Listing 2.11 Adding two more beans to the context

@Configuration

public class ProjectConfig {

@Bean

Parrot parrot() {

var p = new Parrot();

p.setName("Koko");

return p;

}

@Bean ❶

String hello() {

return "Hello";

}

@Bean ❷

Integer ten() {

return 10;

}

}

❶ Adds the string “Hello” to the Spring context

❷ Adds the integer 10 to the Spring context

You can now refer to these 2 new beans in the same way we did with the parrot.

Listing 2.12 Printing the 2 new beans in the console

public class Main {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext(

ProjectConfig.class);

Parrot p = context.getBean(Parrot.class); ❶

System.out.println(p.getName());

String s = context.getBean(String.class);

System.out.println(s);

Integer n = context.getBean(Integer.class);

System.out.println(n);

}

}

❶ You don’t need to do any explicit casting. Spring looks for a bean of the type you requested in its context. If such a bean doesn’t exist, Spring will throw an exception.

Running the app now, the values of the 3 beans will be printed in the console.

## **PROJECT 01 – add more beans**

Let’s create a new project, “sq-ch2-ex3,” to demonstrate how you can add multiple beans of the same type to the Spring context and how you can refer to them afterward.

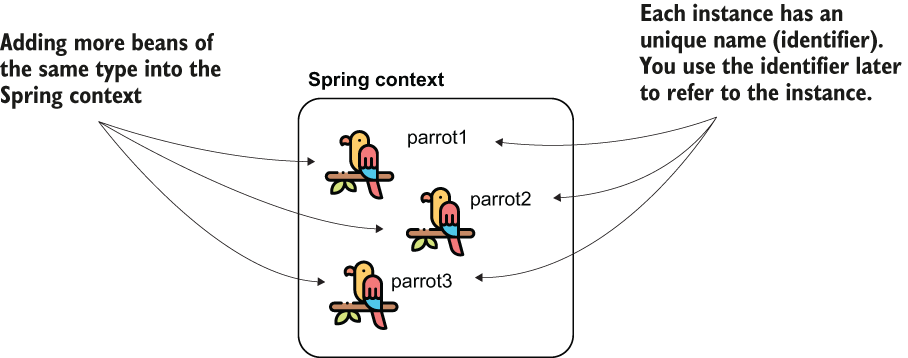


Figure 2.11 You can add more beans of the same type to the Spring context by using multiple methods annotated with @Bean. Each instance will have a unique identifier. To refer to them afterward, you’ll need to use the beans’ identifiers.

NOTE Don’t confuse the name of the bean with the name of the parrot. In our example, the beans’ names (or identifiers) in the Spring context are parrot1, parrot2, and parrot3 (like the name of the @Bean methods defining them). The names I gave to the parrots are Koko, Miki, and Riki. The parrot name is just an attribute of the Parrot object, and it doesn’t mean anything to Spring.

Listing 2.13 Adding multiple beans of the same type to the Spring context

@Configuration

public class ProjectConfig {

@Bean

Parrot parrot1() {

var p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

Parrot parrot2() {

var p = new Parrot();

p.setName("Miki");

return p;

}

@Bean

Parrot parrot3() {

var p = new Parrot();

p.setName("Riki");

return p;

}

}

Of course, you can’t get the beans from the context anymore by only specifying the type. If you do, you’ll get an exception because Spring cannot guess which instance you’ve declared you refer to. Look at the following listing. Running such a code throws an exception in which Spring tells you that you need to be precise, which is the instance you want to use.

Listing 2.14 Referring to a Parrot instance by type

public class Main {

public static void main(String[] args) {

var context = new

AnnotationConfigApplicationContext(ProjectConfig.class);

Parrot p = context.getBean(Parrot.class); ❶

System.out.println(p.getName());

}

}

❶ You’ll get an exception on this line because Spring cannot guess which of the three Parrot instances you refer to.

When running your application, you’ll get an exception similar to the one presented by the next code snippet.

Exception in thread "main" org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'main.Parrot' available: expected single matching bean but found 3:

parrot1,parrot2,parrot3 ❶

at ...

❶ Names of the Parrot beans in the context

To solve this ambiguity problem, you need to refer precisely to 1 of the instances by using the bean’s name. By default, Spring uses the names of the methods annotated with @Bean as the beans’ names themselves.

Listing 2.15 Referring to a bean by its identifier

public class Main {

public static void main(String[] args) {

var context = new

AnnotationConfigApplicationContext(ProjectConfig.class);

Parrot p = context.getBean("parrot2", Parrot.class); ❶

System.out.println(p.getName());

}

}

❶ First parameter is the name of the instance to which we refer

Running the app now, you’ll no longer get an exception. Instead, you’ll see in the console the name of the 2nd parrot, Miki.

If you’d like to give another name to the bean, you can use either one of the name or the value attributes of the @Bean annotation. Any of the following syntaxes will change the name of the bean in "miki":

@Bean(name = "miki")

@Bean(value = "miki")

@Bean("miki")

When you have multiple beans of the same kind in the Spring context you can make one of them primary. You mark the bean you want to be primary using the @Primary annotation. A primary bean is the one Spring will choose if it has multiple options and you don’t specify a name; the primary bean is simply Spring’s default choice. The next code snippet shows you what the @Bean method annotated as primary looks like:

@Bean

@Primary

Parrot parrot2() {

var p = new Parrot();

p.setName("Miki");

return p;

}

## **PROJECT 02 – Add bean to Spring context (v.2 - @Component + @ComponentScan), @PostConstruct**

Spring offers multiple stereotype annotations. We’ll take the most basic of these, **@Component**.

With stereotype annotations, you add the annotation above the class for which you need to have an instance in the Spring context. When doing so, we say that you’ve marked the class as a component. When the app creates the Spring context, Spring creates an instance of the class you marked as a component and adds that instance to its context. We’ll still have a configuration class when we use this approach to tell Spring where to look for the classes annotated with stereotype annotations. Moreover, you can use both the approaches (using @Bean and stereotype annotations together).

The steps we need to follow in the process are as follows:

1. Using the @Component annotation, mark the classes for which you want Spring to add an instance to its context (in our case Parrot).
2. Using **@ComponentScan** annotation over the configuration class, instruct Spring on where to find the classes you marked.



Listing 2.16 Using a stereotype annotation for the Parrot class

@Component ❶

public class Parrot {

private String name;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

❶ By using the @Component annotation over the class, we instruct Spring to create an instance of this class and add it to its context.

By default, Spring doesn’t search for classes annotated with stereotype annotations, so if we just leave the code as-is, Spring won’t add a bean of type Parrot in its context. To tell Spring it needs to search for classes annotated with stereotype annotations, we use the @ComponentScan annotation over the configuration class. Also, with the @ComponentScan annotation, we tell Spring where to look for these classes. We enumerate the packages where we defined the classes with stereotype annotations. In my case, the name of the package is “main.”

Listing 2.17 Using the @ComponentScan annotation to tell Spring where to look

@Configuration

@ComponentScan(basePackages = "main") ❶

public class ProjectConfig {

}

❶ Using the basePackages attribute of the annotation, we tell Spring where to look for classes annotated with stereotype annotations.

Now you told Spring the following:

1. Which classes to add an instance to its context (Parrot)
2. Where to find these classes (using @ComponentScan)

We don’t need methods anymore to define the beans. And it now looks like this approach is better because you achieve the same thing by writing less code. But wait until the end of this chapter. You’ll learn that both approaches are useful, depending on the scenario.

Listing 2.18 Defining the main method to test the Spring configuration

public class Main {

public static void main(String[] args) {

var context = new

AnnotationConfigApplicationContext(ProjectConfig.class);

Parrot p = context.getBean(Parrot.class);

System.out.println(p); ❶

System.out.println(p.getName()); ❷

}

}

❶ Prints the default String representation of the instance taken from the Spring context

❷ Prints *null* because we did not assign any name to the parrot instance added by Spring in its context

Table 2.1 Advantages and disadvantages: A comparison of the 2 ways of adding beans to the Spring context, which tells you when you would use either of them

|  |  |
| --- | --- |
| Using the @Bean annotation | Using stereotype annotations |
| 1. You have full control over the instance creation you add to the Spring context. It is your responsibility to create and configure the instance in the body of the method annotated with @Bean. Spring only takes that instance and adds it to the context as-is. 2. You can use this method to add more instances of the same type to the Spring context. Remember, in section 2.1.1 we added three Parrot instances into the Spring context. 3. You can use the @Bean annotation to add to the Spring context any object instance. The class that defines the instance doesn’t need to be defined in your app. Remember, earlier we added a String and an Integer to the Spring context. 4. You need to write a separate method for each bean you create, which adds boilerplate code to your app. For this reason, we prefer using @Bean as a 2nd option to stereotype annotations in our projects. | 1. You only have control over the instance after the framework creates it. 2. This way, you can only add one instance of the class to the context. 3. You can use stereotype annotations only to create beans of the classes your application owns. For example, you couldn’t add a bean of type String or Integer like we did in section 2.1.1 with the @Bean annotation because you don’t own these classes to change them by adding a stereotype annotation. 4. Using stereotype annotations to add beans to the Spring context doesn’t add boilerplate code to your app. You’ll prefer this approach in general for the classes that belong to your app. |

In real life you’ll only use the @Bean when you can’t add the bean otherwise (e.g., you create the bean for a class that is part of a library so you cannot modify that class to add the stereotype annotation).

Using **@PostConstruct** to manage the instance after its creation

Using @Bean, we were able to define a name for each of the Parrot instances we added to the Spring context, but using @Component, we didn’t get a chance to do something after Spring called the constructor of the Parrot class. What if we want to execute some instructions right after Spring creates the bean? We can use the @PostConstruct annotation.

You just need to define a method in the component class and annotate that method with @PostConstruct, which instructs Spring to call that method after the constructor finishes its execution.

Let’s add to pom.xml the Maven dependency needed to use the @PostConstruct annotation:

<dependency>

<groupId>javax.annotation</groupId>

<artifactId>javax.annotation-api</artifactId>

<version>1.3.2</version>

</dependency>

You don’t need to add this dependency if you use a Java version smaller than Java 11. If you wish to use functionalities that were part of the removed APIs (like @PostConstruct), you now need to explicitly add the dependency in your app.

Now you can define a method in the Parrot class:

@Component

public class Parrot {

private String name;

@PostConstruct

public void init() {

this.name = "Kiki";

}

}

If you now print the name of the parrot in the console, you’ll observe the app prints the value Kiki in the console.

Similarly, but less encountered, you can use the annotation **@PreDestroy**. With this annotation, you define a method that Spring calls immediately before closing and clearing the context. But generally I recommend developers avoid using it and find a different approach to executing something before Spring clears the context, mainly because you can expect Spring to fail to clear the context. Say you defined something sensitive (like closing a database connection) in the @PreDestroy method; if Spring doesn’t call the method, you may get into big problems.

## **PROJECT 03 – Add bean to Spring context (v.3 – registerBean() )**

**Programmatically adding beans to the Spring context**

We’ve had the option of programmatically adding beans to the Spring context with Spring 5, which offers great flexibility because it enables you to add new instances in the context directly by calling a method of the context instance. You’d use this approach when you want to implement a custom way of adding beans to the context and the @Bean or the stereotype annotations are not enough for your needs. With the @Bean and stereotype annotations, you can implement most of the scenarios, but you can’t do something like the code presented in the next snippet:

if (condition) {

registerBean(b1); ❶

} else {

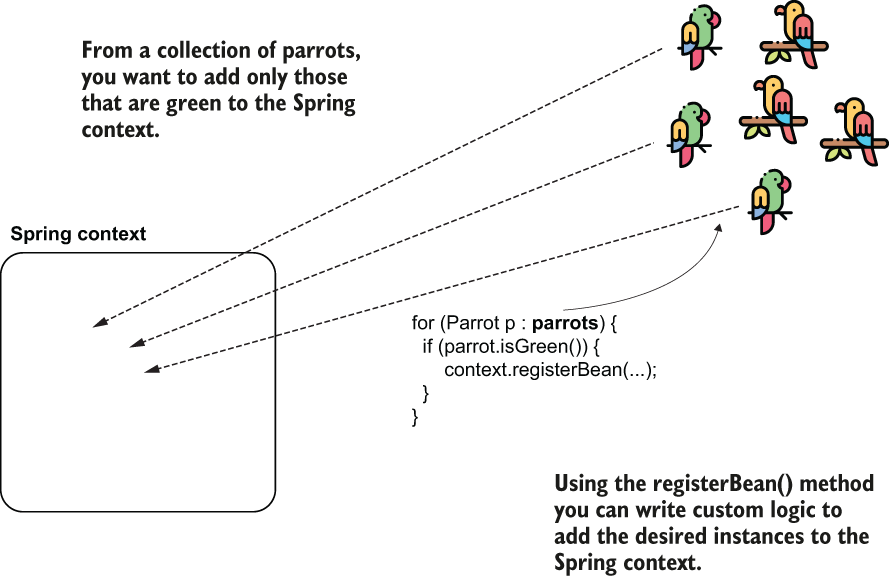
registerBean(b2); ❷

}

❶ If the condition is true, add a specific bean to the Spring context.

❷ Otherwise, add another bean to the Spring context.

The app reads a collection of parrots. Some of them are green; others are orange. You want the app to add to the Spring context only the parrots that are green.



Let’s see how this method works. To add a bean to the Spring context using a programmatic approach, you just need to call the registerBean() method of the ApplicationContext instance. The registerBean() has four parameters, as presented in the next code snippet:

<T> void registerBean(

String beanName,

Class<T> beanClass,

Supplier<T> supplier,

BeanDefinitionCustomizer... customizers);

1. Use the 1st parameter beanName to define a name for the bean you add in the Spring context. If you don’t need to give a name to the bean you’re adding, you can use null as a value when you call the method.
2. The 2nd parameter is the class that defines the bean you add to the context. Say you want to add an instance of the class Parrot; the value you give to this parameter is Parrot.class.
3. The 3rd parameter is an instance of Supplier. The implementation of this Supplier needs to return the value of the instance you add to the context. Remember, Supplier is a functional interface you find in the java.util .function package. The purpose of a supplier implementation is to return a value you define without taking parameters.
4. The 4th and last parameter is a varargs of BeanDefinitionCustomizer. (If this doesn’t sound familiar, that’s okay; the BeanDefinitionCustomizer is just an interface you implement to configure different characteristics of the bean; e.g., making it primary.) Being defined as a varargs type, you can omit this parameter entirely, or you can give it more values of type BeanDefinitionCustomizer.

The project’s configuration class is empty, and the Parrot class we use for our bean definition is just a plain old Java object (POJO); we use no annotation with it:

@Configuration

public class ProjectConfig {

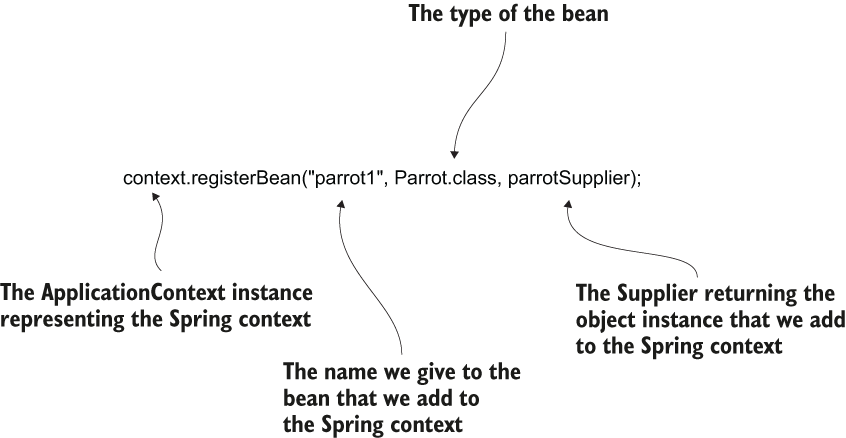
}

public class Parrot {

private String name;

}

In the main method of the project, I’ve used the registerBean() method to add an instance of type Parrot to the Spring context. The syyntax for calling the registerBean() method:



Listing 2.19 Using the registerBean() method to add a bean to the Spring context

public class Main {

public static void main(String[] args) {

var context =

new AnnotationConfigApplicationContext(ProjectConfig.class);

Parrot x = new Parrot(); ❶

x.setName("Kiki");

Supplier<Parrot> parrotSupplier = () -> x; ❷

context.registerBean("parrot1", Parrot.class, parrotSupplier); ❸

Parrot p = context.getBean(Parrot.class); ❹

System.out.println(p.getName()); ❹

}

}

❶ We create the instance we want to add to the Spring context.

❷ We define a Supplier to return this instance.

❸ We call the registerBean() method to add the instance to the Spring context.

❹ To verify the bean is now in the context, we refer to the parrot bean and print its name in the console.

Use one or more bean configurator instances as the last parameters to set different characteristics of the beans you add. For example, you can make the bean primary by changing the registerBean() method call. A primary bean defines the instance Spring selects by default if you have multiple beans of the same type in the context:

context.registerBean("parrot1",

Parrot.class,

parrotSupplier,

bc -> bc.setPrimary(true));

SUMMARIZE: You can add beans to the Spring context in three ways: using the @Bean annotation, using stereotype annotations, and doing it programmatically.

1. Using the @Bean annotation to add instances to the Spring context enables you to add any kind of object instance as a bean and even multiple instances of the same kind to the Spring context. From this point of view, this approach is more flexible than using stereotype annotations. Still, it requires you to write more code because you need to write a separate method in the configuration class for each independent instance added to the context.
2. Using stereotype annotations, you can create beans for only the application classes with a specific annotation (e.g., @Component). This configuration approach requires writing less code, which makes your configuration more comfortable to read. You’ll prefer this approach over the @Bean annotation for classes that you define and can annotate.
3. Using the registerBean() method enables you to implement custom logic for adding beans to the Spring context. Remember, you can use this approach only with Spring 5 and later.

# Chapter 3 Wiring Beans

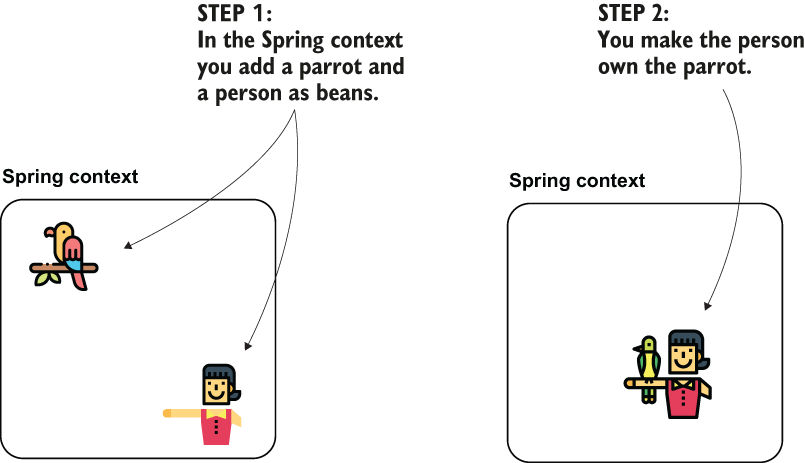
Here we discuss how to access the beans, which we’ve added to the Spring context. In chapter 2, we used the **getBean()** method of the context instance directly to access the beans. But in apps, we need to refer from one bean to another in a straightforward fashion—by telling Spring to provide a reference to an instance from its context where we need it. This way, we establish relationships among the beans (one bean will have a reference to another to delegate calls when it needs).

Here we will implement the relationship between two beans defined in the configuration class annotating methods with the **@Bean** annotation. We use the @Bean annotation to add beans to the Spring context in the cases in which we cannot change the class for which we want to add the bean, for example, if the class is part of the JDK or another dependency.

Say we have two instances in the Spring context: a parrot and a person. We’ll create and add these instances to the context. We want to make the person own the parrot. In other words, we need to link the two instances.

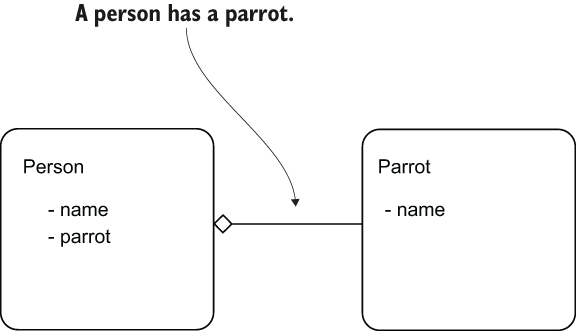
So, for each of the two approaches (wiring and auto-wiring), we have two steps:

1. Add the person and parrot beans to the Spring context (as you learned in chapter 2).
2. Establish a relationship between the person and the parrot.



You can do this using a wiring approach, which implies directly calling the methods that declare the beans to establish the link between them, or through auto-wiring. You use the framework’s dependency injection capabilities.

Technically this is the “has-A” relationship between the person and the parrot object.



## **PROJECT 04 – Wire 2 beans (v.1 – @Bean direct method call)**

In the pom.xml file of the Maven project, we add the dependency for Spring context:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.2.7.RELEASE</version>

</dependency>

We then define a class to describe the Parrot object and one to describe the Person.

public class **Parrot** {

private String name;

// Omitted getters and setters

@Override

public String toString() {

return "Parrot : " + name;

}

}

public class **Person** {

private String name;

private Parrot parrot;

// Omitted getters and setters

}

The following listing shows you how to define the 2 beans using the @Bean annotation in the configuration class.

Listing 3.1 Defining the Person and the Parrot beans

@Configuration

public class ProjectConfig {

@Bean

public Parrot parrot() {

Parrot p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

public Person person() {

Person p = new Person();

p.setName("Ella");

return p;

}

}

You can now write a Main class, and check that the 2 instances aren’t yet linked to one another.

Listing 3.2 The definition of the Main class

public class **Main** {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext (ProjectConfig.class); ❶

Person person = context.getBean(Person.class); ❷

Parrot parrot = context.getBean(Parrot.class); ❸

System.out.println("Person's name: " + person.getName()); ❹

System.out.println("Parrot's name: " + parrot.getName()); ❺

System.out.println("Person's parrot: " + person.getParrot()); ❻

}}

❶ Creates an instance of the Spring context based on the configuration class

❷ Gets a reference to the Person bean from the Spring context

❸ Gets a reference to the Parrot bean from the Spring context

❹ Prints the person’s name to prove that the Person bean is in the context

❺ Prints the parrot’s name to prove that the Parrot bean is in the context

❻ Prints the person’s parrot to prove that there’s not yet a relationship between the instances

When running this app, you’ll see a console output:

Person's name: Ella ❶

Parrot's name: Koko ❷

Person's parrot: null ❸

❶ The Person bean is in the Spring context.

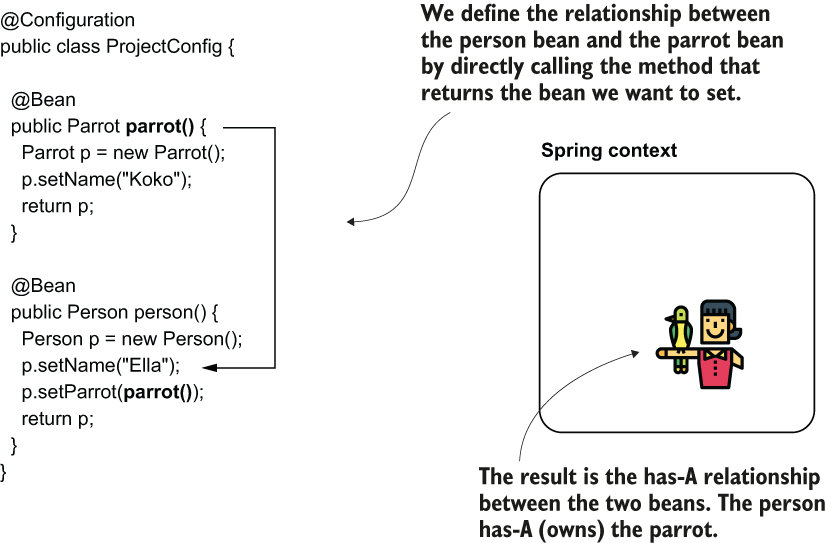
❷ The Parrot bean is in the Spring context.

❸ The relationship between the person and the parrot isn’t established.

The most important thing to observe here is that the person’s parrot (third output line) is null. Both the person and the parrot instances are in the context, however. This output is null, which means there’s not yet a relationship between the instances.

Wiring the beans using a direct method call between the @Bean methods

he first way (wiring) to achieve this is to call one method from another in the configuration class. You’ll find this often used because it’s a straightforward approach.



This approach implies calling the method that returns the bean you want to set directly. You need to call this method from the one that defines the bean for which you set the dependency.

Listing 3.3 Making a link between the beans with a direct method call

@Configuration

public class ProjectConfig {

@Bean

public Parrot parrot() {

Parrot p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

public Person person() {

Person p = new Person();

p.setName("Ella");

p.setParrot(parrot()); ❶

return p;

}

}

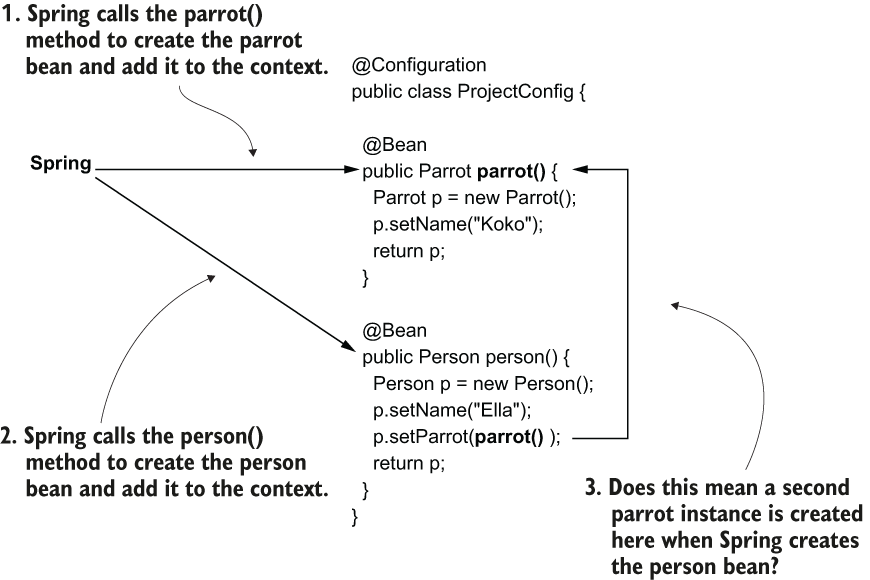
❶ Setting the reference of the parrot bean to the person’s parrot attribute

Running the same app, you’ll observe the output changed in the console.

Person's name: Ella

Person's parrot: Parrot : Koko ❶

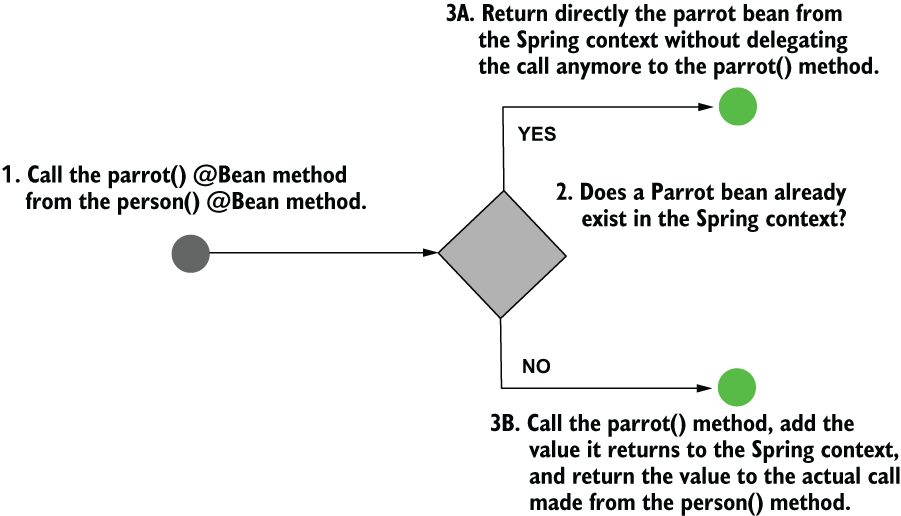
❶ We now observe the relationship between the person and the parrot has been established.



Spring creates a parrot instance when it calls the first @Bean annotated method parrot(). Then, Spring creates a person instance when it calls the second @Bean annotated method person(). The second method, person(), directly calls the first method, parrot(). Does this mean two instances of type parrot are created?

Spring is smart enough to understand that by calling the parrot() method, you want to refer to the parrot bean in its context. When we use the @Bean annotation to define beans into the Spring context, Spring controls how the methods are called and can apply logic above the method call.

If the parrot bean already exists in the context, then instead of calling the parrot() method, Spring will directly take the instance from its context. If the parrot bean does not yet exist in the context, Spring calls the parrot() method and returns the bean (figure 3.6).



It is easy to test this behavior. Just add a no-args constructor to the Parrot class and print a message into the console from it. How many times will the message be printed in the console? If the behavior is correct, you’ll see the message only once.

public class Parrot {

private String name;

public Parrot() {

System.out.println("Parrot created");

}

// Omitted getters and setters

@Override

public String toString() {

return "Parrot : " + name;

}

}

Rerun the app. The output changed, and now the “Parrot created” message appears as well. You’ll observe it appears only once, which proves that Spring manages the bean creation and calls the parrot() method only once:

Parrot created

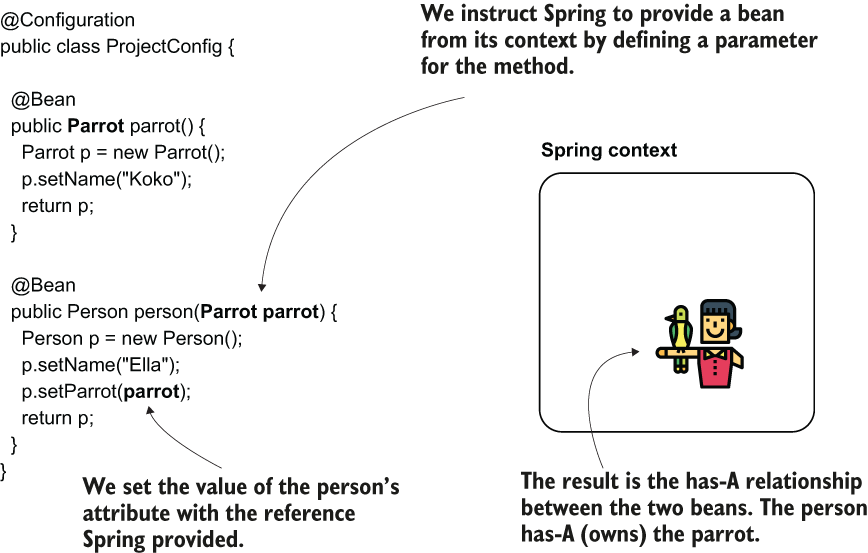
Person's name: Ella

Person's parrot: Parrot : Koko

## **PROJECT 05 – Wire 2 beans (v.2 – @Bean, @Component - using method parameters)**

Wiring the beans using the @Bean annotated method’s parameters

We add a parameter to the method of the corresponding type of object, and we rely on Spring to provide us a value through that parameter. With this approach, it doesn’t matter if the bean we want to refer to is defined with a method annotated with @Bean or using a stereotype annotation like @Component.



Listing 3.4 Injecting bean dependencies by using parameters of the methods

@Configuration

public class ProjectConfig {

@Bean

public Parrot parrot() {

Parrot p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

public Person person(Parrot parrot) { ❶

Person p = new Person();

p.setName("Ella");

p.setParrot(parrot);

return p;

}

}

❶ Spring injects the parrot bean into this parameter.

I refer here to what is called dependency injection (DI). DI is a technique involving the framework setting a value into a specific field or parameter. In our case, Spring sets a particular value into the parameter of the person() method when calling it and resolves a dependency of this method. DI is an application of the IoC principle, and IoC implies that the framework controls the application at execution and not the oposit.

When running the app, you observe that the parrot Koko is indeed linked to the person Ella:

Parrot created

Person's name: Ella

Person's parrot: Parrot : Koko

**Using the @Autowired annotation to inject beans**

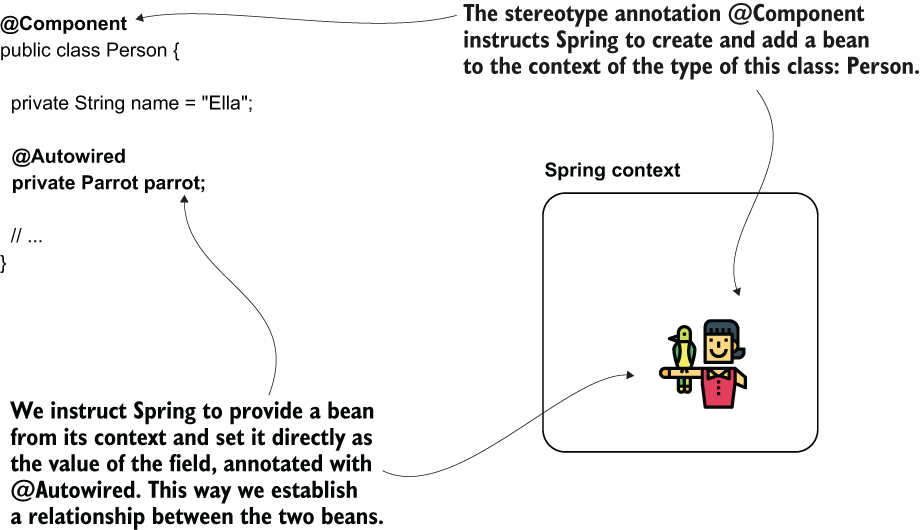
Using the @Autowired annotation, we mark an object’s property where we want Spring to inject a value from the context, and we mark this intention directly in the class that defines the object that needs the dependency. This approach makes it easier to see the relationship between the two objects than the alternatives. There are 3 ways we can use the @Autowired annotation:

1. Injecting the value in the field of the class, which you usually find in examples and proofs of concept
2. Injecting the value through the constructor parameters of the class approach that you’ll use most often in real-world scenarios
3. Injecting the value through the setter, which you’ll rarely use in production-ready code

**PROJECT 06 – (v.3.1) @Autowired over field and @Component**

(1) Using **@Autowired** to inject the values through the **class fields**

Using the @Autowired annotation over the field, we instruct Spring to provide a value for that field from its context. Spring creates the two beans, person and parrot, and injects the parrot object to the field of the bean of type Person.



We annotate the parrot field of the Person class with the @Autowired annotation to tell Spring we want to inject a value there from its context. Let’s start with the classes defining our 2 objects: Person and Parrot:

@Component

public class Parrot {

private String name = "Koko";

// Omitted getters and setters

@Override

public String toString() {

return "Parrot : " + name;

}

}

We use the stereotype annotation @Component here as an alternative to creating the bean using the configuration class. When annotating a class with @Component, Spring knows it has to create an instance of that class and add it to its context.

@Component

public class Person {

private String name = "Ella";

@Autowired ❶

private Parrot parrot;

// Omitted getters and setters

}

❶ Annotating the field with @Autowired, we instruct Spring to inject an appropriate value from its context.

I could have defined the beans using @Bean, but most often, in real-world scenarios, you’ll encounter @Autowired used together with stereotype annotations.

Over the configuration class ProjectConfig, I’ll use the @ComponentScan annotation to tell Spring where to find the classes I’ve annotated with @Component.

@Configuration

@ComponentScan(basePackages = "beans")

public class ProjectConfig {

}

The main class is the same ,to prove that Spring injected the parrot bean’s reference correctly:

public class Main {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext (ProjectConfig.class);

Person p = context.getBean(Person.class);

System.out.println("Person's name: " + p.getName());

System.out.println("Person's parrot: " + p.getParrot());

}

}

This will print in the app’s console something similar to the output presented next. The second line of the output proves that the parrot (in my case, named Koko) belongs to the person bean (named Ella):

Person's name: Ella

Person's parrot: Parrot : Koko

By injecting the value directly in the field you don’t have the option to make the field final, and this way, make sure no one can change its value after initialization:

@Component

public class Person {

private String name = "Ella";

@Autowired

private final Parrot parrot; ❶

}

❶ This doesn’t compile. You cannot define a final field without an initial value.

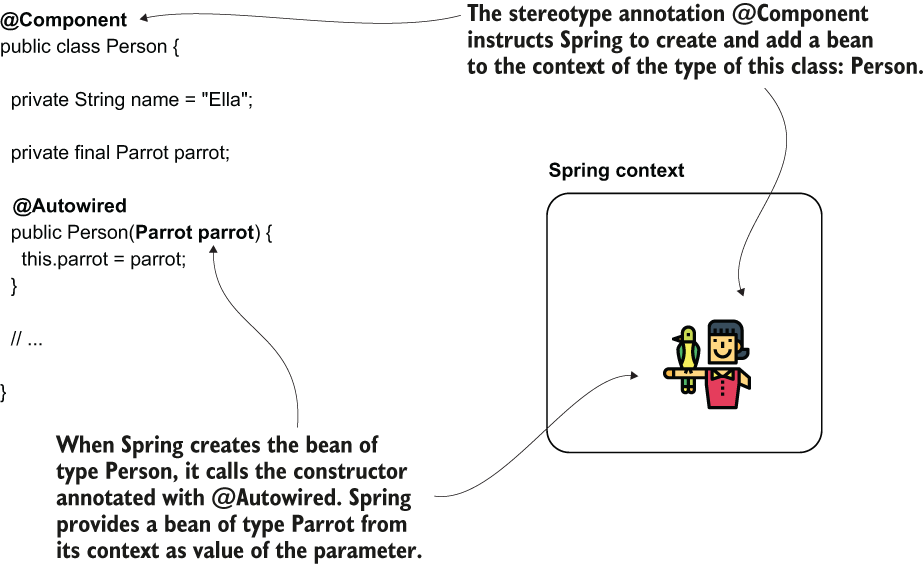
It’s more difficult to manage the value yourself at initialization.

You sometimes need to create instances of the objects and easily manage the unit tests’ dependencies.

**PROJECT 07 – (v.3.2) @Autowired over constructor and @Component**

(2) Using **@Autowired** to inject the values through the **constructor**

The 2nd option you have for injecting values into the object’s attributes when Spring creates a bean is using the class’s constructor defining the instance. This approach is the one used most often in production code and the one I recommend. It enables you to define the fields as final, ensuring no one can change their value after Spring initializes them. The possibility to set the values when calling the constructor also helps you when writing specific unit tests where you don’t want to rely on Spring making the field injection for you.



When you define a parameter of the constructor, Spring provides a bean from its context as a value to that parameter when calling the constructor.

Listing 3.5 Injecting the values through constructor

@Component

public class Person {

private String name = "Ella";

private final Parrot parrot; ❶

@Autowired ❷

public Person(Parrot parrot) {

this.parrot = parrot;

}

// Omitted getters and setters

}

❶ We can now make the field final to ensure its value cannot be changed after initialization.

❷ We use the @Autowired annotation over the constructor.

The app displays the same result as in the previous example. You can see that the person owns the parrot, so Spring established the link between the 2 instances correctly:

Person's name: Ella

Person's parrot: Parrot : Koko

! Starting with Spring version 4.3, when you only have 1 constructor in the class, you can omit writing the @Autowired annotation.

**PROJECT 08 – (v.3.3) @Autowired over setter and @Component**

(3) Using **@Autowired** to inject the values through the **setter**

You won’t often find developers applying the approach of using the setter for dependency injection. This approach has more disadvantages than advantages: it’s more challenging to read, it doesn’t allow you to make the field final, and it doesn’t help you in making the testing easier.

I only needed to change the Person class to implement this:

@Component

public class Person {

private String name = "Ella";

private Parrot parrot;

// Omitted getters and setters

@Autowired

public void setParrot(Parrot parrot) {

this.parrot = parrot;

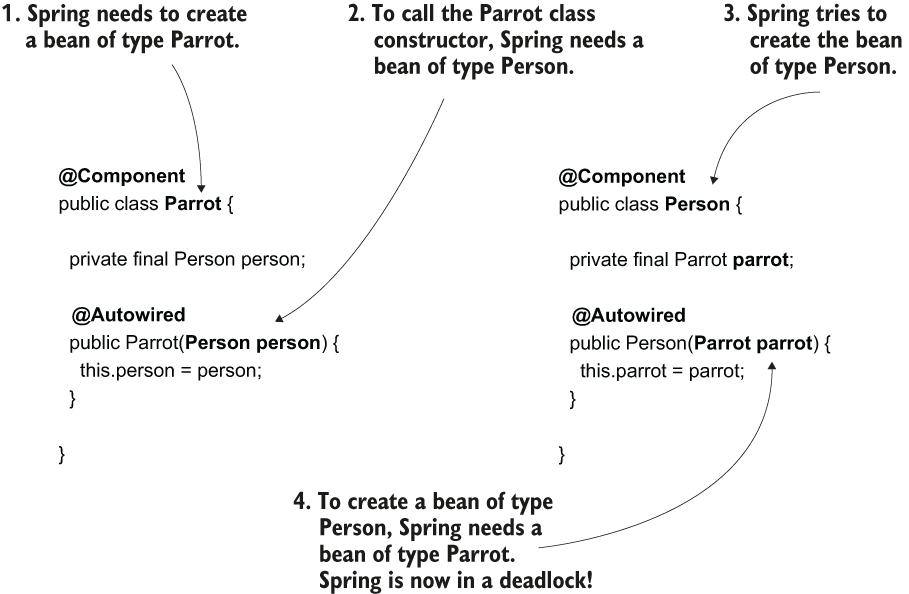
}

}

When running the app, you’ll get the same output as the previously discussed examples of this section.

**3.3 Dealing with circular dependencies**

A circular dependency is a situation in which, to create a bean (let’s name it Bean A), Spring needs to inject another bean that doesn’t exist yet (Bean B). But Bean B also requests a dependency to Bean A. So, to create Bean B, Spring needs first to have Bean A. Spring is now in a deadlock. It cannot create Bean A because it needs Bean B, and it cannot create Bean B because it needs Bean A.



A circular dependency is easy to avoid. You just need to make sure you don’t define objects whose creation depends on the other. Having dependencies from one object to another like this is a bad design of classes. In such a case, you need to rewrite your code.

**Choosing from multiple beans in the Spring context**

In this section, we discuss the scenario in which Spring needs to inject a value into a parameter or class field but has multiple beans of the same type to choose from. Say you have 3 Parrot beans in the Spring context. You configure Spring to inject a value of type Parrot into a parameter. How will Spring behave? Which of the beans of the same type would the framework choose to inject in such a scenario?

Depending on your implementation, you have the following cases:

1. The identifier of the parameter matches the name of one of the beans from the context (which is the same as the name of the method annotated with @Bean that returns its value). In this case, Spring will choose the bean for which the name is the same as the parameter.
2. The identifier of the parameter doesn’t match any of the bean names from the context. Then you have the following options:
   1. You marked one of the beans as primary. In this case, Spring will select the primary bean for injection.
   2. You can explicitly select a specific bean using the @Qualifier annotation.
   3. If none of the beans is primary and you don’t use @Qualifier, the app will fail with an exception, complaining that the context contains more beans of the same type and Spring doesn’t know which one to choose.

**PROJECT 09 – Multiple beans with @Bean – by name**

Listing 3.6 Using parameter injection for more than one bean

@Configuration

public class ProjectConfig {

@Bean

public Parrot parrot1() {

Parrot p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

public Parrot parrot2() {

Parrot p = new Parrot();

p.setName("Miki");

return p;

}

@Bean

public Person person(Parrot parrot2) { ❶

Person p = new Person();

p.setName("Ella");

p.setParrot(parrot2);

return p;

}}

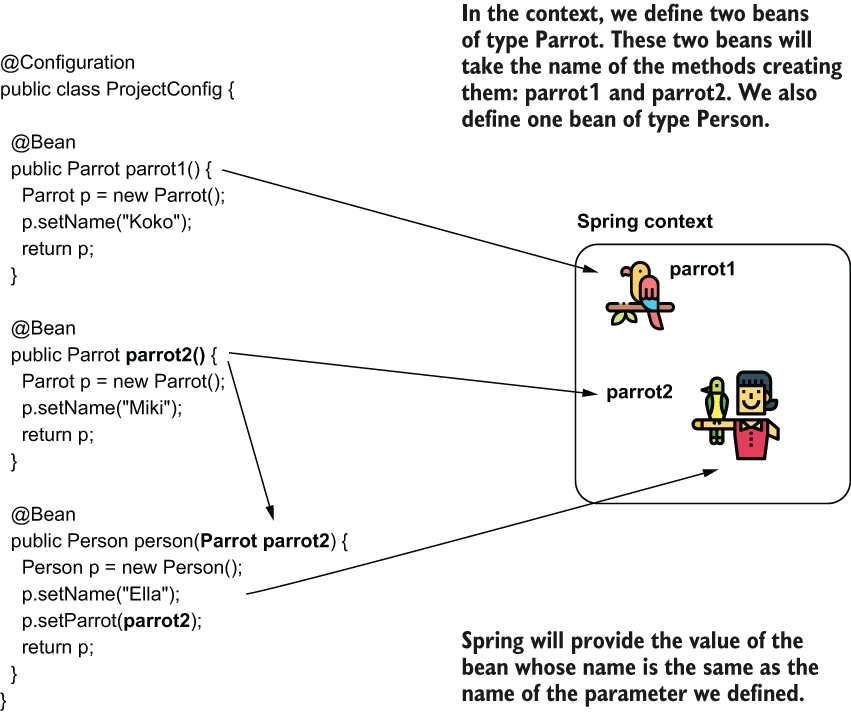
❶ The name of the parameter matches the name of the bean representing parrot Miki.

Running the app with this configuration, you’d observe that Spring linked the person bean to the parrot named Miki because the bean representing this parrot has the name parrot2:

Parrot created

Person's name: Ella

Person's parrot: Parrot : Miki



One way to instruct Spring to provide you a specific instance from its context, when the context contains more than one instance of the same type, is to rely on the name of this instance. Just name the parameter the same as the instance you’d like Spring to provide you.

**PROJECT 10 – Multiple beans with @Bean – @Qualifier**

In a real-world scenario, I prefer to avoid relying on the name of the parameter, which could be easily refactored and changed by mistake by another developer. To feel more comfortable, I usually choose a more visible approach to express my intention to inject a specific bean: using the @Qualifier annotation.

The following listing provides an example using the @Qualifier annotation. Observe that instead of having a specific identifier of the parameter, I now specify the bean I want to inject using the value attribute of the @Qualifier annotation.

Listing 3.7 Using the @Qualifier annotation

@Configuration

public class ProjectConfig {

@Bean

public Parrot parrot1() {

Parrot p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

public Parrot parrot2() {

Parrot p = new Parrot();

p.setName("Miki");

return p;

}

@Bean

public Person person(

@Qualifier("parrot2") Parrot parrot) { ❶

Person p = new Person();

p.setName("Ella");

p.setParrot(parrot);

return p;

}

}

❶ Using the @Qualifier annotation, you clearly mark your intention to inject a specific bean from the context.

Rerunning the application, the app prints the same result into the console:

Parrot created

Person's name: Ella

Person's parrot: Parrot : Miki

**PROJECT 11 – Multiple beans with @Autowired and @Bean – by name**

We define 2 beans of type Parrot (using the @Bean annotation) and an instance of Person (using stereotype annotations). I’ll configure Spring to inject one of the 2 parrot beans in the bean of type Person. I didn’t add the @Component annotation to the Parrot class because I intend to define the two beans of type Parrot using the @Bean annotation in the configuration class:

public class Parrot {

private String name;

// Omitted getters, setters, and toString()

}

We define a bean of type Person using the @Component stereotype annotation. Observe the identifier I gave to the parameter of the constructor in the next code snippet. The reason I gave the identifier “parrot2” is this is the name I’ll also configure for the bean in the context I want Spring to inject into that parameter:

@Component

public class Person {

private String name = "Ella";

private final Parrot parrot;

public Person(Parrot parrot2) {

this.parrot = parrot2;

}

// Omitted getters and setters

}

I define two beans of type Parrot using the @Bean annotation in the configuration class. . I still have to add @ComponentScan to tell Spring where to find the classes annotated with stereotype annotations (Person).

Listing 3.8 Defining the beans of type Parrot in the configuration class

@Configuration

@ComponentScan(basePackages = "beans")

public class ProjectConfig {

@Bean

public Parrot parrot1() {

Parrot p = new Parrot();

p.setName("Koko");

return p;

}

@Bean

public Parrot parrot2() { ❶

Parrot p = new Parrot();

p.setName("Miki");

return p;

}

}

❶ With the current setup, the bean named parrot2 is the one that Spring injects into the Person bean.

Because the name of the constructor’s parameter matches one of the bean’s names in the Spring context (parrot2), Spring injects that bean, so the name of the parrot the app prints in the console is Miki:

public class Main {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext(ProjectConfig.class);

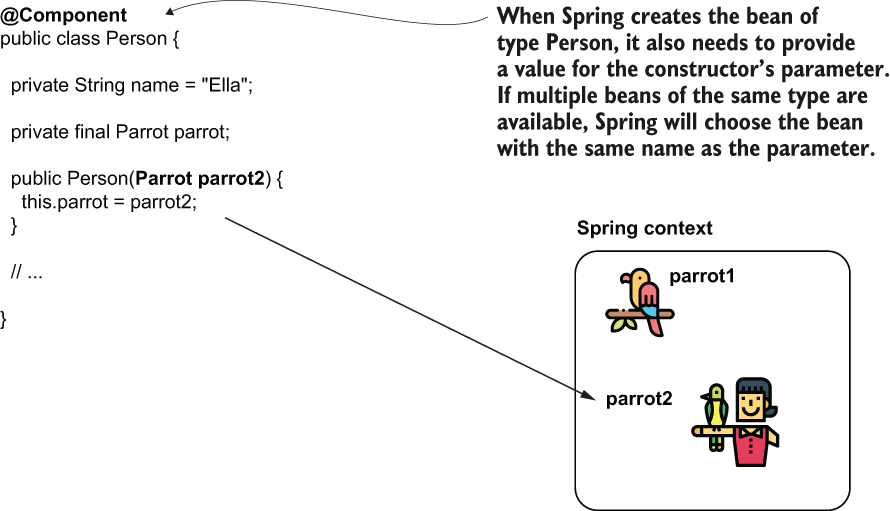
Person p = context.getBean(Person.class);

System.out.println("Person's name: " + p.getName());

System.out.println("Person's parrot: " + p.getParrot());

}

}



When the Spring context contains multiple beans of the same type, Spring will select the bean whose name matches the name of the parameter.

Running this app, the console shows the following output:

Person's name: Ella

Person's parrot: Parrot : Miki

**PROJECT 12 – Multiple beans with @Autowired and @Bean – @Qualifier**

I prefer using the @Qualifier annotation to express my intention clearly: I inject a specific bean from the context. This way, we minimize the chance that someone would refactor the name of the variable and thus affect how the app works.

In the Person class, using the @Qualifier annotation, I specify the name of the bean I want Spring to inject from the context, and I don’t rely on the identifier of the constructor’s parameter:

@Component

public class Person {

private String name = "Ella";

private final Parrot parrot;

public Person(@Qualifier("parrot2") Parrot parrot) {

this.parrot = parrot;

}

// Omitted getters and setters

}

The behavior of the app doesn’t change, and the output remains the same. This approach makes your code less subject to mistakes.

**Summary**

The Spring context is a place in the app’s memory that the framework uses to keep the objects it manages. You need to add any object that needs to be augmented to the Spring context with a feature the framework offers.

When implementing an app, you need to refer from one object to another. This way, an object can delegate actions to other objects when executing their responsibilities. To implement this behavior, you need to establish relationships among the beans in the Spring context.

You can establish a relationship between two beans using one of three approaches:

* + Directly referring to the @Bean annotated method that creates one of them from the method that creates the other. Spring knows you refer to the bean in the context, and if the bean already exists, it doesn’t call the same method again to create another instance. Instead, it returns the reference to the existing bean in the context.
  + Defining a parameter to the method annotated with @Bean. When Spring observes the @Bean method has a parameter, it searches a bean of that parameter’s type in its context and provides that bean as a value to the parameter.
  + Using the @Autowired annotation in three ways:
    1. Annotate the field in the class where you want to instruct Spring to inject the bean from the context. You’ll find this approach often used in examples and proofs of concept.
    2. Annotate the constructor you’d like Spring to call to create the bean. Spring will inject other beans from the context in the constructor’s parameters. You’ll find this approach the most used in real-world code.
    3. Annotate the setter of the attribute where you’d like Spring to inject the bean from the context. You won’t find this approach often used in production-ready code.

Whenever you allow Spring to provide a value or reference through an attribute of the class or a method or constructor parameter, we say Spring uses DI, a technique supported by the IoC principle.

The creation of two beans that depend on one another generates a circular dependency. Spring cannot create the beans with a circular dependency, and the execution fails with an exception. When configuring your beans, make sure you avoid circular dependencies.

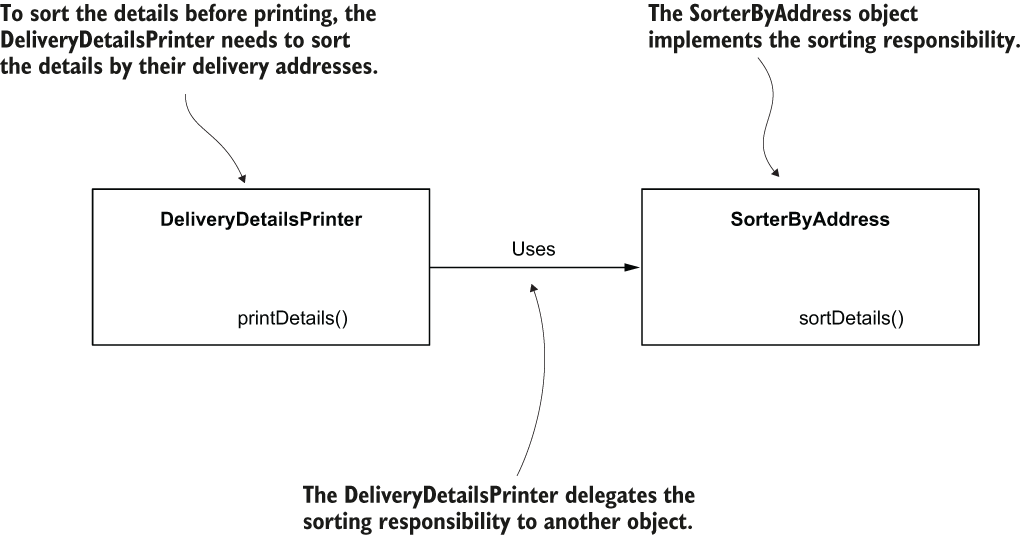
When Spring has more than one bean of the same type in its context, it can’t decide which of those beans need to be injected. You can tell Spring which is the instance it needs to inject by

* + using the @Primary annotation, which marks one of the beans as the default for dependency injection, or
  + naming the beans and injecting them by name using the @Qualifier annotation.

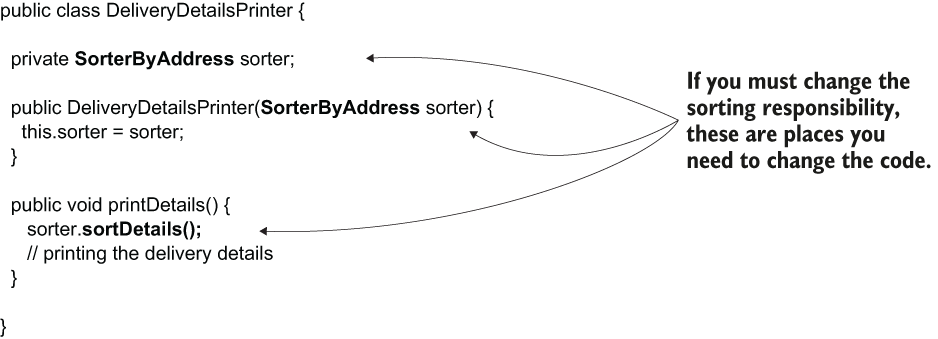
# Chapter 4 The Spring context: Using abstractions

1. Using interfaces for decoupling implementations

The DeliveryDetailsPrinter object delegates the responsibility of sorting the delivery details by the delivery addresses to another object named SorterByAddress.



The DeliveryDetailsPrinter directly delegates the sorting responsibility to the SorterByAddress object. If we keep this class design, we may face difficulties later if we need to change this functionality. Let’s imagine you need to change the printed details order later, and the new order is by the sender’s name. You’d need to replace the SorterByAddress object with another one implementing the new responsibility, but you’d also need to change the DeliveryDetailsPrinter object that uses the sorting responsibility.

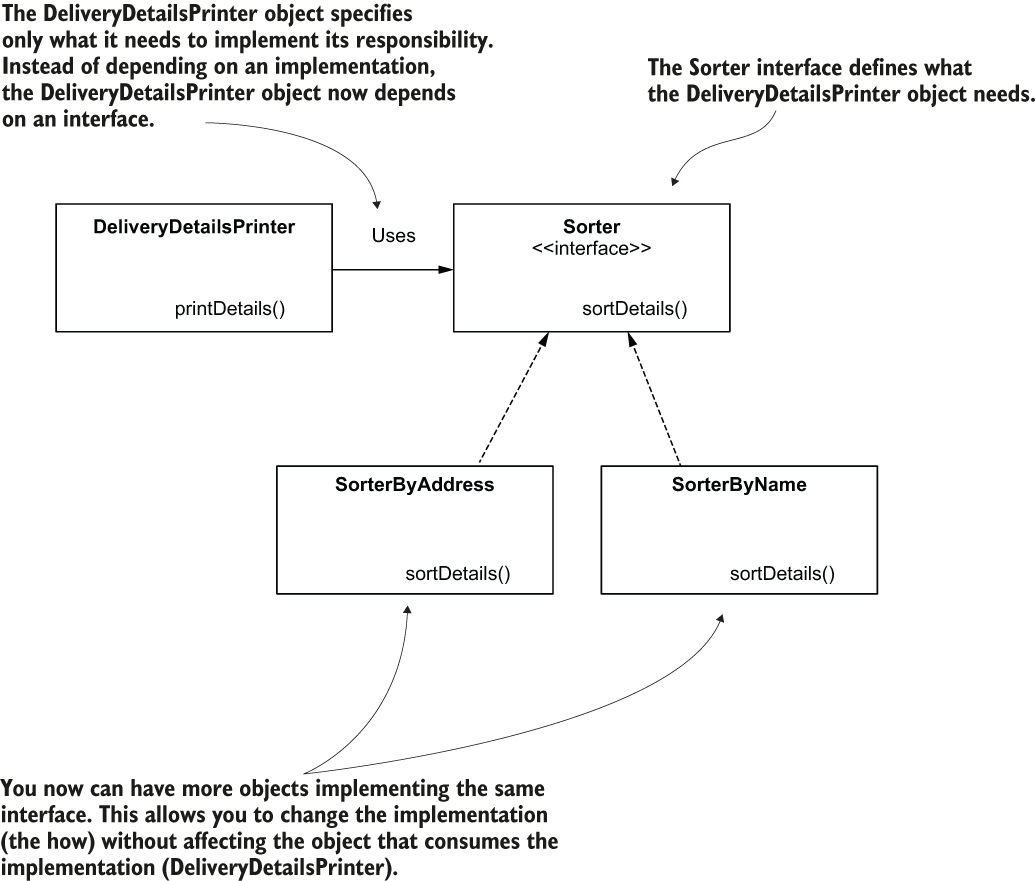


Because the two objects are strongly coupled, if you want to change the sorting responsibility, you also need to change the object using this responsibility. A better design would allow you to change the sorting responsibility without changing the object that uses the responsibility.

When changing an object’s responsibility, we want to avoid the need to change other objects using the changed responsibility. This design’s problem occurs because the DeliveryDetailsPrinter object specifies both what it needs and how it needs. An object only needs to specify what it needs and stay completely unaware of how is implemented. We do this, by using interfaces.

I introduced an interface named Sorter to decouple the two objects. Instead of declaring a SorterByAddress, the DeliveryDetailsPrinter object only specifies it needs a Sorter. You can now have as many objects as you’d like to solve the what requested by the DeliveryDetailsPrinter. Any object implementing the Sorter interface can satisfy the dependency of the DeliveryDetailsPrinter object at any time.

Instead of depending directly on an implementation, the DeliveryDetailsPrinter object depends on an interface (a contract). DeliveryDetailsPrinter can use any object implementing this interface instead of being stuck to a specific implementation.



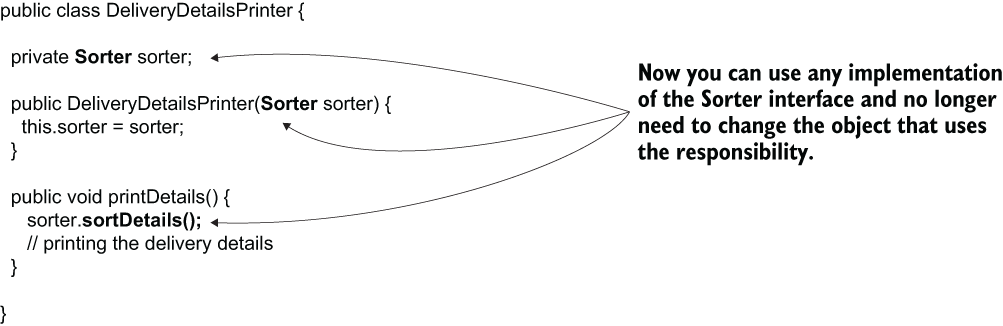
Sorter interface definition:

public interface Sorter {

void sortDetails();

}

DeliveryDetailsPrinter object depends on the interface instead of the implementation directly, you don’t need to change it further if you change the way the delivery details are sorted.



**PROJECT 13 – CommentApp without framework – Implementing interfaces**

Implementing an app a team uses to manage their tasks. One of the app’s features is allowing the users to leave comments for the tasks. When a user publishes a comment, it is stored somewhere (e.g., in a database), and the app sends an email to a specific address configured in the app.

In standard real-world applications, we usually refer to the objects implementing uses cases as **services**, and that’s what we’ll do here. We’ll need a service that implements the “publish comment” use case. Let’s name this object *CommentService*. I prefer to give the service classes a name that ends with “service” so that their role in the project stands out.

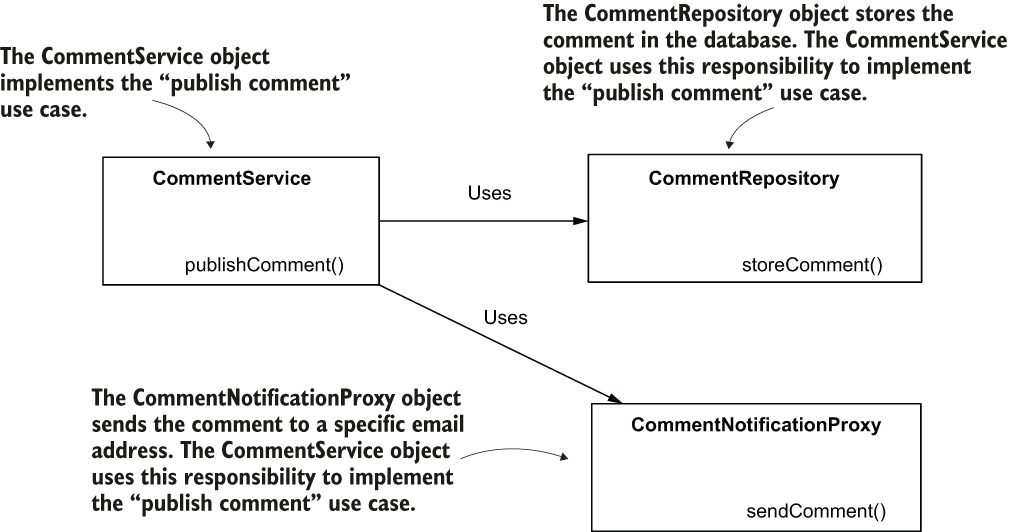
The use case consists of two actions: storing the comment and sending the comment by mail. As they are quite different from one another, we consider these actions to be 2 different responsibilities, and thus we need to implement 2 different objects.

When we have an object working directly with a database, we generally name such an object **repository**. Sometimes you also find such objects referred to as data access objects (**DAO**). Let’s name the object that implements the storing comment responsibility *CommentRepository*.

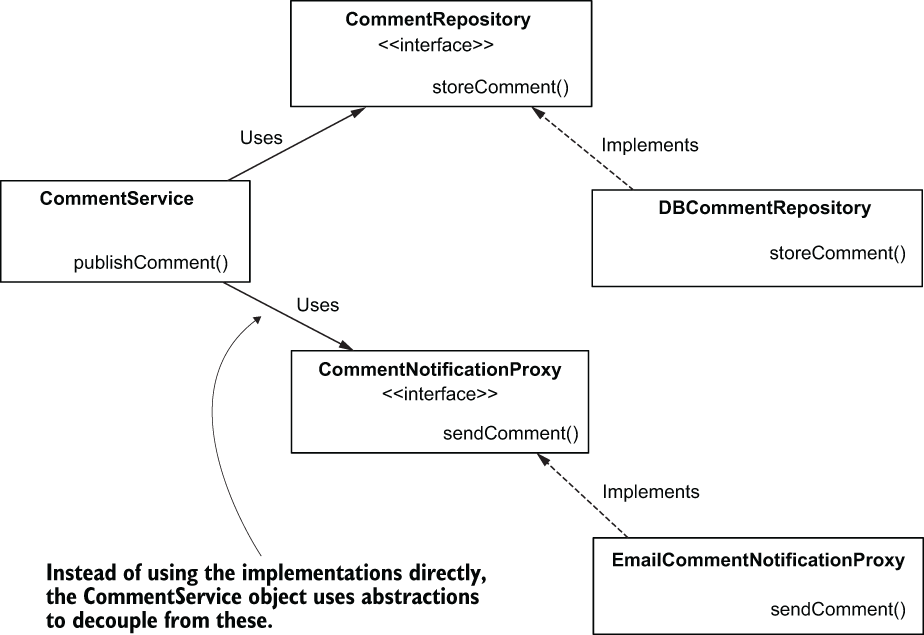
When implementing objects whose responsibility is to establish communication with something outside the app, we name these objects **proxies**, so let’s name the object whose responsibility is sending the email *Comment-NotificationProxy*.

The CommentService object implements the “publish comment” use case. To do this, it needs to delegate to the responsibilities implemented by the CommentRepository and the CommentNotificationProxy objects.

We need to make sure we decouple the implementations by using interfaces. In the end, the CommentRepository might now use a database to store the comments. But in the future, maybe this needs to be changed to use some other technology or an external service. We can say the same for the CommentNotificationProxy object. Now it sends the notification by email, but maybe in a future version the comment notification needs to be sent through some other channel.

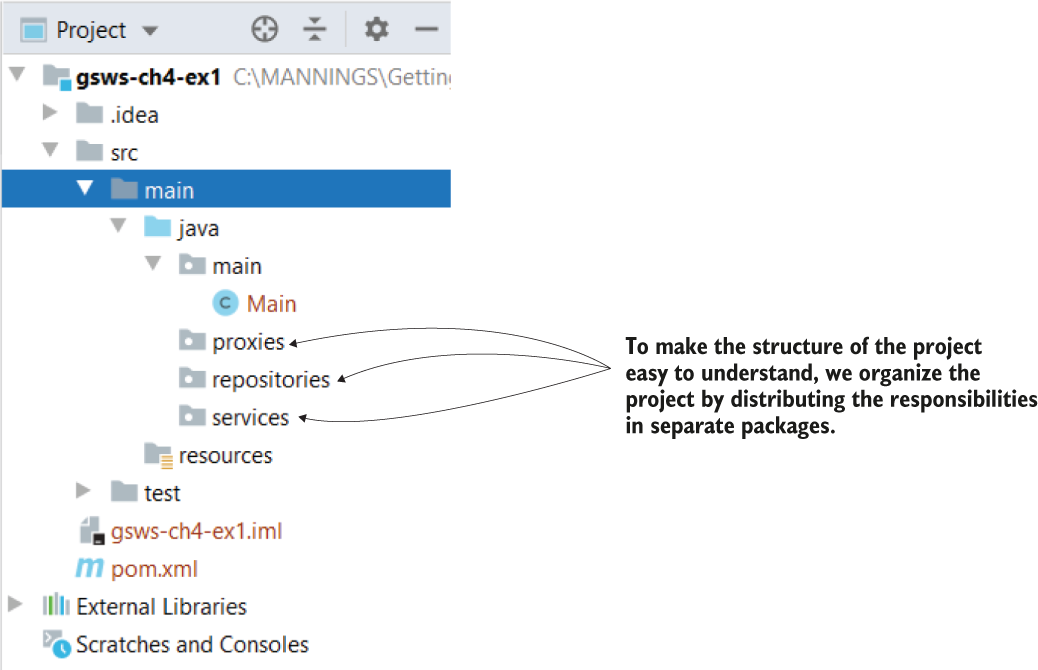


Here how to decouple this class design by using abstractions. Instead of designing CommentRepository and CommentNotificationProxy as classes, we design them as interfaces that we can implement to define the functionality:



The CommentService object depends on the abstractions provided by CommentRepository and CommentNotificationProxy interfaces. The classes DBCommentRepository and EmailCommentNotificationProxy further implement these interfaces. This design decouples the implementation of the “publish comment” use case from its dependencies and makes the application easier to change for future developments.

We create a plain Maven project, without adding any external dependencies to the pom.xml file.



The project structure. We declare a separate package for each responsibility to make the structure of the project easy to read and understand.

We’ll also have to represent the comment somehow. We just need to write a small POJO class for defining the comment. We start the implementation of the use case with writing this POJO class. The responsibility of this type of object is simply to model the data the app uses, and we call it model. I’ll consider a comment that has two attributes: a text and an author. Let’s create a package model in which we define a class Comment.

*POJO* is a simple object without dependencies, only described by its attributes and methods.

Listing 4.1 Defining the comment

public class Comment {

private String author;

private String text;

// Omitted getters and setters

}

We can now define the responsibilities of repository and proxy.

Listing 4.2 Defining the CommentRepository interface

public interface CommentRepository {

void storeComment(Comment comment);

}

We store this interface and the class implementing it in the **repositories** package of the project.

The interface only gives the what the CommentService object needs for implementing the use case: store a comment. When you define an object that implements this contract, it needs to override the storeComment(Comment comment) method to define the how.

Listing 4.3 Implementing the CommentRepository interface

public class DBCommentRepository implements CommentRepository {

@Override

public void storeComment(Comment comment) {

System.out.println("Storing comment: " + comment.getText());

}

}

Similarly, we define an interface for the second responsibility the CommentService object needs: CommentNotificationProxy. We define this interface and the class implementing it in the **proxies** package of the project.

Listing 4.4 Define a CommentNotificationProxy interface

public interface CommentNotificationProxy {

void sendComment(Comment comment);

}

Listing 4.5 Implementation of the CommentNotificationProxy interface

public class EmailCommentNotificationProxy implements CommentNotificationProxy {

@Override

public void sendComment(Comment comment) {

System.out.println("Sending notification for comment: " + comment.getText());

}

}

We can now implement the object itself with the two dependencies of the Comment- Service object (the CommentRepository and the CommentNotificationProxy). In the **service** package, we write the CommentService class as presented in the following listing.

Listing 4.6 Implementing the CommentService object

public class CommentService {

private final CommentRepository commentRepository; ❶

private final CommentNotificationProxy commentNotificationProxy; ❶

public CommentService( ❷

CommentRepository commentRepository,

CommentNotificationProxy commentNotificationProxy) {

this.commentRepository = commentRepository;

this.commentNotificationProxy = commentNotificationProxy;

}

public void publishComment(Comment comment) { ❸

commentRepository.storeComment(comment);

commentNotificationProxy.sendComment(comment);

}

}

❶ We define the 2 dependencies as attributes of the class.

❷ We provide the dependencies when the object is built through the parameters of the constructor.

❸ We implement the use case that delegates the “store comment” and “send notification” responsibilities to the dependencies.

Let’s now write a Main class, as presented in the next listing, and test the whole class design.

Listing 4.7 Calling the use case in the Main class

public class Main {

public static void main(String[] args) {

var commentRepository = new DBCommentRepository(); ❶

var commentNotificationProxy = new EmailCommentNotificationProxy(); ❶

var commentService = new CommentService(commentRepository, commentNotificationProxy); ❷

var comment = new Comment(); ❸

comment.setAuthor("Laurentiu"); ❸

comment.setText("Demo comment"); ❹

commentService.publishComment(comment); ❹

}

}

❶ Creates the instance for the dependencies

❷ Creates the instance of the service class and providing the dependencies

❸ Creates an instance of comment to send as a parameter to the publish comment use case

❹ Calls the publish comment use case

When running this application, you’ll observe the two lines in the console printed by the CommentRepository and the CommentNotificationProxy objects.:

Storing comment: Demo comment

Sending notification for comment: Demo comment

**PROJECT 14 – CommentApp with Spring – using constructor DI (dependency injection)**

Here we apply the Spring framework over the class design we implemented.

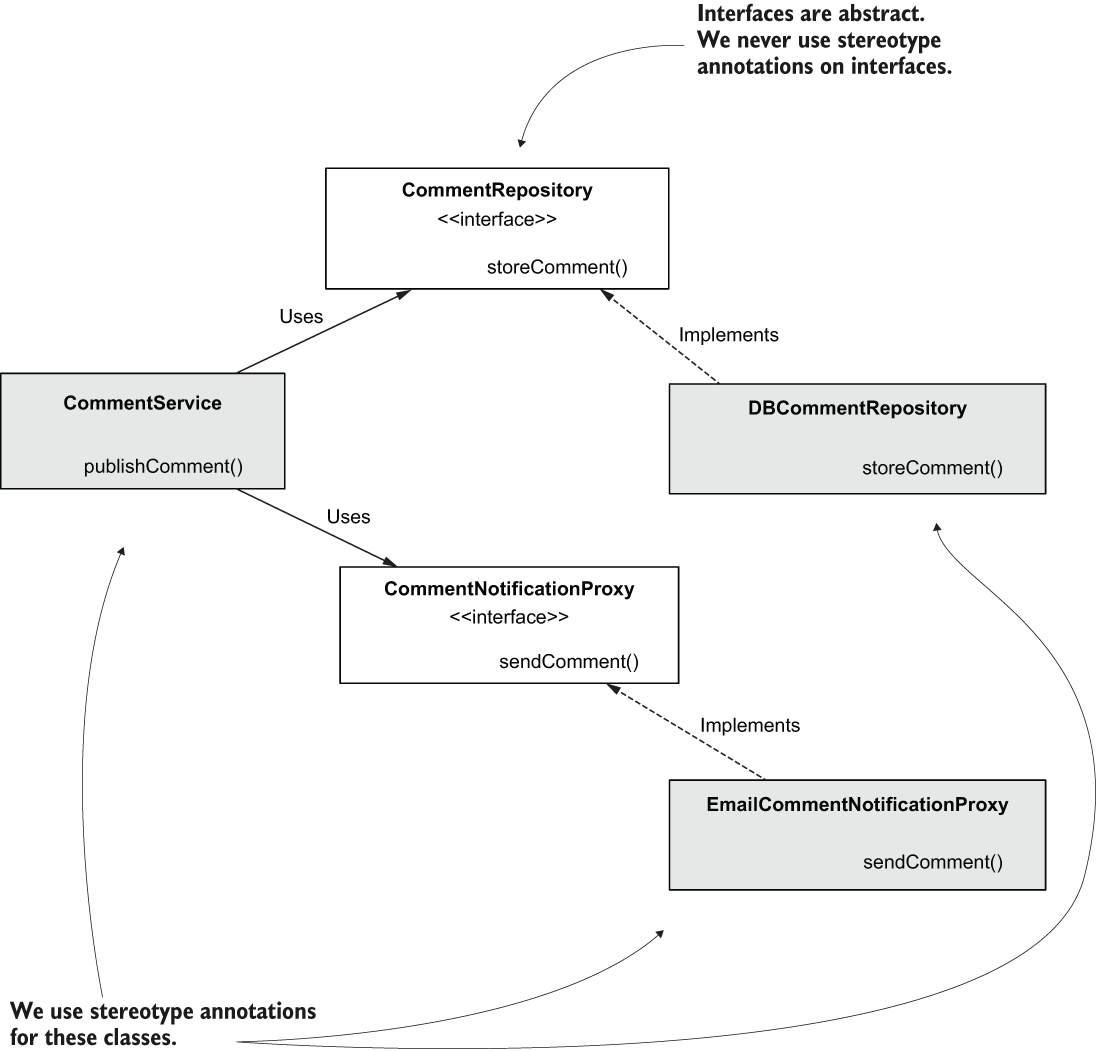
We’ll start by adding the Spring dependency to our project.

Deciding which objects should be part of the Spring context

The main reason to add an object to the Spring context is to allow Spring to control it and further augment it with functionalities the framework provides. So the decision should be based on the question, “Does this object need to be managed by the framework?” We need to add the object to the Spring context if it either has a dependency we need to inject from the context or if it’s a dependency itself. Looking at our implementation, the only object that doesn’t have a dependency and is also not a dependency itself is Comment.

Adding objects to the Spring context without needing the framework to manage them adds unnecessary complexity to your app, making the app both more challenging to maintain and less performant. If you add the object to be managed by Spring without getting any benefit from the framework, you just over-engineer your implementation.

Observe that the 2 interfaces in remain white (we don’t mark them with @Component). We use stereotype annotations for the classes that Spring needs to create instances and add these instances to its context. It doesn’t make sense to add stereotype annotations on interfaces or abstract classes because these cannot be instantiated. The classes we’ll mark with the @Component stereotype annotation are shaded gray. When the context is loaded, Spring creates instances of these classes and adds them to its context.



Listing 4.8 Adding @Component to the DBCommentRepository class

@Component ❶

public class DBCommentRepository implements CommentRepository {

@Override

public void storeComment(Comment comment) {

System.out.println("Storing comment: " + comment.getText());

}

}

❶ Marking the class with @Component instructs Spring to instantiate the class and add an instance as a bean in its context.

Listing 4.9 Adding @Component to the EmailCommentNotificationProxy class

@Component

public class EmailCommentNotificationProxy implements CommentNotificationProxy {

@Override

public void sendComment(Comment comment) {

System.out.println( "Sending notification for comment: " + comment.getText());

}

}

We change the CommentService class as well by annotating it with @Component. Spring sees the attributes are defined with interface types and is smart enough to search in its context for beans created with classes that implement these interfaces. As we have only one constructor in the class, the @Autowired annotation is optional.

Listing 4.10 Making the CommentService class a component

@Component ❶

public class CommentService {

private final CommentRepository commentRepository;

private final CommentNotificationProxy commentNotificationProxy;

❷

public CommentService( ❸

CommentRepository commentRepository,

CommentNotificationProxy commentNotificationProxy) {

this.commentRepository = commentRepository;

this.commentNotificationProxy = commentNotificationProxy;

}

public void publishComment(Comment comment) {

commentRepository.storeComment(comment);

commentNotificationProxy.sendComment(comment);

}

}

❶ Spring creates a bean of this class and adds it to its context.

❷ We would have to use @Autowired if the class had more than one constructor.

❸ Spring uses this constructor to create the bean and injects references from its context in the parameters when creating the instance.

We only need to tell Spring where to find the classes annotated with stereotype annotations and test the app.

Listing 4.11 Using @ComponentScan in the configuration class

@Configuration ❶

@ComponentScan( ❷

basePackages = {"proxies", "services", "repositories"}

)

public class ProjectConfiguration {

}

❶ The @Configuration annotation marks the configuration class.

❷ We use the @ComponentScan annotation to tell Spring in which packages to search for the classes annotated with stereotype annotations. Observe that the model package is not specified because it doesn’t contain classes annotated with stereotype annotations.

! I use the *basePackages* attribute of the @ComponentScan annotation. Spring also offers the feature of directly specifying the classes (by using the **basePackageClasses** attribute of the same annotation). The advantage of defining the packages is that you only have to mention the package name. In case it contains 20 component classes, you write only one line (the name of the package) instead of 20. The disadvantage is that if a developer renames the package, they might not realize they also have to change the value of the @ComponentScan annotation. Mentioning the classes directly, you might write more, but when someone changes the code, they immediately see they also need to change the @ComponentScan annotation; otherwise, the app doesn’t compile.

Listing 4.12 The Main class

public class Main {

public static void main(String[] args) {

var context = new AnnotationConfigApplicationContext(ProjectConfiguration.class);

var comment = new Comment();

comment.setAuthor("Laurentiu");

comment.setText("Demo comment");

var commentService = context.getBean(CommentService.class);

commentService.publishComment(comment);

}

}

Running the application demonstrates that the two dependencies were accessed and correctly called by the CommentService object:

Storing comment: Demo comment

Sending notification for comment: Demo comment

It might not look like Spring improves a lot the experience. By using the DI feature, we don’t create the instance of the CommentService object and its dependencies ourselves, and we don’t need to explicitly make the relationship between them. In a real-world scenario, where you have more than 3 classes, letting Spring manage the objects and dependencies among them really makes a difference. It eliminates boilerplate code, which allows you to focus on what the application does. And remember that adding these instances to the context enables Spring to control and augment them with features.

**PROJECT 15 – CommentApp – using field DI**

The only thing we need to change is the CommentService class. We remove the constructor and mark the fields of the class with the @Autowired annotation:

@Component

public class CommentService {

@Autowired ❶

private CommentRepository commentRepository; ❶

@Autowired ❶

private CommentNotificationProxy commentNotificationProxy; ❶

public void publishComment(Comment comment) {

commentRepository.storeComment(comment);

commentNotificationProxy.sendComment(comment);

}

}

❶ Fields are no longer final, and they are marked with @Autowired. Spring uses the default constructor to create the instance of the class and then injects the two dependencies from its context

**PROJECT 16 – CommentApp – using @Bean DI**

I completely removed the stereotype annotation (@Component) of the CommentService class and its two dependencies.

I changed the configuration class to create these beans and establish the relationships among them:

@Configuration ❶

public class ProjectConfiguration {

@Bean ❷

public CommentRepository commentRepository() {

return new DBCommentRepository();

}

@Bean ❷

public CommentNotificationProxy commentNotificationProxy() {

return new EmailCommentNotificationProxy();

}

@Bean

public CommentService commentService(

CommentRepository commentRepository, ❸

CommentNotificationProxy commentNotificationProxy) {

return new CommentService(commentRepository, commentNotificationProxy);

}

}

❶ Because we don’t use stereotype annotations, we no longer need to use the @ComponentScan annotation.

❷ We create a bean for each of the 2 dependencies.

❸ We use parameters of the @Bean method (which are now defined with the interface type) to instruct Spring to provide references for beans from its context, compatible with the type of the parameters

**PROJECT 17 – CommentApp – @Primary with multiple implementations**

What happens if the Spring context contains more instances that match a requested abstraction.

Suppose we have two beans created with two different classes that implement the CommentNotificationProxy interface. If more than one bean of the same type exists in the Spring context, you need to tell Spring which of these beans to inject.



When using dependency injection on the interface, you need to instruct Spring which is the implementation it should inject.

Let’s add a new class, CommentPushNotificationProxy (which implements the CommentNotificationProxy interface)

Listing 4.13 A new implementation of the CommentNotificationProxy interface

@Component

public class CommentPushNotificationProxy implements CommentNotificationProxy { ❶

@Override

public void sendComment(Comment comment) {

System.out.println("Sending push notification for comment: " + comment.getText());

}

}

❶ The class implements the CommentNotificationProxy interface

If you run this application as-is, you’ll get an exception because Spring doesn’t know which of the two beans in its context to choose for injection – NoUniqueBeanDefinitionException.

MARKING AN IMPLEMENTATION AS DEFAULT FOR INJECTION WITH @PRIMARY

The first solution is using @Primary. The only thing you need to do is add @Primary near the @Component annotation to mark the implementation provided by this class as the default for implementation, as shown in the following listing.

Listing 4.14 Using @Primary to mark the implementation as default

@Component

@Primary ❶

public class CommentPushNotificationProxy

implements CommentNotificationProxy {

@Override

public void sendComment(Comment comment) {

System.out.println( "Sending push notification for comment: " + comment.getText());

}

}

❶ Using @Primary, we mark this implementation as a default for dependency injection.

Observe that Spring indeed injected the implementation provided by the newly created class:

Storing comment: Demo comment

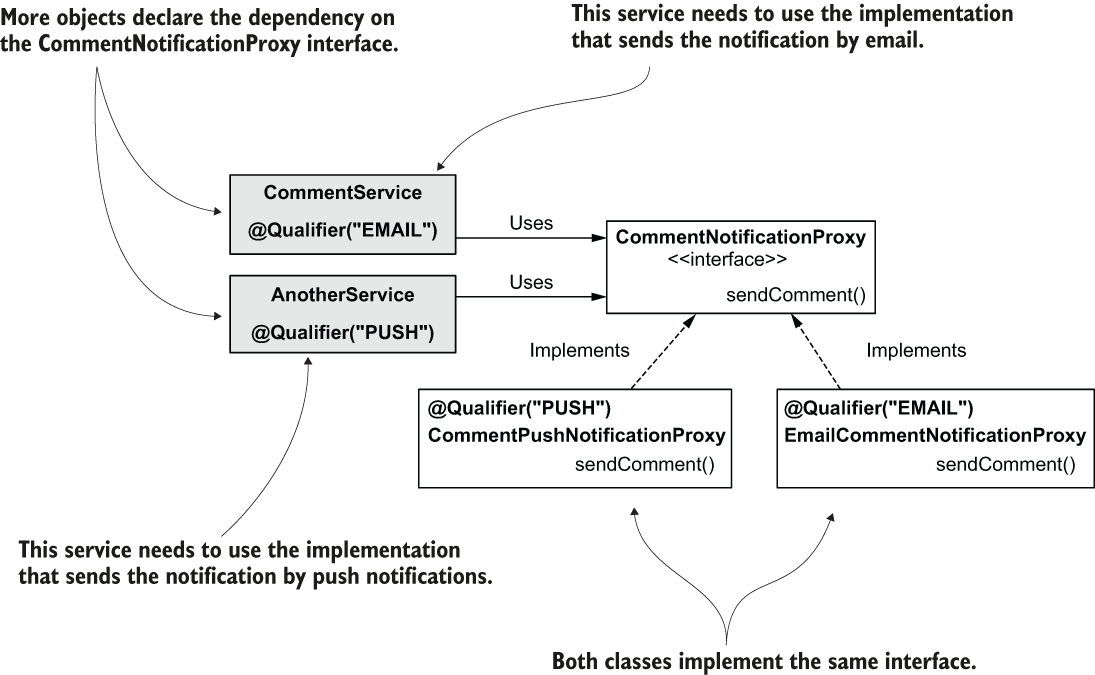
Sending push notification for comment: Demo comment ❶

❶ Spring injected the new implementation because we marked it as primary.

Sometimes you use dependencies that already provide implementations for specific interfaces. When you need to have custom implementations of those interfaces, you can use @Primary to mark your implementation as a default for DI. This way, Spring knows to inject the implementation you define and not the one provided by the dependency.

**PROJECT 18 – CommentApp – @Qualifier with multiple implementations**

Sometimes, in production apps, you need to define more implementations of the same interface, and different objects use these implementations.



If different objects need to use different implementations of the same contract, we can use @Qualifier to name them and tell Spring where and what it needs to inject.

Тhe CommentPushNotification class:

@Component

@Qualifier("PUSH") ❶

public class CommentPushNotificationProxy implements CommentNotificationProxy {

// Omitted code

}

❶ Using @Qualifier, we name this implementation “PUSH.”

The EmailCommentNotificationProxy class:

@Component

@Qualifier("EMAIL") ❶

public class EmailCommentNotificationProxy implements CommentNotificationProxy {

// Omitted code

}

❶ Using @Qualifier, we name this implementation “EMAIL.”

When you want Spring to inject one of these, you just need to specify the implementation’s name using the @Qualifier annotation again. Here is how to inject a specific implementation as a dependency of the CommentService object:

Listing 4.15 Specifying the implementation Spring needs to inject with @Qualifier

@Component

public class CommentService {

private final CommentRepository commentRepository;

private final CommentNotificationProxy commentNotificationProxy;

public CommentService( ❶

CommentRepository commentRepository,

@Qualifier("PUSH") CommentNotificationProxy commentNotificationProxy) {

this.commentRepository = commentRepository;

this.commentNotificationProxy = commentNotificationProxy;

}

// Omitted code

}

❶ For each parameter where we want to use a specific implementation, we annotate the parameter with @Qualifier.

Spring injects the dependency you specified using @Qualifier when you run the app:

Storing comment: Demo comment

Sending push notification for comment: Demo comment ❶

❶ Observe that Spring injected the implementation for push notifications.

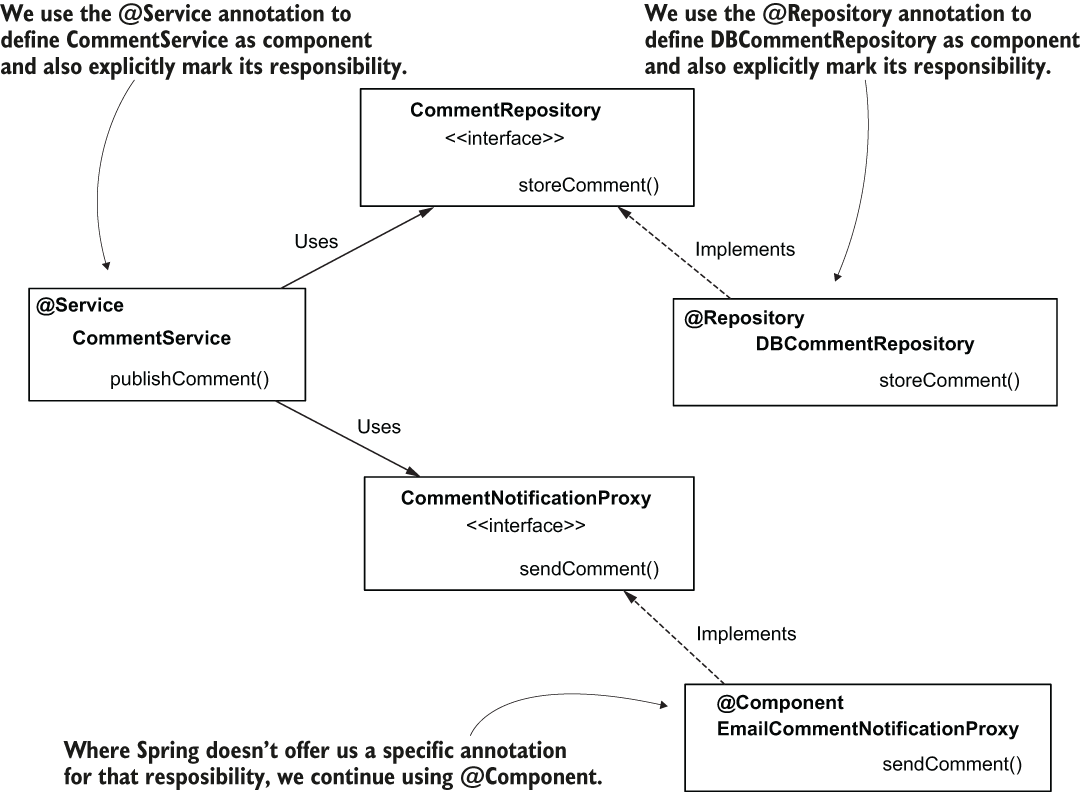
**PROJECT 19 – CommentApp – @Service and @Repository – stereotype annotations**

**Object responsibilities with stereotype annotations**

In real-world projects, it’s a common practice to define the component’s purpose using the stereotype annotation explicitly. Using @Component is generic and gives you no detail about the responsibility of the object you’re implementing. But developers generally use objects with some known responsibilities. Two of the responsibilities are the service and the repository, for which we use 2 more stereotype annotations: **@Service** and **@Repository**.

The services are the objects with the responsibility of implementing the use cases, while repositories are the objects managing the data persistence. Because these responsibilities are so common in projects, and they are important in the class design, having a distinctive way of marking them helps the developer better understand the app design.

All 3 (@Component, @Service, and @Repository) are stereotype annotations and instruct Spring to create and add an instance of the annotated class to its context.



Where Spring doesn’t offer a specific annotation for that responsibility, we continue to use @Component.

In our examples, you mark the CommentService class with @Service instead of @Component. This way, you explicitly mark the object’s responsibility and make this aspect more visible for any developer reading the class.

@Service ❶

public class CommentService {

// Omitted code

}

❶ We use @Service to define this object as a component having the responsibility of service.

Similarly, you explicitly mark the repository class’s responsibility using the @Repository annotation:

@Repository ❶

public class DBCommentRepository implements CommentRepository {

// Omitted code

}

❶ We use @Repository to define this object as a component with the responsibility of the repository.

**Summary**

* Decoupling implementations through abstractions is a good practice in implementing a class design. Decoupling objects makes implementations easy to change without affecting too many parts of the application. This aspect makes your application more easily extended and maintained.
* In Java, we use interfaces to decouple implementations. We also say that we define contracts between implementations through interfaces.
* When using abstraction with dependency injection, Spring knows to search for a bean created with an implementation of the requested abstraction.
* You use stereotype annotations on classes for which Spring needs to create instances and add these instances as beans to its context. You never use stereotype annotations on interfaces.
* When the Spring context has more beans created with multiple implementations of the same abstraction, to instruct Spring which bean to inject, you can
  + use the @Primary annotation to mark one of them as default, or
  + use the @Qualifier annotation to name the bean and then instruct Spring to inject that bean by name.
* When we have service responsibility components, we use the @Service stereotype annotation instead of @Component. Likewise, when a component has repository responsibility, we use the @Repository stereotype annotation instead of @Component. This way, we mark the component’s responsibility explicitly, and we make the class design more comfortable to read and understand.

# Chapter 5 The Spring context: Bean scopes and life cycle

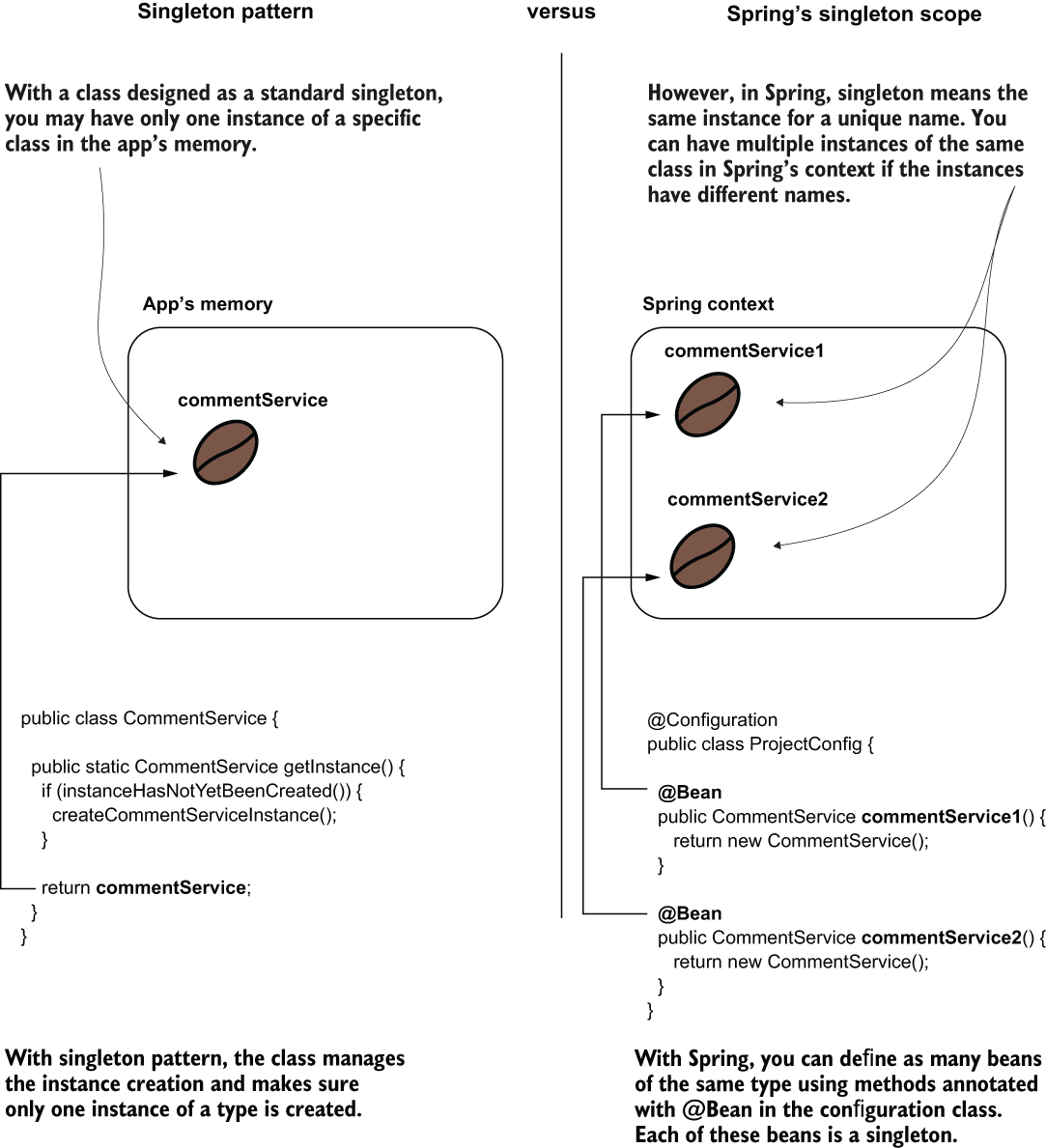
Spring has multiple different approaches for creating beans and managing their life cycle, and in the Spring world we name these approaches scopes. Here we will discuss 2 scopes you’ll often find in Spring apps: *singleton* and *prototype*.

The singleton bean scope defines Spring’s default approach for managing the beans in its context, and it’s what we’ve been using up to now. It is also the bean scope you’ll most encounter in production apps.

**5.1.1 How singleton beans work**

Spring creates a singleton bean when it loads the context and assigns the bean a name (sometimes also referred to as bean ID). We name this scope singleton because you always get the same instance when you refer to a specific bean. But be careful! You can have more instances of the same type in the Spring context if they have different names.

But if you know what singleton pattern is, the way it works in Spring might look strange to you because you have only one instance of a type in the app. For Spring, the singleton concept allows multiple instances of the same type, and singleton means unique per name but not unique per app.



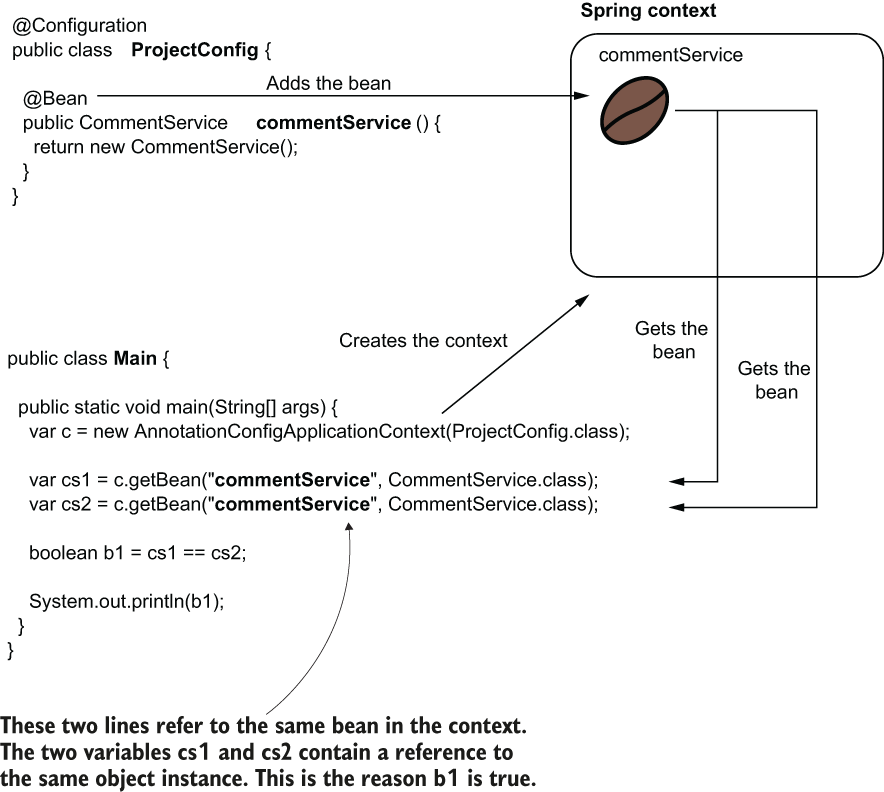
When one refers to a singleton class in an app, they mean a class that offers only one instance to the app and manages the creation of that instance. In Spring, however, singleton doesn’t mean the context has only one instance of that type. It just means that a name is assigned to the instance, and the same instance will always be referred through that name.

**PROJECT 20 – singleton-scoped beans with @Bean**

***DECLARING SINGLETON-SCOPED BEANS WITH @BEAN***

Let’s demonstrate a singleton bean’s behavior with an example using the @Bean annotation to add an instance to the Spring context and then simply refer to it multiple times in a main class. We do this to prove we get the same instance every time we refer to the bean.

Figure 5.2 is a visual representation of the context near the code that configures it. The coffee bean in the visual represents the instance that Spring adds to its context. Observe that the context contains only one instance (coffee bean) with an associated name. As we discussed in chapter 2, when using the @Bean annotation approach to add a bean to the context, the name of the method annotated with @Bean becomes the bean’s name.



The app initializes the context when starting and adds a bean. In this case, we use the approach with the @Bean annotation to declare the bean. The name of the method becomes the identifier of the bean. Wherever you use that identifier, you get a reference to the same instance.

The result would have been the same if we used stereotype annotations (like @Component) to add the bean to the context.

Note that I’ve explicitly used the bean name when getting the bean from the Spring context. You know that when you have only one bean of a kind in the Spring context, you no longer need to use its name. You can get that bean by its type. In this example, I used the name simply to enforce that we refer to the same bean. I could have just referred to the type, and in both cases we would get the reference to the same (and only) instance of CommentService in the context.

We need to define an empty CommentService class. You then write the configuration class and the main class, as in figure 5.2:

public class CommentService {

}

In the next listing, you find the configuration class definition, which uses a method annotated with @Bean to add an instance of type CommentService to the Spring context.

Listing 5.1 Adding a bean to the Spring context

@Configuration

public class ProjectConfig {

@Bean ❶

public CommentService commentService() {

return new CommentService();

}

}

❶ Adds the CommentService bean to the Spring context

In the next listing, you find the Main class we use to test Spring’s behavior for our singleton bean. We get the reference to the CommentService bean twice, and we expect to get the same reference each time.

Listing 5.2 The Main class used to test Spring’s behavior for the singleton bean

public class Main {

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

var cs1 = c.getBean("commentService", CommentService.class);

var cs2 = c.getBean("commentService", CommentService.class);

boolean b1 = cs1 == cs2; ❶

System.out.println(b1);

}

}

❶ Because the two variables hold the same reference, the result of this operation is true.

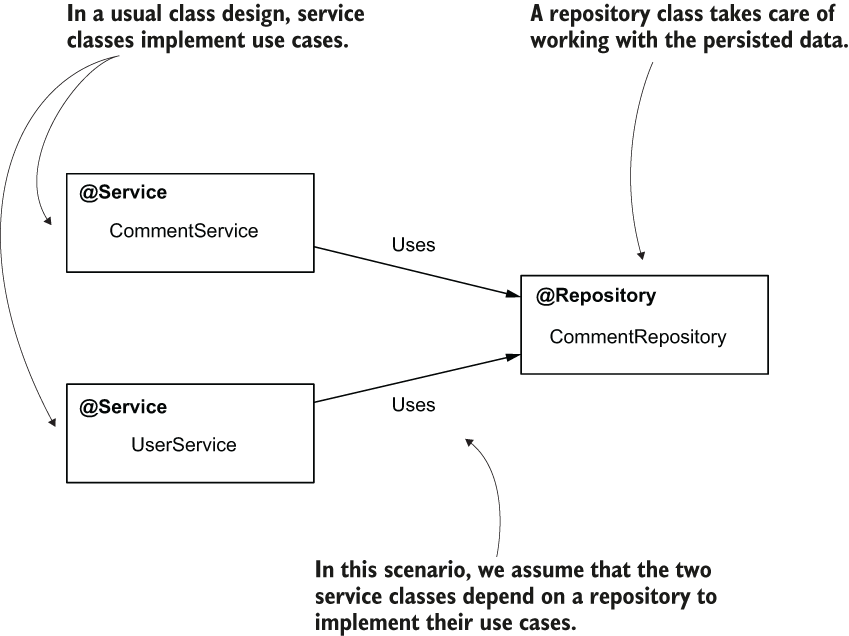
Running the app will print “true” in the console because, being a singleton bean, Spring returns the same reference every time.

**PROJECT 21 – singleton-scoped beans with @Component/ @Service/ @Repository**

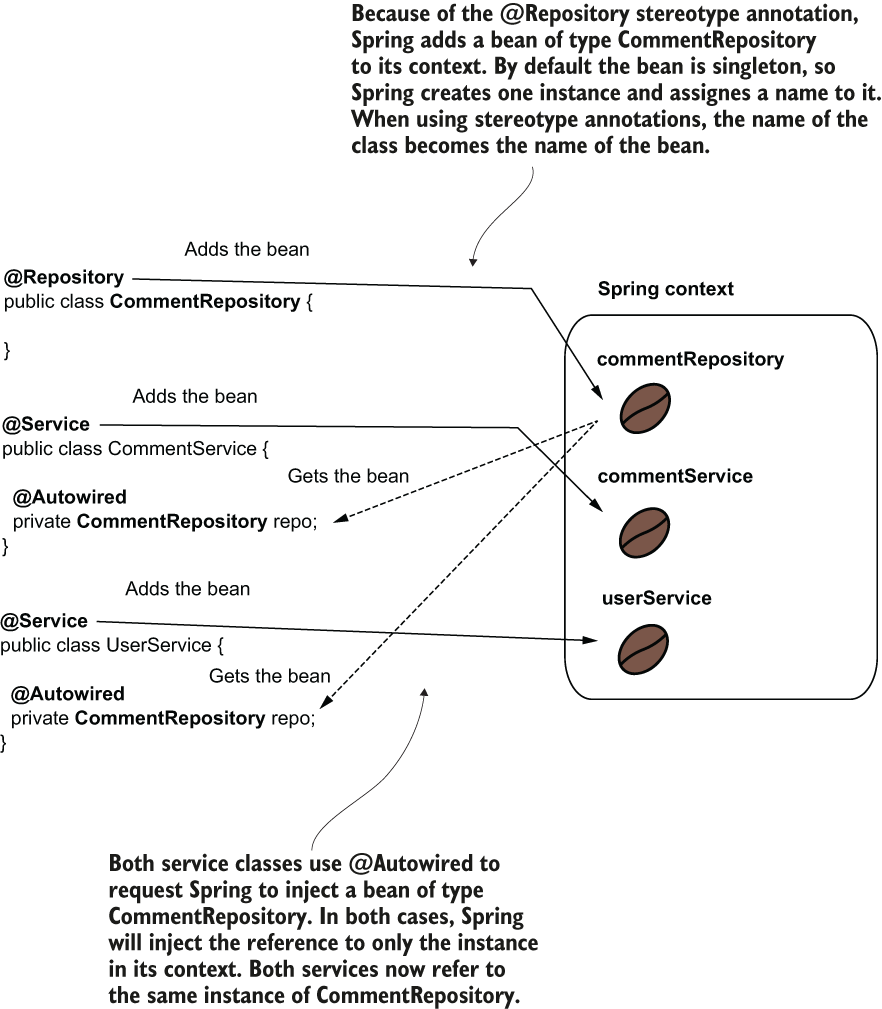
***DECLARING SINGLETON-SCOPED BEANS STEREOTYPE ANNOTATIONS***

Spring’s behavior for singleton beans isn’t any different when using stereotype annotations than when you declared them with the @Bean annotation.

Consider a class design scenario where two service classes depend on a repository. Say we have both CommentService and UserService depending on a repository named CommentRepository:



Two service classes depend on a repository to implement their use cases. When designed as singleton beans, Spring’s context will have one instance of each of these classes.



The beans are also singleton-scoped when using stereotype annotations to create them. When using @Autowired to request Spring to inject a bean reference, the framework injects the reference to the singleton bean in all the requested places.

Let’s prove this behavior by creating the three classes and comparing the references Spring injects in the service beans.

@Repository

public class CommentRepository {

}

Here is the definition of the CommentService class. Observe that I used @Autowired to instruct Spring to inject an instance of type CommentRepository in an attribute declared in the class. I also defined a getter method that I intend to use later to prove Spring injects the same object reference in both service beans:

@Service

public class CommentService {

@Autowired

private CommentRepository commentRepository;

public CommentRepository getCommentRepository() {

return commentRepository;

}

}

Following the same logic for CommentService for the UserService class:

@Service

public class UserService {

@Autowired

private CommentRepository commentRepository;

public CommentRepository getCommentRepository() {

return commentRepository;

}

}

Unlike the previous example, the configuration class remains empty. We only need to tell Spring where to find the classes annotated with stereotype annotations - we use the @ComponentScan annotation:

@Configuration

@ComponentScan(basePackages = {"services", "repositories"})

public class ProjectConfig {

}

In the Main class, we get the references for the 2 services, and we compare their dependencies to prove that Spring injected the same instance in both.

Listing 5.3 Testing Spring’s behavior for injecting the singleton bean in the Main class

public class Main {

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext( ❶

ProjectConfig.class);

var s1 = c.getBean(CommentService.class); ❷

var s2 = c.getBean(UserService.class); ❷

boolean b = ❸

s1.getCommentRepository() == s2.getCommentRepository();

System.out.println(b); ❹

}

}

❶ Creates the Spring context based on the configuration class

❷ Gets the references of the two service beans in the Spring context

❸ Compares the references for the repository dependency injected by Spring

❹ Because the dependency (CommentRepository) is singleton, both services contain the same reference, so this line prints “true.”

**5.1.2 Singleton beans in real-world scenarios**

Let’s start by considering some scenarios where you should or shouldn’t use singleton beans.

Because the singleton bean scope assumes that multiple components of the app can share an object instance, the most important thing to consider is that these beans must be *immutable*. Most often, a real-world app executes actions on multiple threads (e.g., any web app). In such a scenario, multiple threads share the same object instance. If these threads change the instance, you encounter a race-condition scenario (figure 5.5).

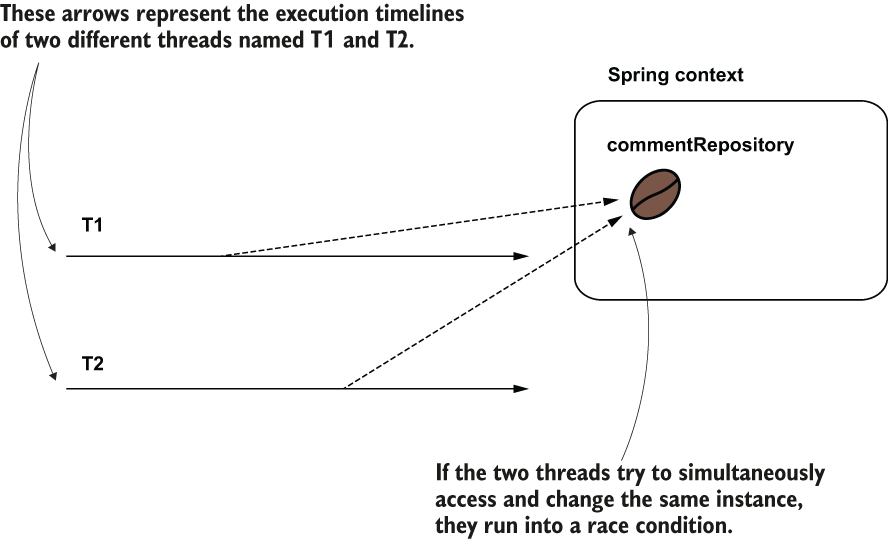


Figure 5.5 When multiple threads access a singleton bean, they access the same instance. If these threads try to change the instance simultaneously, they run into a race condition. The race condition causes unexpected results or execution exceptions if the bean is not designed for concurrency.

A race condition is a situation that can happen in multithreaded architectures when multiple threads try to change a shared resource. In case of a race condition, the developer needs to properly synchronize the threads to avoid unexpected execution results or errors.

If you want mutable singleton beans (whose attributes change), you need to make these beans concurrent by yourself (mainly by employing thread synchronization). But singleton beans aren’t designed to be synchronized. They’re commonly used to define an app’s backbone class design and delegate responsibilities one to another. Technically, synchronization is possible, but it’s not a good practice. Synchronizing the thread on a concurrent instance can dramatically affect the app’s performance. In most cases, you will find other means to solve the same problem and avoid thread concurrency.

**PROJECT 22 – singleton-scoped beans – constructor DI**

Constructor DI is a good practice and preferred over field injection. 1 of the advantages of constructor injection is that it allows you to make the instance immutable (define the bean’s fields as final). In our previous example, we can enhance the CommentService class’s definition by replacing the field injection with constructor injection. A better design of the class would look like the following code snippet:

@Service

public class CommentService {

private final CommentRepository commentRepository; ❶

public CommentService(CommentRepository commentRepository) {

this.commentRepository = commentRepository;

}

public CommentRepository getCommentRepository() {

return commentRepository;

}

}

❶ Making the field final highlights that this field was intended not to be changed.

**3 main points when using beans:**

1. Make an object bean in the Spring context only if you need Spring to manage it so that the framework can augment that bean with a specific capability. If the object doesn’t need any capability offered by the framework, you don’t need to make it a bean.
2. If you need to make an object bean in the Spring context, it should be singleton only if it’s immutable. Avoid designing mutable singleton beans.
3. If a bean needs to be mutable, an option could be to use the prototype scope.

**5.1.3 Using eager and** **lazy instantiation**

In most cases, Spring creates all singleton beans when it initializes the context—this is Spring’s default behavior. We’ve used only this default behavior, which is also called eager instantiation. With lazy instantiation, Spring doesn’t create the singleton instances when it creates the context. Instead, it creates each instance the first time someone refers to the bean.

**PROJECT 23 – eager instantiation**

In our initial scenario, we only need a bean to test the default (eager) initialization.

I’ll name this class CommentService. You make this class a bean, either using the @Bean annotation approach or a stereotype annotation. But either way, make sure to add an output to the console in the class’s constructor. This way, we’ll easily observe if the framework calls it:

@Service

public class CommentService {

public CommentService() {

System.out.println("CommentService instance created!");

}

}

@Configuration

@ComponentScan(basePackages = {"services"})

public class ProjectConfig {

}

In the Main class, we only instantiate the Spring context. A critical aspect to observe is that no one uses the CommentService bean. However, Spring will create and store the instance in the context. We know that Spring creates the instance because we’ll see the output from the CommentService bean class’s constructor when running the app.

public class Main {

public static void main(String[] args) { ❶

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

}

}

❶ This app creates the Spring context, but it doesn’t use the CommentService bean anywhere.

Even if the app doesn’t use the bean anywhere, when running the app you’ll find the following output in the console:

CommentService instance created!

**PROJECT 24 – lazy instantiation**

Now change by adding the @Lazy annotation above the class (for stereotype annotations approach) or above the @Bean method (for the @Bean method approach). You’ll observe the output no longer appears in the console when running the app because we instructed Spring to create the bean only when someone uses it. And, in our example, nobody uses the CommentService bean.

@Service

@Lazy ❶

public class CommentService {

public CommentService() {

System.out.println("CommentService instance created!");

}

}

❶ The @Lazy annotation tells Spring that it needs to create the bean only when someone refers to the bean for the first time.

Change the Main class and add a reference to the CommentService bean:

public class Main {

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

System.out.println("Before retrieving the CommentService");

var service = c.getBean(CommentService.class); ❶

System.out.println("After retrieving the CommentService");

}

}

❶ At this line, where Spring needs to provide a reference to the CommentService bean, Spring also creates the instance.

Rerun the app, and you’ll find the output again in the console. The framework creates the bean only if it’s used:

Before retrieving the CommentService

CommentService instance created!

After retrieving the CommentService

When should you use eager instantiation and when should you use lazy? In most cases, it’s more comfortable to let the framework create all the instances at the beginning when the context is instantiated (eager); this way, when one instance delegates to another, the second bean already exists in any situation.

In a lazy instantiation, the framework has to first check if the instance exists and eventually create it if it doesn’t, so from the performance point of view, it’s better to have the instances in the context already (eager) because it spares some checks the framework needs to do when one bean delegates to another. Another advantage of eager instantiation is when something is wrong and the framework cannot create a bean; we can observe this issue when starting the app. With lazy instantiation, someone would observe the issue only when the app is already executing and it reaches the point that the bean needs to be created.

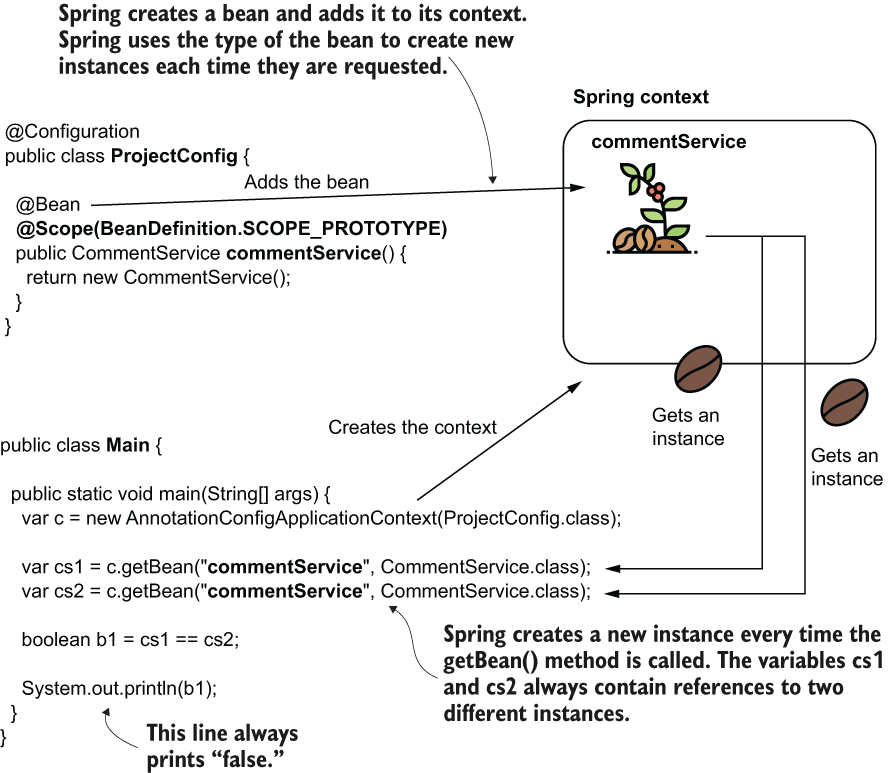
But lazy instantiation is not all evil. Some time ago, I worked on a vast monolithic application. This app was installed in different locations where it was used in various scopes by its clients. In most cases, a specific client didn’t use a big part of the functionality, so instantiating the beans together with the Spring context unnecessarily occupied a lot of memory. For that app, the developers designed most of the beans to be lazily instantiated so that the app would create only the necessary instances.

My advice is to go with the default, which is an eager instantiation. This approach generally brings more benefits. If you find yourself in a situation like the one I presented with the monolithic app, first see if you can do something about the app’s design. Often, the need for using lazy instantiation is a sign something might be wrong with the app’s design. For example, in my story, it would have been better if the app had been designed in a modular way or as microservices. Such an architecture would have helped the developers deploy only what specific clients needed, and then making the instantiation of the beans lazy wouldn’t have been necessary.

**5.2 Using the prototype bean scope**

**5.2.1 How prototype beans work**

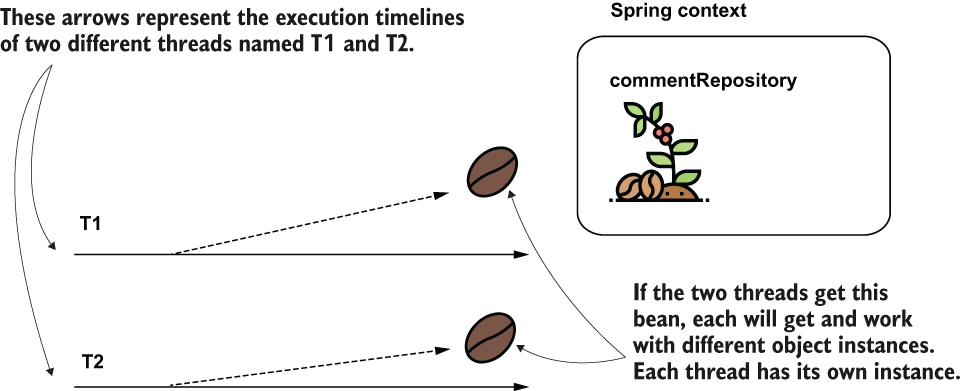
Every time you request a reference to a prototype-scoped bean, Spring creates a new object instance. For prototype beans, Spring doesn’t create and manage an object instance directly. The framework manages the object’s type and creates a new instance every time someone requests a reference to the bean. In figure 5.6, I represented the bean as a coffee plant (every time you request a bean, you get a new instance).



We use the @Scope annotation to change the bean scope in prototype. The bean is now represented as a coffee plant because you get a new object instance each time you refer to it. For this reason, variables cs1 and cs2 will always contain different references, so the output of the code is always “false.”

When you create the bean using the @Bean annotation approach, @Scope goes together with @Bean over the method that declares the bean. When declaring the bean with stereotype annotations, you use the @Scope annotation and the stereotype annotation over the class that declares the bean.

With prototype beans, we no longer have concurrency problems because each thread that requests the bean gets a different instance, so defining mutable prototype beans is not a problem.



When multiple threads request a certain prototype bean, each thread gets a different instance. This way, the threads cannot run into a race condition.

**PROJECT 25 – prototype scope over @Bean**

We create a bean named CommentService and declare it as prototype to prove we get a new instance every time we request that bean.

public class CommentService {

}

We define a bean with the CommentService class in the configuration class.

Listing 5.4 Declaring the prototype bean in the configuration class

@Configuration

public class ProjectConfig {

@Bean

@Scope(BeanDefinition.SCOPE\_PROTOTYPE) ❶

public CommentService commentService() {

return new CommentService();

}

}

❶ Makes this bean prototype-scoped

To prove that every time we request the bean we get a new instance, we create a Main class and request the beans twice from the context. We observe that the references we get are different.

Listing 5.5 Testing Spring’s behavior for the prototype bean in the Main class

public class Main {

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

var cs1 = c.getBean("commentService", CommentService.class);

var cs2 = c.getBean("commentService", CommentService.class);

boolean b1 = cs1 == cs2; ❶

System.out.println(b1); ❷

}

}

❶ The two variables cs1 and cs2 contain references to different instances.

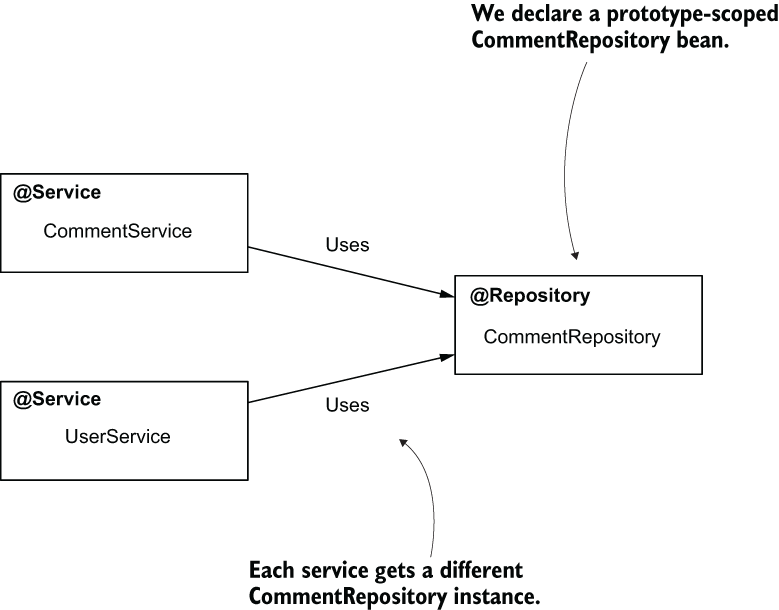
❷ This line always prints “false” in the console.

When you run the app, you’ll see it always displays “false” in the console. This output proves that the 2 instances received when calling the getBean() method are different.

**PROJECT 26 – prototype scope over @Component/@Service/@Repository (stereotype annotations)**

Let’s observe the behavior for auto-wiring prototype-scoped beans. We’ll define a CommentRepository prototype bean, and we inject the bean using @Autowired in two other service beans. We will observe that each service bean has a reference to a different instance of CommentRepository. This time the CommentRepository bean is prototype.

Figure 5.8 Each service class requests an instance of CommentRepository. Because CommentRepository is a prototype bean, each service gets a different CommentRepository instance.



@Repository

@Scope(BeanDefinition.SCOPE\_PROTOTYPE)

public class CommentRepository {

}

The 2 service classes request an instance of type CommentRepository using the @Autowired annotation.

@Service

public class CommentService {

@Autowired

private CommentRepository commentRepository;

public CommentRepository getCommentRepository() {

return commentRepository;

}

}

In the configuration class, we need to use the @Component-Scan annotation to tell Spring where to find the classes annotated with stereotype annotations:

@Configuration

@ComponentScan(basePackages = {"services", "repositories"})

public class ProjectConfig {

}

We add the Main class to our project and test how Spring injects the CommentRepository bean.

Listing 5.6 Testing Spring’s behavior for injecting the prototype bean in the Main class

public class Main {

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

var s1 = c.getBean(CommentService.class); ❶

var s2 = c.getBean(UserService.class); ❶

boolean b = ❷

s1.getCommentRepository() == s2.getCommentRepository();

System.out.println(b);

}

}

❶ Gets references from the context for the service beans

❷ Compares the references for the injected CommentRepository instances. Because CommentRepository is a prototype bean, the result of the comparison is always false.

**PROJECT 27 – prototypе scope for mutating object**

**5.2.2 Prototype beans in real-world scenarios**

Now we focus on the use cases and where you should use prototype-scoped beans in production apps. We’ll consider the discussed characteristics and analyze which scenarios prototype beans are good for and where should you avoid them (by using singleton beans).

You won’t find prototype beans as often as you’ll find singleton beans. But there is a good pattern you can use to decide if a bean should be prototype. Remember that singleton beans are not quite good friends with mutating objects. Say you design an object named CommentProcessor that processes the comments and validates them. A service uses the CommentProcessor object to implement a use case. But the CommentProcessor object stores the comment to be processed as an attribute, and its methods change this attribute (figure 5.9).

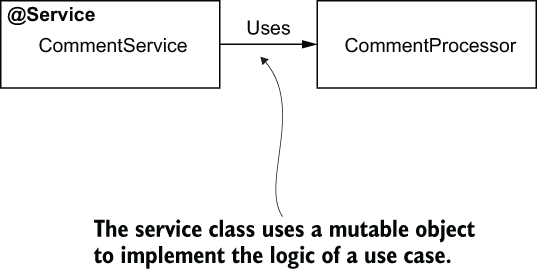


Figure 5.9 A service class uses a mutable object to implement the logic of a use case.

Here is the implementation of the CommentProcessor bean.

Listing 5.7 A mutable object; a potential candidate to the prototype scope

public class CommentProcessor {

private Comment comment;

public void setComment(Comment comment) {

this.comment = comment;

}

public void getComment() {

return this.comment;

}

public void processComment() { ❶

// changing the comment attribute

}

public void validateComment() { ❶

// validating and changing the comment attribute

}

}

❶ These 2 methods alter the value of the Comment attribute.

The next listing presents this service that uses the CommentProcessor class to implement a use case. The service method creates an instance of CommentProcessor using the class’s constructor and then uses the instance in the method’s logic.

Listing 5.8 A service using a mutable object to implement a use case

@Service

public class CommentService {

public void sendComment(Comment c) {

CommentProcessor p = new CommentProcessor(); ❶

p.setComment(c); ❷

p.processComment(c); ❷

p.validateComment(c); ❷

c = p.getComment(); ❸

// do something further

}}

❶ Creates a CommentProcessor instance

❷ Uses the CommentProcessor instance to alter the Comment instance

❸ Gets the modified Comment instance and uses it further

The CommentProcessor object is not even a bean in the Spring context. If we leave our scenario like this, the CommentProcessor object doesn’t need to be a bean at all.

But let’s suppose further that the CommentProcessor bean needs to use an object CommentRepository to persist some data, and CommentRepository is a bean in the Spring context (figure 5.10).

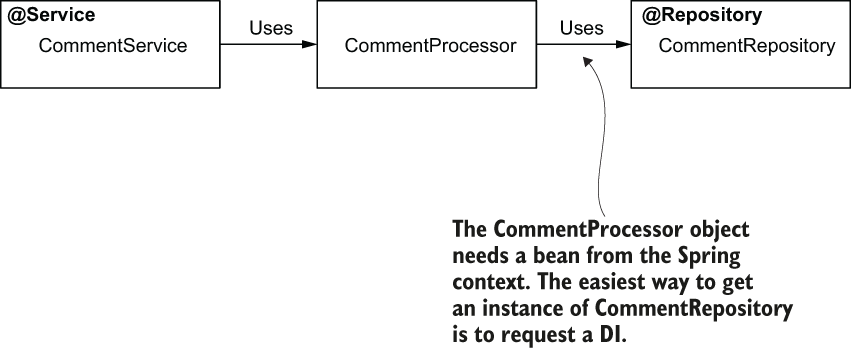


Figure 5.10 If the CommentProcessor object needs to use an instance of CommentRepository, the easiest way to get an instance is to request a DI. But to do this, Spring needs to know about CommentProcessor, so the CommentProcessor object needs to be a bean in the context.

In this scenario, the CommentProcessor bean needs to become a bean to benefit from the DI capability Spring offers.

We make CommentProcessor a bean in the Spring context. But can it be singleton-scoped? No. If we define this bean as singleton and multiple threads use it concurrently, we get into a race condition. We would not be sure which comment provided by which thread is processed and if the comment was processed correctly. In this scenario, we want each method call to get a different instance of the CommentProcessor object. We can change the CommentProcessor class to be a prototype bean:

@Component

@Scope(BeanDefinition.SCOPE\_PROTOTYPE)

public class CommentProcessor {

@Autowired

private CommentRepository commentRepository;

// Omitted code

}

You can now get an instance of CommentProcessor from the Spring context. But be careful! You need this instance for every call of the sendComment() method in the CommentService, so the request to the bean should be inside the method itself. To achieve such a result, you can directly inject the Spring context (ApplicationContext) into the CommentService bean using @Autowired. In the sendComment() method, you retrieve the CommentProcessor instance using getBean() from the application context.

Listing 5.9 Using CommentProcessor as prototype bean

@Service

public class CommentService {

@Autowired

private ApplicationContext context;

public void sendComment(Comment c) {

CommentProcessor p =

context.getBean(CommentProcessor.class); ❶

p.setComment(c);

p.processComment(c);

p.validateComment(c);

c = p.getComment();

// do something further

}}

❶ A new CommentProcessor instance is always provided here.

**PROJECT 28 – direct DI of a mutating prototype bean into singleton bean = singleton X**

Don’t make the mistake of injecting the CommentProcessor directly in the CommentService bean. The CommentService bean is a singleton, which means that Spring creates only an instance of this class. As a consequence, Spring will also inject the dependencies of this class just once when it creates the CommentService bean itself. In this case, you’ll end up with only an instance of the CommentProcessor. Each call of the sendComment() method will use this unique instance, so with multiple threads you’ll run into the same race condition issues as with a singleton bean. The next listing presents this approach. Use this as an exercise to try out and prove this behavior.

Listing 5.10 Injecting a prototype into a singleton

@Service

public class CommentService {

@Autowired

private CommentProcessor p; ❶

public void sendComment(Comment c) {

p.setComment(c);

p.processComment(c);

p.validateComment(c);

c = p.getComment();

// do something further

}}

❶ Spring injects this bean when creating the CommentService bean. But because CommentService is singleton, Spring will also create and inject the CommentProcessor just once.

I generally avoid using prototype beans, and mutable instances in general, in the apps I develop. But sometimes you need to refactor or work with old applications. In my case, I faced such a scenario when I worked in an app refactoring for adding Spring to an old application. That app used mutating objects in many places, and refactoring all these places in a short time was impossible. We needed to use prototype bean, which allowed the team to refactor each of these cases progressively.

Table 5.1 A quick comparison between singleton and prototype bean scopes

|  |  |
| --- | --- |
| **Singleton** | **Prototype** |
| 1. The framework associates a name with an actual object instance. 2. Every time you refer to a bean name you’ll get the same object instance. 3. You can configure Spring to create the instances when the context is loaded or when first referred. (eager or lazy) 4. Singleton is the default bean scope in Spring. 5. It’s not recommended that a singleton bean to have mutable attributes. | 1. A name is associated with a type. 2. Every time you refer to a bean name, you get a new instance. 3. The framework always creates the object instances for the prototype scope when you refer to the bean. (lazy) 4. You need to explicitly mark a bean as a prototype. 5. A prototype bean can have mutable attributes. |

**Summary**

* In Spring, the scope of beans defines how the framework manages the object instances.
* Spring offers 2 bean scopes: singleton and prototype.
  + With singleton, Spring manages the object instances directly in its context. Each instance has a unique name, and using that name you always refer to that specific instance. Singleton is Spring’s default.
  + With prototype, Spring considers only the object type. Each type has a unique name associated with it. Spring creates a new instance of that type every time you refer to the bean name.
* You can configure Spring to create a singleton bean either when the context is initialized (eager) or when the bean is referred for the first time (lazy). By default, a bean is eagerly instantiated.
* In apps, we most often use singleton beans. Because anyone referring to the same name gets the same object instance, multiple different threads could access and use this instance. For this reason, it’s advisable to have the instance immutable. If, however, you prefer to have mutating operations on the bean’s attribute, it’s your responsibility to take care of the thread synchronization.
* If you need to have a mutable object like a bean, using the prototype scope could be a good option.
* Be careful with injecting a prototype-scoped bean into a singleton-scoped bean. When you do something like this, the singleton instance always uses the same prototype instance, which Spring injects when it creates the singleton instance. This is usually a vicious design because the point of making a bean prototype-scoped is to get a different instance for every use.

# Chapter 6 Using aspects with Spring AOP

Here, you’ll learn how to use the technique supported by the IoC principle: aspects.

Aspects are a way the framework intercepts method calls and possibly alters the execution of methods. You can affect the execution of specific method calls you select. This technique helps you extract part of the logic belonging to the executing method. In certain scenarios, decoupling a part of the code, helps make that method easier to understand (figure 6.1). It allows the developer to focus only on the relevant details discussed when reading the method logic. In this chapter, we’ll discuss how to implement aspects and when you should use them. If you don’t use aspects carefully, you might end up with a less maintainable app. This approach is called aspect-oriented programming (AOP).



Sometimes it’s not relevant to have parts of the code in the same place with the business logic because it makes the app more difficult to understand. A solution is to move part of the code aside from the business logic implementation using aspects. In this scene, Jane, the programmer, is discouraged by the logging lines written together with the business code. Count Dracula shows her the magic of aspects by decoupling the logs into an aspect.

Another important reason for learning aspects is that Spring uses them in implementing a lot of the crucial capabilities it offers. Understanding how the framework works can save you many hours of debugging later when you face a specific problem. A pertinent example of Spring capability that uses aspects is transactionality. Transactionality is one of the main capabilities most apps use today to keep the persisted data’s consistency. Another important capability relying on aspects is security configurations, which help your app protect its data and make sure data cannot be seen or changed by unwanted individuals.

**6.1 How aspects work in Spring**

An aspect is a piece of logic the framework executes when you call specific methods of your choice. When designing an aspect, you define the following:

* ***What code*** you want Spring to execute when you call specific methods. This is named an **aspect**. (the method)
* ***When*** the app should execute this logic of the aspect (e.g., before or after the method call, instead of the method call). This is named the **advice**. (the order of methods)
* ***Which methods*** the framework needs to intercept and execute the aspect ***for*** them. This is named a **pointcut**. (over which method)

With aspects terminology, you’ll also find the concept of **a join point**, which defines the ***event that triggers the execution*** of an aspect. But with Spring, this event is always a method call.

As in the case of the dependency injection, to use aspects you need the framework to manage the objects for which you want to apply aspects. ***The bean that declares the method*** intercepted by an aspect is named **the target object**.

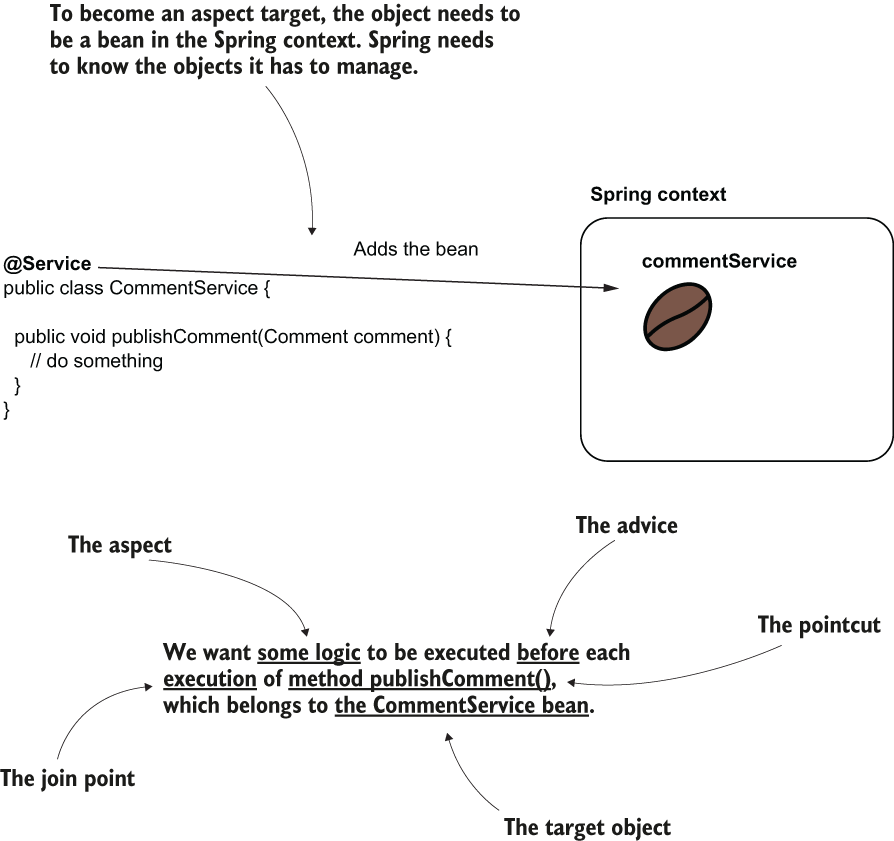


Figure 6.2 The aspect terminology: Spring executes some logic (the aspect) when someone calls a specific method (the pointcut). We need to specify when the logic is executed according to the pointcut (e.g., before). The when is the advice. For Spring to intercept the method, the object that defines the intercepted method needs to be a bean in the Spring context. So, the bean becomes the target object of the aspect.

But how does Spring intercept each method call and apply the aspect logic? As said, the object needs to be a bean in the Spring context. But because you made the object an aspect target, Spring won’t directly give you an instance reference for the bean when you request it from the context. Instead, Spring gives you an object that calls the aspect logic instead of the actual method. We say that Spring gives you a ***proxy object*** instead of the real bean. You will now receive the proxy instead of the bean anytime you get the bean from the context, either if you directly use the getBean() method of the context or if you use DI (figure 6.3). This approach is named **weaving**. (тъчене)



Figure 6.3 Weaving an aspect. Instead of giving you a reference to the real bean, Spring gives you a reference to a proxy object, intercepts the method calls, and manages the aspect logic.

In figure 6.4, you find a comparison between calling the method when it isn’t intercepted by an aspect versus an aspect intercepting the method call. You observe that calling an aspected method assumes you call the method through the proxy object provided by Spring. The proxy applies the aspect logic and delegates the call to the actual method.

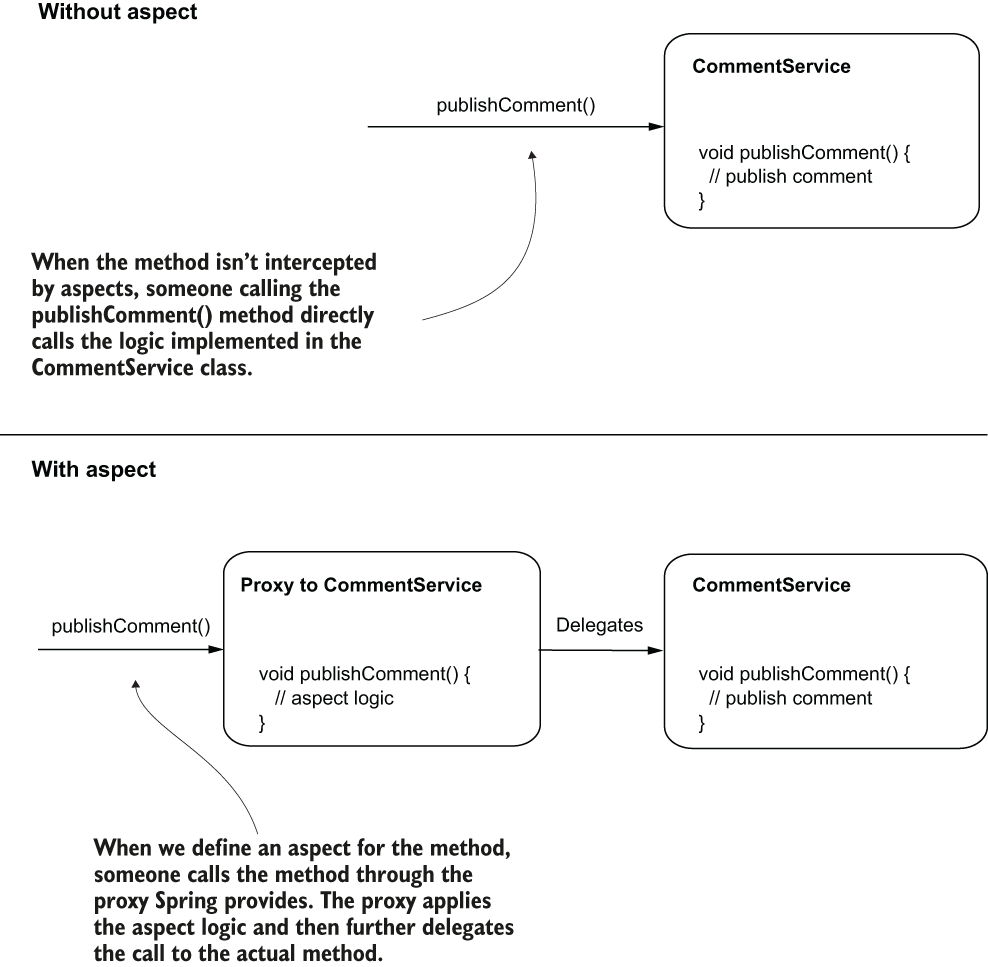


Figure 6.4 When a method isn’t aspected, the call goes directly to that method. When we define an aspect for a method, the call goes through the proxy object. The proxy object applies the logic defined by the aspect and then delegates the call to the real method.

**PROJECT 29 – add Logger – without aspect**

**6.2 Implementing aspects with Spring AOP**

Suppose you have an application that implements multiple use cases in its service classes. Some new regulations require your app to store the time it started and ended for each use case execution. In your team, you decided to take responsibility for implementing a functionality to log all the events where a use case begins and ends.

We’ll use an aspect to solve this scenario in the simplest way possible.

In addition to the spring-context dependency, for this example we also need the spring-aspects dependency.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.2.8.RELEASE</version>

</dependency>

<dependency> ❶

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>5.2.8.RELEASE</version>

</dependency>

❶ We need this dependency to implement the aspects.

We’ll only consider one service object named CommentService and a use case it defines named publishComment(Comment comment). This method, defined in the CommentService class, receives a parameter of type Comment. Comment is a model class:

public class Comment {

private String text;

private String author;

// Omitted getters and setters

}

Next there is the definition of the CommentService class. We annotate the CommentService class with the @Service stereotype annotation to make it a bean in the Spring context. The CommentService class defines the publishComment(Comment comment) method, representing our scenario’s use case.

In this example instead of *System.out*, I used an object of type *Logger* to write messages in the console. In real-world apps, you’ll generally use a logging framework that offers you more flexibility in customizing the logging features and standardizing the logging messages. Some good options for a logging framework are as follows:

* Log4j (<https://logging.apache.org/log4j/2.x/> )
* Logback (<http://logback.qos.ch/> )
* Java Logging API, which comes with the JDK (<http://mng.bz/v4Xq> )

The logging frameworks are compatible with any Java app, whether it’s using Spring or not. We can start to use these additional frameworks in our examples to familiarize with syntaxes closer to production-ready apps.

Listing 6.1 The Service class

@Service ❶

public class CommentService {

private Logger logger = ❷

Logger.getLogger(CommentService.class.getName());

public void publishComment(Comment comment) { ❸

logger.info("Publishing comment:" + comment.getText());

}}

❶ We use the stereotype annotation to make this a bean in the Spring context.

❷ To log a message in the app’s console every time someone calls the use case, we use a logger object.

❸ This method defines the use case for our demonstration.

In this example, I use the JDK logging capabilities to avoid adding other dependencies to our project. When declaring a logger object, you need to give it a name as a parameter. This name then appears in the logs and makes it easy for you to observe the log message source. Often, we use the class name, which I did in our example: CommentService.class.getName().

We also need to add a configuration class to tell Spring where to look for the classes annotated with stereotype annotations. In my case, I added the service class in the package named "services", and this is what I need to specify with the @ComponentScan annotation:

@Configuration

@ComponentScan(basePackages = "services") ❶

public class ProjectConfig {

}

❶ We use @ComponentScan to tell Spring where to search for classes annotated with stereotype annotations.

Let’s write the Main class that calls the publishComment() method in the service class and observe the current behavior.

Listing 6.2 The Main class we use to test the app’s behavior

public class Main {

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

var service = c.getBean(CommentService.class); ❶

Comment comment = new Comment(); ❷

comment.setText("Demo comment");

comment.setAuthor("Natasha");

service.publishComment(comment); ❸

}}

❶ Gets the CommentService bean from the context

❷ Creates a Comment instance to give as a parameter to the publishComment() method

❸ Calls the publishComment() method

If you run the app, you’ll observe an output in the console similar to what you see in the next snippet:

Sep 26, 2020 12:39:53 PM services.CommentService publishComment

INFO: Publishing comment:Demo comment

You will see the output generated by the publishComment() method. This is how the app looks before we solve the example we discussed. Remember, we need to print messages in the console before and after the service method call.

**PROJECT 30 – add Logger – using aspect class**

Let’s now enhance the project with an aspect class that intercepts the method call and adds an output before and after the call.

To create an aspect, you follow these steps (figure 6.5):

1. Enable the aspect mechanism in your Spring app by annotating the configuration class with the **@EnableAspectJAutoProxy** annotation.
2. Create a new class, and annotate it with the **@Aspect** annotation. Using either @Bean or stereotype annotations, add a bean for this class in the Spring context.
3. Define a method that will implement the aspect logic and tell Spring when and which methods to intercept using an advice annotation.
4. Implement the aspect logic.

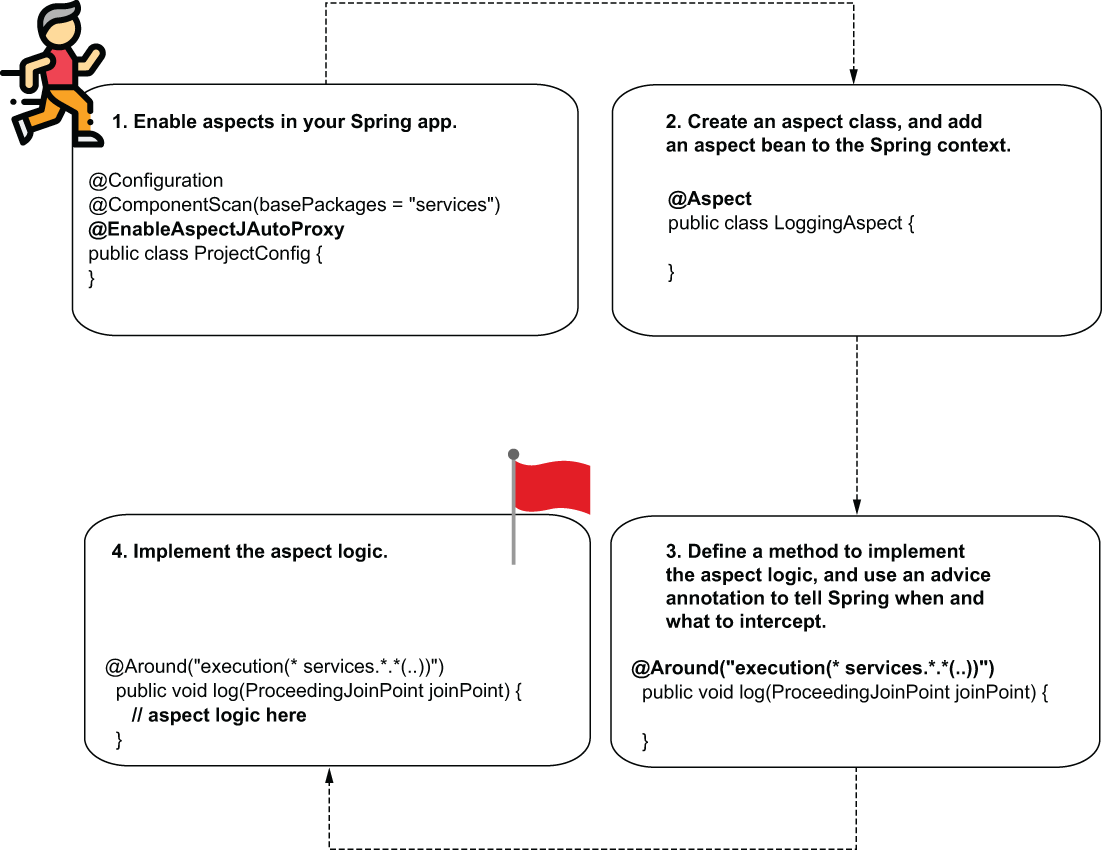


Figure 6.5 To implement an aspect, you follow 4 easy steps:

* + 1. you need to enable the aspect capability in your app.
    2. Then you create an aspect class,
    3. Define a method, and instruct Spring when and what to intercept.
    4. Finally, you implement the aspect logic.

*STEP 1: ENABLING THE ASPECTS MECHANISM FOR YOUR APPLICATION*

You need to tell Spring you’ll use aspects in your app. Whenever you use a specific mechanism provided by Spring, you have to explicitly enable it by annotating your configuration class with a particular annotation. In most cases, the names of these annotations start with “Enable.”

In this example, we need to use the **@EnableAspectJAutoProxy** annotation to enable the aspect capabilities.

Listing 6.3 Enabling the aspects mechanism in a Spring app

@Configuration

@ComponentScan(basePackages = "services")

@EnableAspectJAutoProxy ❶

public class ProjectConfig {

}

❶ Enables the aspects mechanism in our Spring app

*STEP 2: CREATE A CLASS THAT DEFINES THE ASPECT, AND ADD AN INSTANCE FOR THIS CLASS IN THE SPRING CONTEXT*

We need to create a new bean in the Spring context that defines the aspect. This object holds the methods, which will intercept specific method calls and augment them with specific logic.

Listing 6.4 Defining an aspect class

@Aspect

public class LoggingAspect {

public void log() {

// To implement later

}}

Add an instance of this class to the Spring context.

@Configuration

@ComponentScan(basePackages = "services")

@EnableAspectJAutoProxy

public class ProjectConfig {

@Bean ❶

public LoggingAspect aspect() {

return new LoggingAspect();

}

}

❶ Adds an instance of the LoggingAspect class to the Spring context

Also, the @Aspect annotation isn’t a stereotype annotation. Using @Aspect, you tell Spring that the class implements the definition of an aspect, but Spring won’t also create a bean for this class. You need to explicitly create a bean for your class and allow Spring to manage it this way. It’s a common mistake to forget that annotating the class with @Aspect doesn’t also add a bean to the context.

*STEP 3: USE AN ADVICE ANNOTATION TO TELL SPRING WHEN AND WHICH METHOD CALLS TO INTERCEPT*

Now that we have defined the aspect class, we choose the advice and annotate the method accordingly. In the next listing, you see how I annotated the method with the **@Around** annotation.

Listing 6.5 Using an advice annotation to weave the aspect to specific methods

@Aspect

public class LoggingAspect {

@Around("execution(\* services.\*.\*(..))") ❶

public void log(ProceedingJoinPoint joinPoint) {

joinPoint.proceed(); ❷

}}

❶ Defines which are the intercepted methods

❷ Delegates to the actual intercepted method

Other than using the @Around annotation, you also observe a string expression as the value of the annotation, and I have added a parameter to the aspect method.

The expression used as a parameter to the @Around annotation tells Spring which method calls to intercept. This expression language is called *AspectJ pointcut language*, and you won’t need to learn it by heart. In practice, you don’t use complex expressions. When I need to write such an expression, I always refer to the documentation (<http://mng.bz/4K9g> ).

Theoretically, you can write very complex AspectJ pointcut expressions to identify a particular set of method calls to be intercepted. This language is really powerful. But as we’ll discuss later in this chapter, it’s always better to avoid writing complex expressions. In most cases, you can find simpler alternatives.

Look at the expression I used (figure 6.6). It means Spring intercepts any method defined in a class that is in the services package, regardless of the method’s return type, the class it belongs to, the name of the method, or the parameters the method receives.

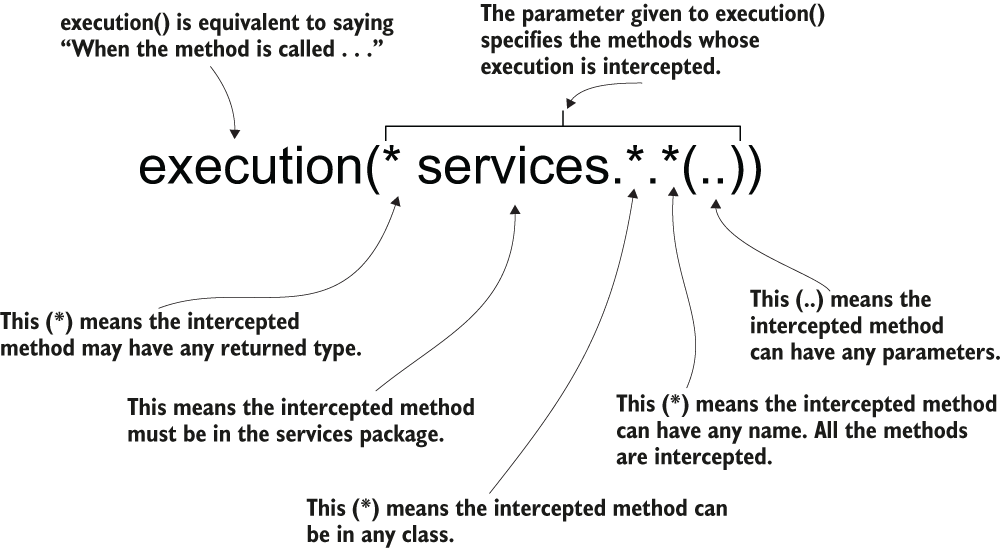


Figure 6.6 The AspectJ pointcut expression used in the example. It tells Spring to intercept the calls for all the methods in the services package, regardless of their return type, the class they belong to, name, or the parameters they receive.

Now let’s look at the second element I’ve added to the method: the ProceedingJoinPoint parameter, which represents the intercepted method. The main thing you do with this parameter is tell the aspect when it should delegate further to the actual method.

*STEP 4: IMPLEMENT THE ASPECT LOGIC*

In listing 6.6, I’ve added the logic for our aspect. Now the aspect

1. Intercepts the method
2. Displays something in the console before calling the intercepted method
3. Calls the intercepted method
4. Displays something in the console after calling the intercepted method

Figure 6.7 visually presents the aspect’s behavior

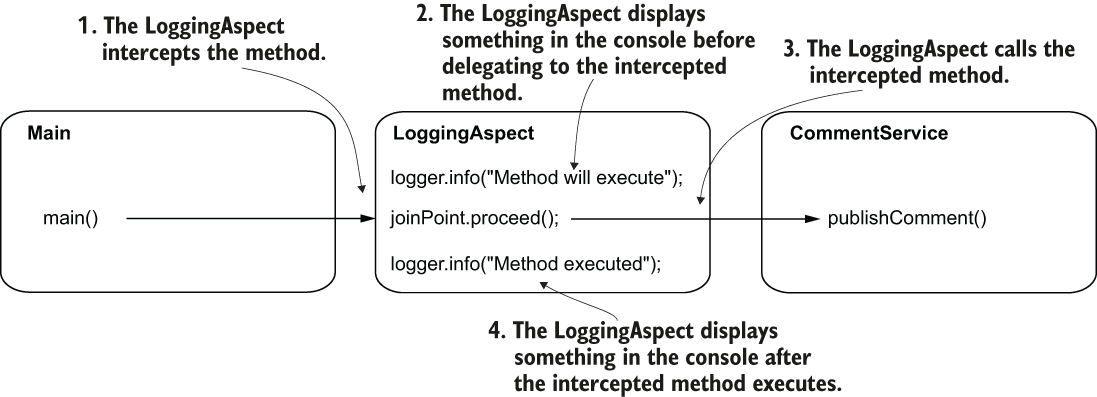


Figure 6.7 The aspect behavior. LoggingAspect wraps the method execution by displaying something before and after the method call. This way, you observe a simple implementation of an aspect.

Listing 6.6 Implementing the aspect logic

@Aspect

public class LoggingAspect {

private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

@Around("execution(\* services.\*.\*(..))")

public void log(ProceedingJoinPoint joinPoint) throws Throwable {

logger.info("Method will execute"); ❶

joinPoint.proceed(); ❷

logger.info("Method executed"); ❸

}

}

❶ Prints a message in the console before the intercepted method’s execution

❷ Calls the intercepted method

❸ Prints a message in the console after the intercepted method’s execution

The method *proceed()* of the ProceedingJoinPoint parameter calls the intercepted method, *publishComment()*, of the CommentService bean. If you don’t call proceed(), the aspect never delegates further to the intercepted method (figure 6.8).

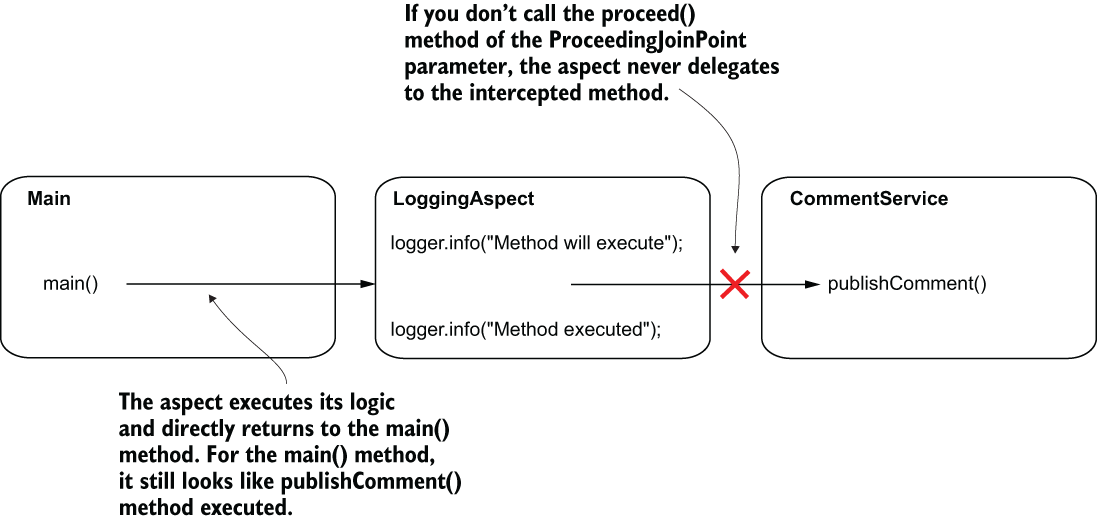


Figure 6.8 If you don’t call the proceed() method of the ProceedingJoinPoint parameter of the aspect, the aspect never delegates further to the intercepted method. In this case, the aspect simply executes instead of the intercepted method. The caller of the method doesn’t know that the real method is never executed.

You can even implement logic where the actual method isn’t called anymore. For example, an aspect that applies some authorization rules decides whether to delegate further to a method the app protects. If the authorization rules aren’t fulfilled, the aspect doesn’t delegate to the intercepted method it protects (figure 6.9).

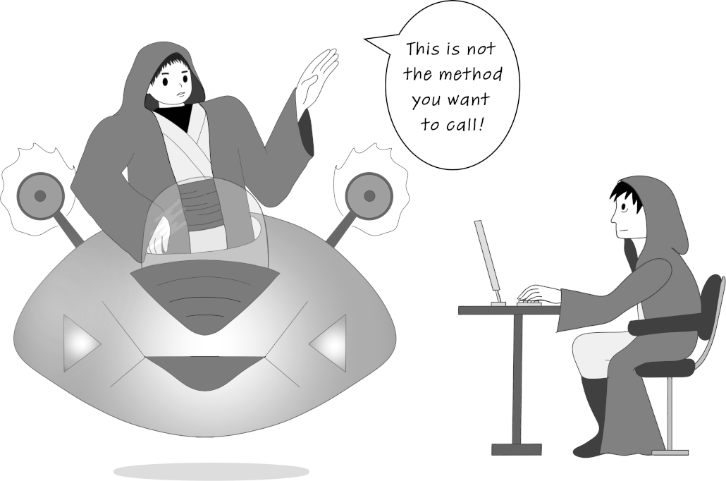


Figure 6.9 An aspect can decide not to delegate at all to the method it intercepts. This behavior looks like the aspect applies a mind trick to the caller of the method. The caller ends up executing another logic than the one it actually called.

Also, observe that the proceed() method throws a Throwable. The method proceed() is designed to throw any exception coming from the intercepted method. In this example, I chose the easy way to propagate it further, but you can use a try-catch-finally block to treat this throwable if you need it.

In the console output, you’ll find the logs from both the aspect and the intercepted method. The output you see should look similar to the one presented in the following snippet:

Sep 27, 2020 1:11:11 PM aspects.LoggingAspect log

INFO: Method will execute ❶

Sep 27, 2020 1:11:11 PM services.CommentService publishComment

INFO: Publishing comment:Demo comment ❷

Sep 27, 2020 1:11:11 PM aspects.LoggingAspect log

INFO: Method executed ❸

❶ This line is printed from the aspect.

❷ This line is printed from the actual method.

❸ This line is printed from the aspect.

**PROJECT 31 – add Logger – alter the intercepted method**

**6.2.2 Altering the intercepted method’s parameters and the returned value**

Aspects are really powerful. Not only can they intercept a method and alter its execution, but they can also intercept the parameters used to call the method and possibly alter them or the value the intercepted method returns.

Suppose you want to log the parameters used to call the service method and what the method returned.

Because we also refer to what the method returns, I changed the service method and made it return a value:

@Service

public class CommentService {

private Logger logger = Logger.getLogger(CommentService.class.getName());

public String publishComment(Comment comment) {

logger.info("Publishing comment:" + comment.getText());

return "SUCCESS"; ❶

}

}

❶ For our demonstration, the method now returns a value.

The **ProceedingJoinPoint** parameter of the aspect method represents the intercepted method. You can use this parameter to get any information related to the intercepted method (parameters, method name, target object, and so on). The next code snippet shows you how to get the method name and the parameters used to call the method before intercepting the call:

**String methodName = joinPoint.getSignature().getName();**

**Object [] arguments = joinPoint.getArgs();**

Now we can change the aspect also to log these details.

Listing 6.7 Obtaining the method name and parameters in the aspect logic

@Aspect

public class LoggingAspect {

private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

@Around("execution(\* services.\*.\*(..))")

public Object log(ProceedingJoinPoint joinPoint) throws Throwable {

String methodName = ❶

joinPoint.getSignature().getName(); ❶

Object [] arguments = joinPoint.getArgs(); ❶

logger.info("Method " + methodName + ❷

" with parameters " + Arrays.asList(arguments) +

" will execute");

Object returnedByMethod = joinPoint.proceed(); ❸

logger.info("Method executed and returned " + returnedByMethod);

return returnedByMethod; ❹

}

}

❶ Obtains the name and parameters of the intercepted method

❷ Logs the name and parameters of the intercepted method

❸ Calls the intercepted method

❹ Returns the value returned by the intercepted method

Observe how the aspect intercepts the call and can access the parameters and the returned value.

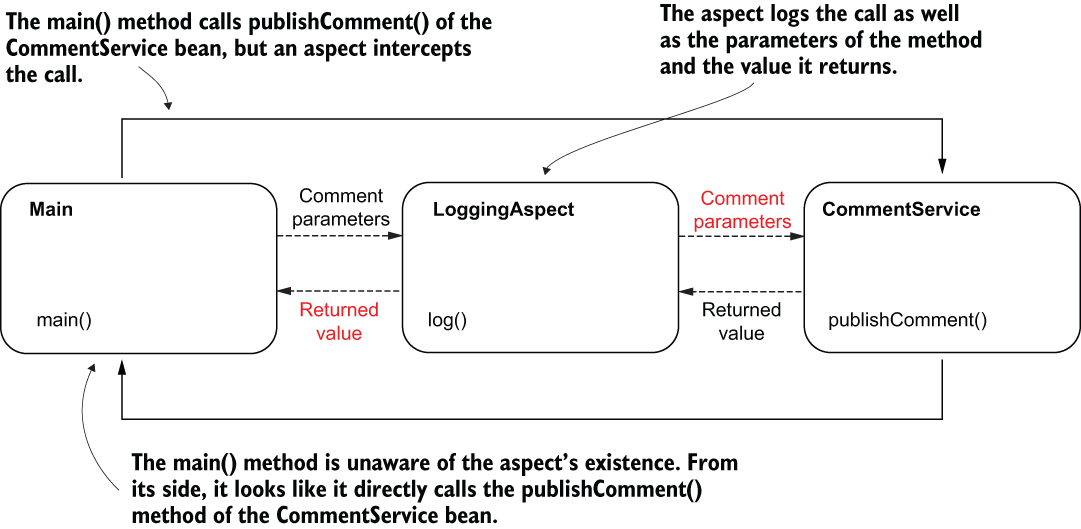


Figure 6.10 The aspect intercepts the method call, so it can access the parameters and the value returned by the intercepted method after execution. For the main() method, it looks like it directly calls the publishComment() method of the CommentService bean. The caller isn’t aware that an aspect intercepted the call.

I’ve changed the main() method to print the value returned by publishComment()

Listing 6.8 Printing the returned value to observe the aspect’s behavior

public class Main {

private static Logger logger = Logger.getLogger(Main.class.getName());

public static void main(String[] args) {

var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

var service = c.getBean(CommentService.class);

Comment comment = new Comment();

comment.setText("Demo comment");

comment.setAuthor("Natasha");

String value = service.publishComment(comment);

logger.info(value); ❶

}}

❶ Prints the value returned by the publishComment() method

When running the app, in the console you see the values logged from the aspect and the returned value logged by the main() method:

Sep 28, 2020 10:49:39 AM aspects.LoggingAspect log ❶

INFO: Method publishComment with parameters [Comment{text='Demo comment', ❶

➥ author='Natasha'}] will execute ❶

Sep 28, 2020 10:49:39 AM services.CommentService publishComment

INFO: Publishing comment:Demo comment ❷

Sep 28, 2020 10:49:39 AM aspects.LoggingAspect log

INFO: Method executed and returned SUCCESS ❸

Sep 28, 2020 10:49:39 AM main.Main main

INFO: SUCCESS ❹

❶ Parameters printed by the aspect

❷ Message printed by the intercepted method

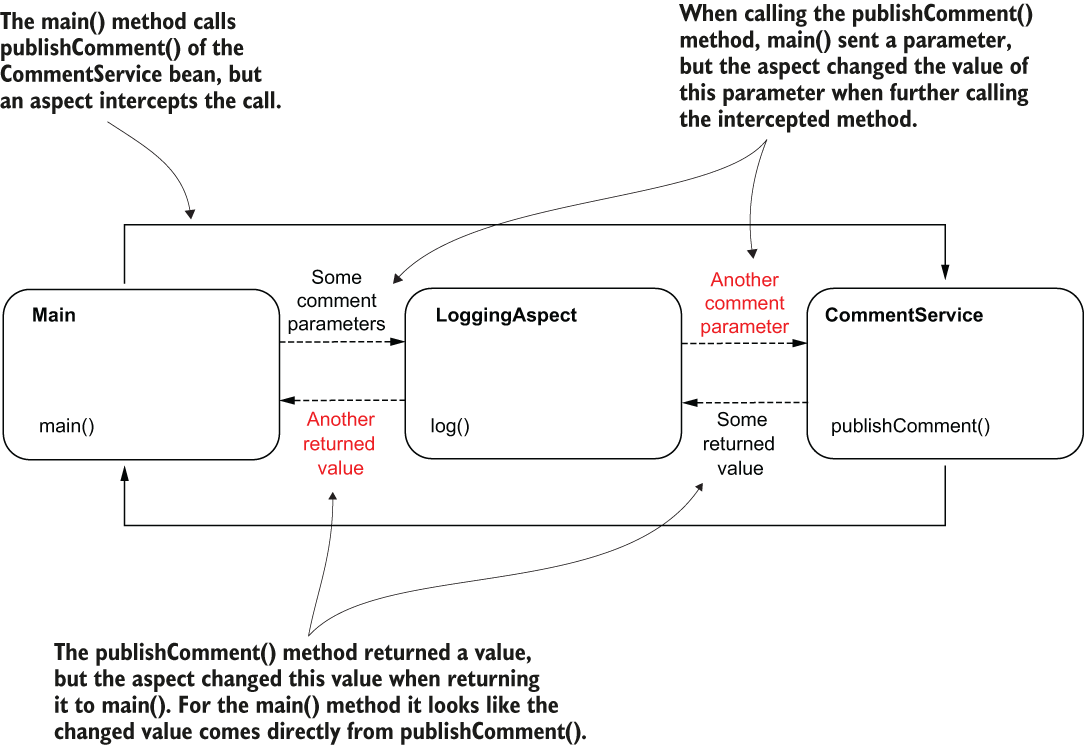
❸ Returned value printed by the aspect

❹ Returned value printed in main

But aspects are even more powerful. They can alter the execution of the intercepted method by

* Changing the value of the parameters sent to the method
* Changing the returned value received by the caller
* Throwing an exception to the caller or catching and treating an exception thrown by the intercepted method

You can be extremely flexible in altering the call of an intercepted method. You can even change its behavior completely (figure 6.11). But be careful! When you alter the logic through an aspect, you make a part of the logic transparent. Make sure you don’t hide things that aren’t obvious. The whole idea of decoupling a part of the logic is to avoid duplicating code and hide what’s irrelevant, so a developer can easily focus on the business logic code.



**PROJECT 32 – add Logger – alter parameters & return values of the method**

The following listing shows that when you call the proceed() method without sending any parameters, the aspect sends the original parameters to the intercepted method. But you can choose to provide a parameter when calling the proceed() method. This parameter is an array of objects that the aspect sends to the intercepted method instead of the original parameter values. The aspect logs the value returned by the intercepted method, but it returns to the caller a different value.

Listing 6.9 Altering the parameters and the returned value

@Aspect

public class LoggingAspect {

private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

@Around("execution(\* services.\*.\*(..))")

public Object log(ProceedingJoinPoint joinPoint) throws Throwable {

String methodName = joinPoint.getSignature().getName();

Object [] arguments = joinPoint.getArgs();

logger.info("Method " + methodName +

" with parameters " + Arrays.asList(arguments) +

" will execute");

Comment comment = new Comment();

comment.setText("Some other text!");

Object [] newArguments = {comment};

Object returnedByMethod = joinPoint.proceed(newArguments); ❶

logger.info("Method executed and returned " + returnedByMethod);

return "FAILED"; ❷

}}

❶ We send a different comment instance as a value to the method’s parameter.

❷ We log the value returned by the intercepted method, but we return a different value to the caller.

Running the app generates an output like the one in the next snippet. The values of the parameters received by the publishComment() method are different than the ones sent when calling the method. The publishComment() method returns a value, but main() gets a different one:

Sep 29, 2020 10:43:51 AM aspects.LoggingAspect log

INFO: Method publishComment with parameters [Comment{text='Demo comment',

➥ author='Natasha'}] will execute ❶

Sep 29, 2020 10:43:51 AM services.CommentService publishComment

INFO: Publishing comment:Some other text! ❷

Sep 29, 2020 10:43:51 AM aspects.LoggingAspect log

INFO: Method executed and returned SUCCESS ❸

Sep 29, 2020 10:43:51 AM main.Main main

INFO: FAILED ❹

❶ The publishComment() method is called with a comment having the text “Demo comment.”

❷ The publishComment() method receives a comment with the text “Some other text!”

❸ The method publishComment() returns “SUCCESS.”

❹ The returned value main() receives is “FAILED.”

Be careful with using aspects! You should only use them to hide irrelevant lines of code that can easily be implied. Aspects are so powerful they can bring you to the “dark side” of hiding relevant code and make your app more difficult to maintain. Use aspects with caution!

**PROJECT 33 – add Logger – @Around , custom annotation**

**6.2.3 Intercepting annotated methods**

In the CommentService class, we’ll add three methods: publishComment(), deleteComment(), and editComment().

We want to define a custom annotation and log only the execution of the methods we mark using the custom annotation. To achieve this objective, you need to do the following:

1. Define a custom annotation, and make it accessible at runtime. We’ll call this annotation @ToLog.
2. Use a different AspectJ pointcut expression for the aspect method to tell the aspect to intercept the methods annotated with the custom annotation.

We don’t need to change the logic of the aspect. For this example, our aspect does the same thing as the previous examples: logs the intercepted method’s execution.

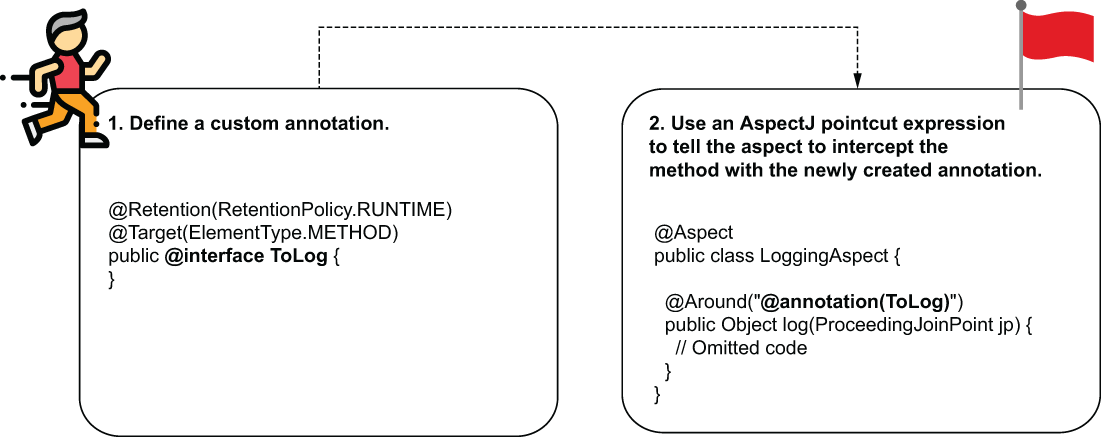


Figure 6.12 The steps for intercepted annotated methods. You need to create a custom annotation you want to use to annotate the methods your aspect needs to intercept. Then you use a different AspectJ pointcut expression to configure the aspect to intercept the methods annotated with the custom annotation you created.

Here is the declaration of the custom annotation. The definition of the retention policy with **@Retention(RetentionPolicy.RUNTIME)** is critical. By default, in Java annotations cannot be intercepted at runtime. The **@Target** annotation specifies which language elements we can use this annotation for. By default, you can annotate any elements, but it’s always a good idea to restrict the annotation to only what you make it for—in our case, methods:

@Retention(RetentionPolicy.RUNTIME) ❶

@Target(ElementType.METHOD) ❷

public @interface ToLog {

}

❶ Enables the annotation to be intercepted at runtime

❷ Restricts this annotation to only be used with methods

The CommentService class now defines 3 methods. We annotated only the deleteComment() method, so we expect the aspect will intercept only this one.

Listing 6.10 The CommentService class defining three methods

@Service

public class CommentService {

private Logger logger = Logger.getLogger(CommentService.class.getName());

public void publishComment(Comment comment) {

logger.info("Publishing comment:" + comment.getText());

}

@ToLog ❶

public void deleteComment(Comment comment) {

logger.info("Deleting comment:" + comment.getText());

}

public void editComment(Comment comment) {

logger.info("Editing comment:" + comment.getText());

}}

❶ We use the custom annotation for the methods we want the aspect to intercept

To weave the aspect to the methods annotated with the custom annotation (figure 6.13), we use the following AspectJ pointcut expression: @annotation(ToLog). This expression refers to any method annotated with the annotation named @ToLog (which is, in this case, our custom annotation). In the next listing, you now find the aspect class, which uses the new pointcut expression to weave the aspect logic to the intercepted methods.

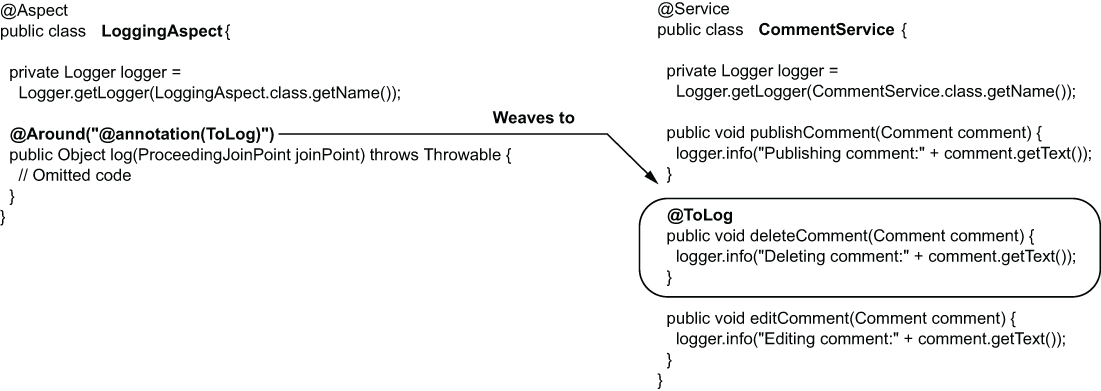


Figure 6.13 Using an AspectJ pointcut expression, we weave the aspect logic to any method annotated with the custom annotation we defined. This is a comfortable way to mark the methods to which specific aspect logic applies.

Listing 6.11 Changing the pointcut expression to weave aspect to annotated methods

@Aspect

public class LoggingAspect {

private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

@Around("@annotation(ToLog)") ❶

public Object log(ProceedingJoinPoint joinPoint) throws Throwable {

// Omitted code

}}

❶ Weaving the aspect to the methods annotated with @ToLog

When you run the app, only the annotated method (deleteComment(), in our case) is intercepted, and the aspect logs the execution of this method in the console. You should see an output in the console similar to the one:

Sep 29, 2020 2:22:42 PM services.CommentService publishComment

INFO: Publishing comment:Demo comment

Sep 29, 2020 2:22:42 PM aspects.LoggingAspect log ❶

INFO: Method deleteComment with parameters [Comment{text='Demo comment', ❶

➥ author='Natasha'}] will execute ❶

Sep 29, 2020 2:22:42 PM services.CommentService deleteComment ❶

INFO: Deleting comment:Demo comment ❶

Sep 29, 2020 2:22:42 PM aspects.LoggingAspect log ❶

INFO: Method executed and returned null ❶

Sep 29, 2020 2:22:42 PM services.CommentService editComment

INFO: Editing comment:Demo comment

❶ The aspect intercepts only the deleteComment() method, which we annotated with the custom @ToLog annotation.

**6.2.4 Other advice annotations you can use**

So far, we’ve used the advice annotation **@Around**. This is indeed the most used of the advice annotations in Spring apps because you can cover any implementation case: you can do things before, after, or even instead of the intercepted method. You can alter the logic any way you want from the aspect.

But you don’t necessarily always need all this flexibility. A good idea is to look for the most straightforward way to implement what you need to implement. Any app implementation should be defined by simplicity. By avoiding complexity, you make the app easier to maintain. For simple scenarios, Spring offers four alternative advice annotations that are less powerful than @Around. It’s recommended you use these when their capabilities are enough to keep the implementation simple.

Other than @Around, Spring offers the following advice annotations:

* **@Before**—Calls the method defining the aspect logic before the execution of the intercepted method.
* **@AfterReturning**—Calls the method defining the aspect logic after the method successfully returns, and provides the returned value as a parameter to the aspect method. The aspect method isn’t called if the intercepted method throws an exception.
* **@AfterThrowing**—Calls the method defining the aspect logic if the intercepted method throws an exception, and provides the exception instance as a parameter to the aspect method.
* **@After**—Calls the method defining the aspect logic only after the intercepted method execution, whether the method successfully returned or threw an exception.

You use these advice annotations the same way as for @Around. You provide them with an AspectJ pointcut expression to weave the aspect logic to specific method executions. The aspect methods don’t receive the ProceedingJoinPoint parameter, and they cannot decide when to delegate to the intercepted method. This event already happens based on the annotation’s purpose (for example, for @Before, the intercepted method call will always happen after the aspect logic execution).

**PROJECT 34 – add Logger – @AfterReturning, custom annotation**

In the next code snippet, you find the @AfterReturning annotation used. We use it the same way we did with @Around.

@Aspect

public class LoggingAspect {

private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

@AfterReturning(value = "@annotation(ToLog)", ❶

returning = "returnedValue") ❷

public void log(Object returnedValue) { ❸

logger.info("Method executed and returned " + returnedValue);

}}

❶ The AspectJ pointcut expression specifies which methods this aspect logic weaves to.

❷ Optionally, when you use @AfterReturning, you can get the value returned by the intercepted method. In this case, we add the “returning” attribute with a value that corresponds to the name of the method’s parameter where this value will be provided.

❸ The parameter name should be the same as the value of the “returning” attribute of the annotation or missing if we don’t need to use the returned value.

**PROJECT 35 – Logger and Security – no @Order**

**6.3 The aspect execution chain**

Suppose, for a method, we need to apply some security restrictions as well as log its executions. We have 2 aspects that take care of these responsibilities:

* SecurityAspect—Applies the security restrictions. This aspect intercepts the method, validates the call, and in some conditions doesn’t forward the call to the intercepted method.
* LoggingAspect—Logs the beginning and end of the intercepted method execution.

When you have multiple aspects weaved to the same method, they need to execute one after another. One way is to have the SecurityAspect execute first and then delegate to the LoggingAspect, which further delegates to the intercepted method. The second option is to have the LoggingAspect execute first and then delegate to the SecurityAspect, which eventually delegates further to the intercepted method. This way, the aspects create an execution chain.

The order in which the aspects execute is important because executing the aspects in different orders can have different results. Take our example: we know that the SecurityAspect doesn’t delegate the execution in all the cases, so if we choose this aspect to execute first, sometimes the LoggingAspect won’t execute. If we expect the LoggingAspect to log the executions that failed due to security restrictions, this isn’t the way we need to go (figure 6.14).

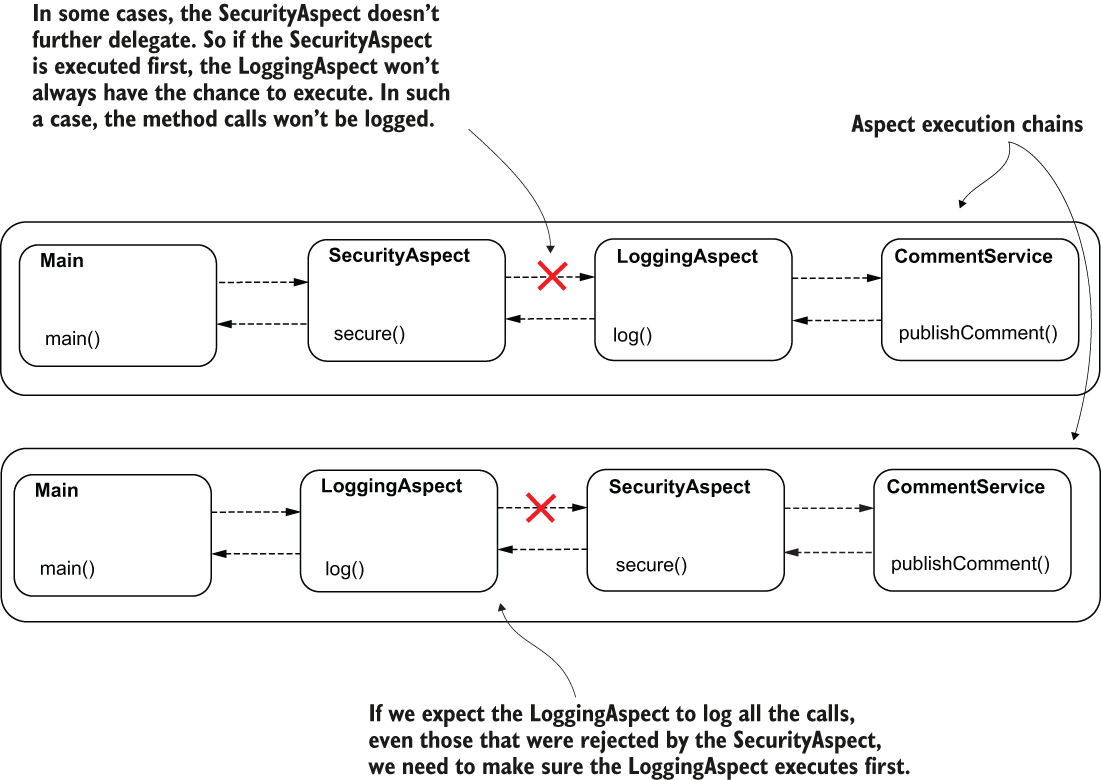


Figure 6.14 The aspect execution order matters. Depending on your app’s requirements, you need to choose a specific order for the aspects to execute. In this scenario, the LoggingAspect cannot log all the method executions if the SecurityAspect executes first.

By default, Spring doesn’t guarantee the order in which 2 aspects in the same execution chain are called. If the execution order is not relevant, then you just need to define the aspects and leave the framework to execute them in whatever order. If you need to define the aspects’ execution order, you can use the **@Order** annotation. This annotation receives an ordinal (a number) representing the order in the execution chain for a specific aspect. The smaller the number, the earlier that aspect executes. If 2 values are the same, the order of execution is again not defined.

I define 2 aspects that intercept the publishComment() method of a CommentService bean. First is the aspect named LoggingAspect.We don’t initially define any order for our aspects.

Listing 6.12 The implementation of the LoggingAspect class

@Aspect

public class LoggingAspect {

private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

@Around(value = "@annotation(ToLog)")

public Object log(ProceedingJoinPoint joinPoint) throws Throwable {

logger.info("Logging Aspect: Calling the intercepted method");

Object returnedValue = joinPoint.proceed(); ❶

logger.info("Logging Aspect: Method executed and returned " + returnedValue);

return returnedValue;

}}

❶ The proceed() method here delegates further in the aspect execution chain. It can either call the next aspect or the intercepted method

The 2nd aspect we define is named SecurityAspect. It prints a message in the console, so we observe when it is executed.

Listing 6.13 The implementation of the SecurityAspect class

@Aspect

public class SecurityAspect {

private Logger logger = Logger.getLogger(SecurityAspect.class.getName());

@Around(value = "@annotation(ToLog)")

public Object secure(ProceedingJoinPoint joinPoint) throws Throwable {

logger.info("Security Aspect: Calling the intercepted method");

Object returnedValue = joinPoint.proceed(); ❶

logger.info("Security Aspect: Method executed and returned " + returnedValue);

return returnedValue;

}}

❶ The proceed() method here delegates further in the aspect execution chain. It can call either the next aspect or the intercepted method.

Listing 6.14 The implementation of the CommentService class

@Service

public class CommentService {

private Logger logger = Logger.getLogger(CommentService.class.getName());

@ToLog

public String publishComment(Comment comment) {

logger.info("Publishing comment:" + comment.getText());

return "SUCCESS";

}}

Both aspects need to be beans in the Spring context. For this example, I chose the @Bean approach to add the beans in the context. My configuration class is presented next.

Listing 6.15 Declaring the aspect beans in the Configuration class

@Configuration

@ComponentScan(basePackages = "services")

@EnableAspectJAutoProxy

public class ProjectConfig {

@Bean ❶

public LoggingAspect loggingAspect() {

return new LoggingAspect();

}

@Bean ❶

public SecurityAspect securityAspect() {

return new SecurityAspect();

}}

❶ Both aspects need to be added as beans in the Spring context.

The main() method calls the publishComment() method of the CommentService bean. In my case, the output after the execution looks like the one in the next code snippet:

Sep 29, 2020 6:04:22 PM aspects.LoggingAspect log ❶

INFO: Logging Aspect: Calling the intercepted method ❶

Sep 29, 2020 6:04:22 PM aspects.SecurityAspect secure ❷

INFO: Security Aspect: Calling the intercepted method ❷

Sep 29, 2020 6:04:22 PM services.CommentService publishComment ❸

INFO: Publishing comment:Demo comment ❸

Sep 29, 2020 6:04:22 PM aspects.SecurityAspect secure ❹

INFO: Security Aspect: Method executed and returned SUCCESS ❹

Sep 29, 2020 6:04:22 PM aspects.LoggingAspect log ❺

INFO: Logging Aspect: Method executed and returned SUCCESS ❺

❶ The LoggingAspect is called first and delegates to the SecurityAspect.

❷ The SecurityAspect is called second and delegates to the intercepted method.

❸ The intercepted method executes.

❹ The intercepted method returns to the SecurityAspect.

❺ The SecurityAspect returns to the LoggingAspect.

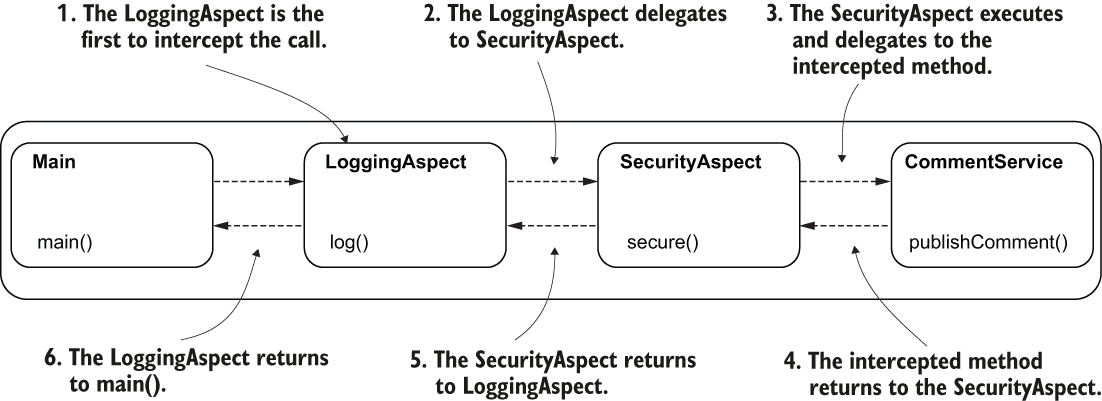


Figure 6.15 The execution flow. The LoggingAspect was first to intercept the method call. The LoggingAspect delegates further in the execution chain to the SecurityAspect, which further delegates the call to the intercepted method. The intercepted method returns to the SecurityAspect, which returns further to the LoggingAspect.

**PROJECT 36 – Logger and Security – @Order of the aspects**

To reverse the order in which LoggingAspect and SecurityAspect execute, we use the **@Order** annotation. I used the @Order annotation to specify an execution position for SecurityAspect:

@Aspect

@Order(1) ❶

public class SecurityAspect {

// Omitted code

}

❶ Gives an execution order position to the aspect

For the LoggingAspect, I use @Order to place the aspect in a higher order position:

@Aspect

@Order(2) ❶

public class LoggingAspect {

// Omitted code

}

❶ Places the LoggingAspect as second to be executed

Rerun the application and observe that the order in which the aspects execute has changed. The logging should now look like the one in the next snippet:

Sep 29, 2020 6:38:20 PM aspects.SecurityAspect secure ❶

INFO: Security Aspect: Calling the intercepted method ❶

Sep 29, 2020 6:38:20 PM aspects.LoggingAspect log ❷

INFO: Logging Aspect: Calling the intercepted method ❷

Sep 29, 2020 6:38:20 PM services.CommentService publishComment ❸

INFO: Publishing comment:Demo comment ❸

Sep 29, 2020 6:38:20 PM aspects.LoggingAspect log ❹

INFO: Logging Aspect: Method executed and returned SUCCESS ❹

Sep 29, 2020 6:38:20 PM aspects.SecurityAspect secure ❺

INFO: Security Aspect: Method executed and returned SUCCESS ❺

❶ The SecurityAspect is first to intercept the method call and delegates further in the execution chain to LoggingAspect.

❷ The LoggingAspect executes and delegates further to the intercepted method.

❸ The intercepted method executes and returns to the LoggingAspect.

❹ The LoggingAspect executes and returns to the SecurityAspect.

❺ The SecurityAspect returns to the main() method, which made the initial call.

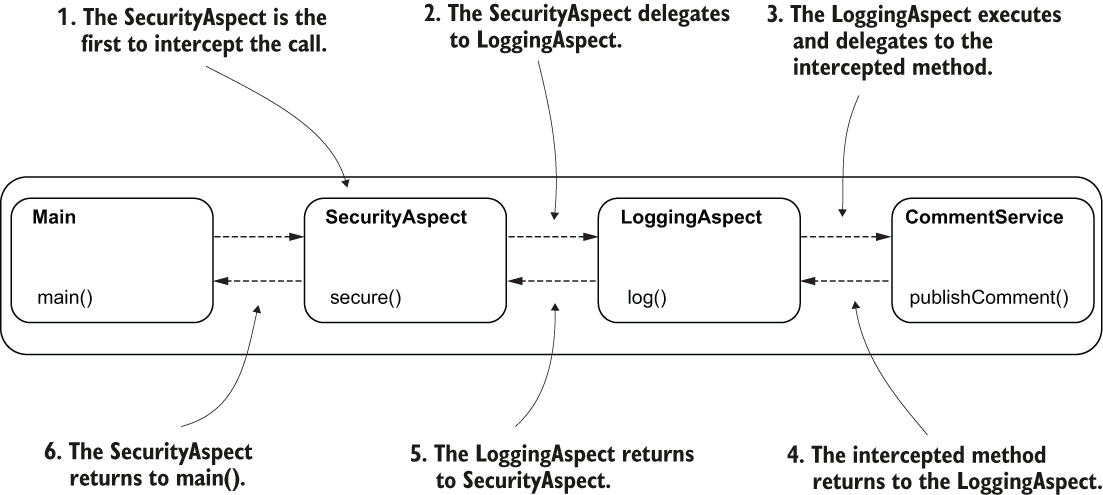


Figure 6.16 The execution flow after changing the order of the aspects. The SecurityAspect was first to intercept the method call and delegates further to the LoggingAspect, which further delegates to the intercepted method. The intercepted method returns to the LoggingAspect, which returns to the SecurityAspect.

**Summary**

* An aspect is an object that intercepts a method call and can execute logic before, after, and even instead of executing the intercepted method. This helps you decouple part of the code from the business implementation and makes your app easier to maintain.
* Using an aspect, you can write logic that executes with a method execution while being completely decoupled from that method. This way, someone who reads the code only sees what’s relevant regarding the business implementation.
* However, aspects can be a dangerous tool. Overengineering your code with aspects will make your app less maintainable. You don’t need to use aspects everywhere. When using them, make sure they really help your implementation.
* Aspects support many essential Spring capabilities like transactions and securing methods.
* To define an aspect in Spring, you annotate the class implementing the aspect logic with the **@Aspect** annotation. But remember that Spring needs to manage an instance of this class, so you need to also add a bean of its type in the Spring context.
* To tell Spring which methods an aspect needs to intercept, you use AspectJ pointcut expressions. You write these expressions as values to advice annotations. Spring offers you 5 advice annotations: **@Around**, **@Before**, **@After**, **@AfterThrowing**, and **@AfterReturning**. In most cases we use @Around, which is also the most powerful.
* Multiple aspects can intercept the same method call. In this case, it’s recommended that you define an order for the aspects to execute using the **@Order** annotation.