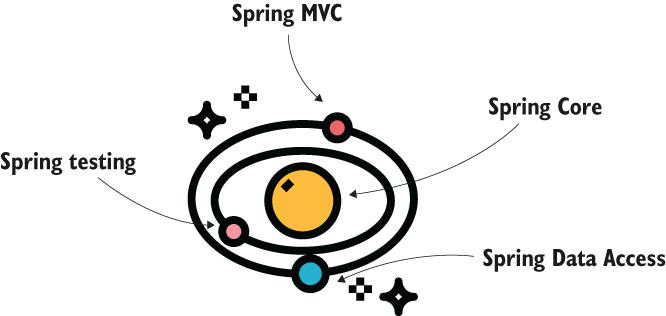
**Can you explain Spring framework?**

We refer to Spring as a framework, but it is much more complex. Spring is an ecosystem of frameworks. Usually, when developers refer to the Spring framework, they refer to a part of the software capabilities that include the following as displayed in figure below.

You can imagine the Spring framework as a solar system with the Spring Core in the center. The software capabilities are planets around Spring Core kept close to it by its gravitational field.



**What is Spring Core ?**

Spring Core is the part of the Spring framework that provides the foundational mechanisms to integrate into apps. Spring works based on the principle inversion of control (IoC). When using this principle, instead of allowing the app to control the execution, we give control to some other piece of software—in our case, the Spring framework.

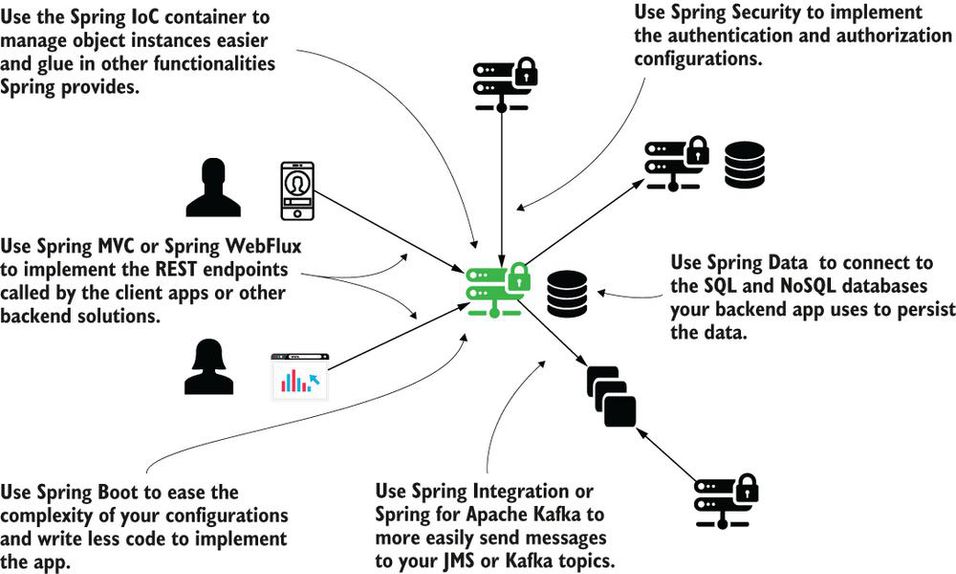
In this context the term “controls” refers to actions like “creating an instance” or “calling a method.” A framework can create objects of the classes you define in your app. Based on the configurations that you write, Spring intercepts the method to augment it with various features. For example, Spring can intercept a specific method to log any error that might appear during the method's execution.

The IoC container glues Spring components and components of your application to the framework together. Using the IoC container, to which you often refer as the Spring context, you make certain objects known to Spring, which enables the framework to use them in the way you configured.

**Using Spring in the Development of a Backend App?**

The possibilities of using Spring in a backend application are endless, from exposing

functionalities that other applications can call to managing the database access, and from securing the application to managing integration though third-party message brokers.



**What are scenarios when you should consider not using Spring 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.

**What is the reason behind using some other tools for replacing JDBC code?**

To remove ugly JDBC code.

You might be young enough not to have encountered JDBC direct usage in apps, but trust me, it's a long and ugly code.

At that time, some frameworks using another methodology to work with the database were becoming more and more popular. I remember when I first encountered Hibernate. This is an ORM framework, which allows you to treat the tables and their relationships in a database as objects and relationships among objects. When used correctly, it enables you to write less code and more intuitive functionality. When misused, it may slow down your app, make the code less intuitive, and even introduce bugs.

The application we were developing needed a change. We knew we could improve that ugly JDBC code. In my mind, we could at least minimize the number of lines. This change would have brought great benefits to maintainability. Together with other developers, we suggested using a tool provided by Spring called JdbcTemplate (you'll learn this tool in [chapter 12](https://cdn2.percipio.com/1692098451.93c1996282fd68bd9258f915581c5505bbf27fa9/eod/books/158139/OEBPS/chapter-12-67.xhtml#ch12)). But others strongly pushed the decision to use Hibernate. It was quite popular, so why not to use it? (Actually it still is one of the most popular frameworks of its kind, and you'll learn about integrating it with Spring in [chapter 13](https://cdn2.percipio.com/1692098451.93c1996282fd68bd9258f915581c5505bbf27fa9/eod/books/158139/OEBPS/chapter-13-72.xhtml#ch13).) I could see changing that code to a completely new methodology would be a challenge. Moreover, I could see no benefits. The change also implied a greater risk of introducing bugs.

Fortunately, the change started with a proof of concept. After a couple of months, lots of effort, and stress, the team decided to quit.

After analyzing our options, we finished the implementation using JdbcTemplate. We managed to write cleaner code by eliminating a large number of lines of code, and we didn't need to introduce any new framework for this change.

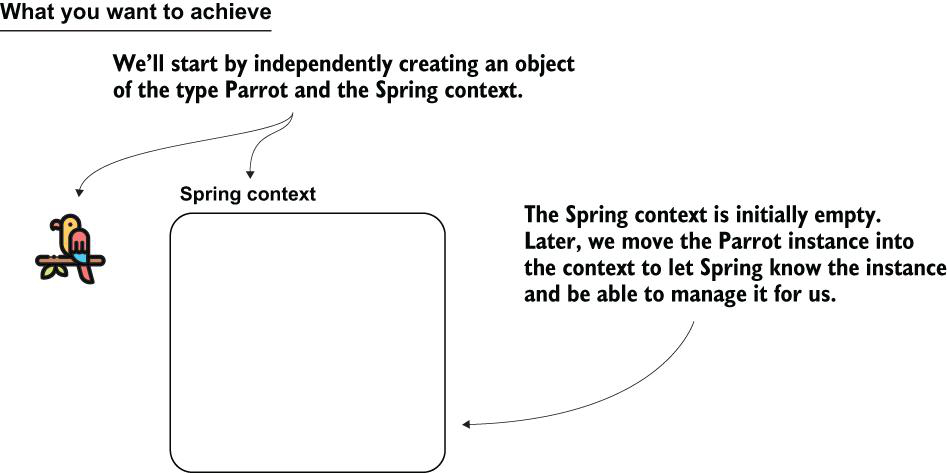
**What are different ways of adding beans into Spring context?**

You can add beans in the context in the following:

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

**Can you give typical example of managing bean by Spring context?**

To start, we create an object instance and the empty Spring context.



#### Creating the instance of the Spring context

public class Main {

public static void main(String[] args) {

var context =

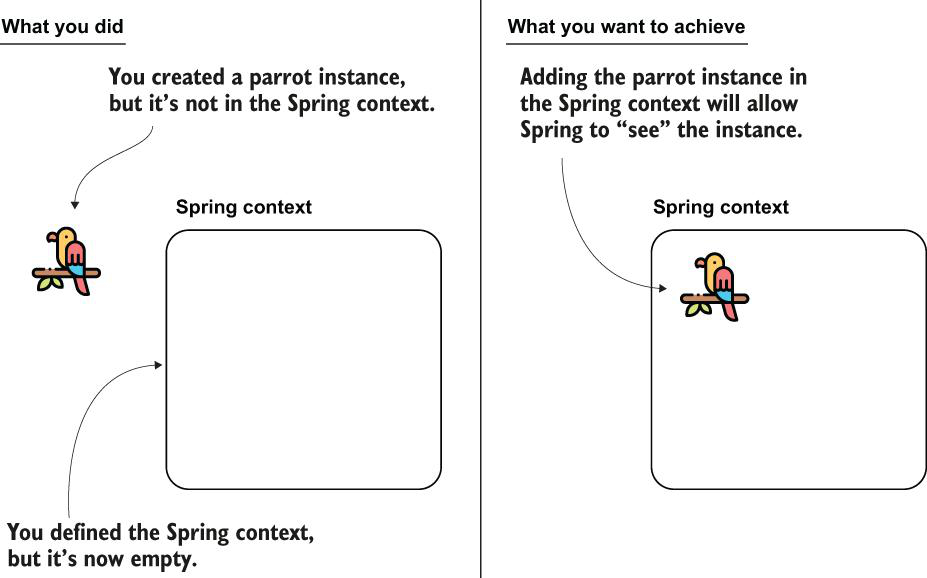
new AnnotationConfigApplicationContext();

Parrot p = new Parrot();

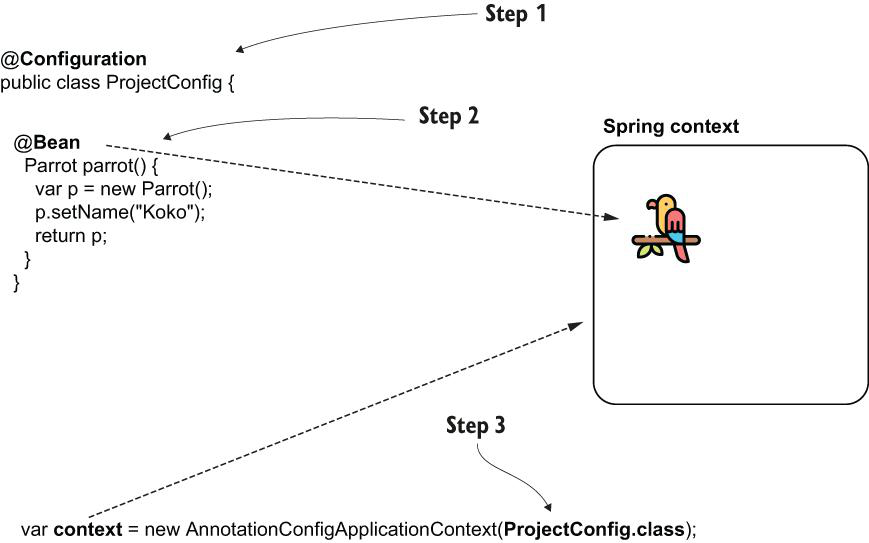
}

}

You created the Spring context instance and a Parrot instance. Now, you want to add the Parrot instance inside the Spring context to make Spring aware of this instance.



Steps for adding the bean to the context using the @Bean annotation. By adding the instance to the Spring context, you make the framework aware of the object, enabling it to manage the instance.



You can add any object to the Spring context to make Spring aware of it.

@Configuration

public class ProjectConfig {

@Bean

Parrot parrot() {

var p = new Parrot();

p.setName("Koko");

return p;

}

@Bean ❶

String hello() {

return "Hello";

}

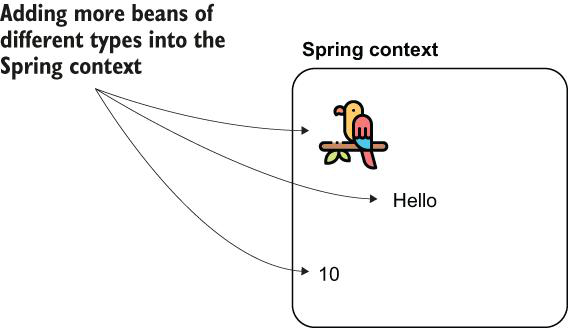
@Bean ❷

Integer ten() {

return 10;

}

}



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.

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;

}

}



#### 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());

}

}

**What are different ways of naming a bean using Bean annotation attributes?**

 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”)

**How do you define a primary bean?**

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");

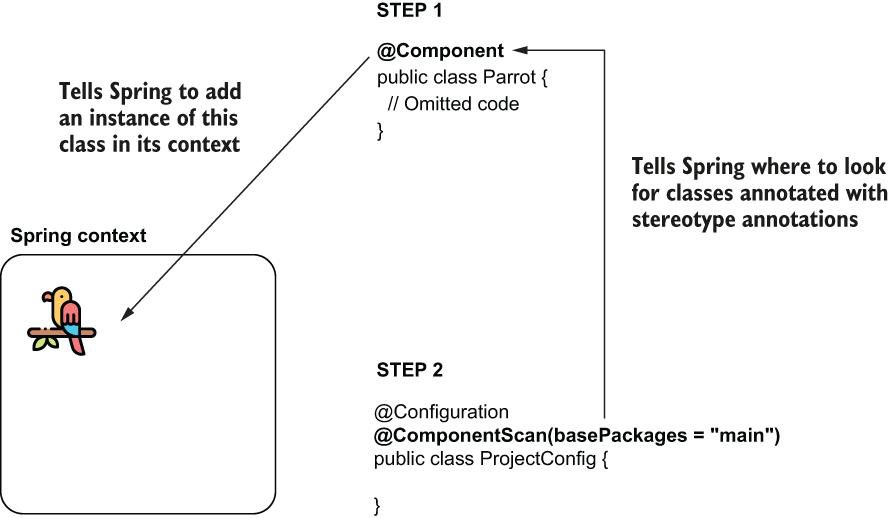
return p;

}

If you refer to a Parrot without specifying the name, Spring will now select Miki by default. Of course, you can only define one bean of a type as primary.

**How do you configure beans using stereotype annotations?**

When using stereotype annotations, consider two steps. First, use the stereotype annotation (@Component) to annotate the class for which you want Spring to add a bean to its context. Second, use the @ComponentScan annotation to tell Spring where to look for classes annotated with stereotype annotations



#### 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

**How do you enable dependency injection annotations in Spring?**

The <context:annotation-config> annotation is mainly used to activate the dependency injection annotations. @Autowired, @Qualifier, @PostConstruct, @PreDestroy, and @Resource are some of the ones that <context:annotation-config> can resolve.

**What are differences between in approaches to create beans in Spring context, for example using @Bean annotation and using Stereotype annotations?**

What you'll observe is that in real-world scenarios you'll use stereotype annotations as much as possible (because this approach implies writing less code), and 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).

| **Table 2.1: Advantages and disadvantages: A comparison of the two 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** |
| 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.  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.  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.  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 second option to stereotype annotations in our projects. | You only have control over the instance after the framework creates it.  This way, you can only add one instance of the class to the context.  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.  Using stereotype annotations to add beans to the Spring context doesn't add boilerplate code to your app. You'll prefer this approach in g |

**What is use of @PostConstruct when bean creation is done using stereotype annotatinos?**

As we've discussed in this section, using stereotype annotations you instruct Spring to create a bean and add it to its context. But, unlike using the @Bean annotation, you don't have full control over the instance 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.

Spring borrows the @PostConstruct annotation from Java EE. We can also use this annotation with Spring beans to specify a set of instructions Spring executes after the bean creation. 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. Before Java 11, the Java EE dependencies were part of the JDK. With Java 11, the JDK was cleaned of the APIs not related to SE, including the Java EE dependencies.

Now you can define a method in the Parrot class, as presented in the next code snippet:

@Component

public class Parrot {

private String name;

@PostConstruct

public void init() {

this.name = "Kiki";

}

// Omitted code

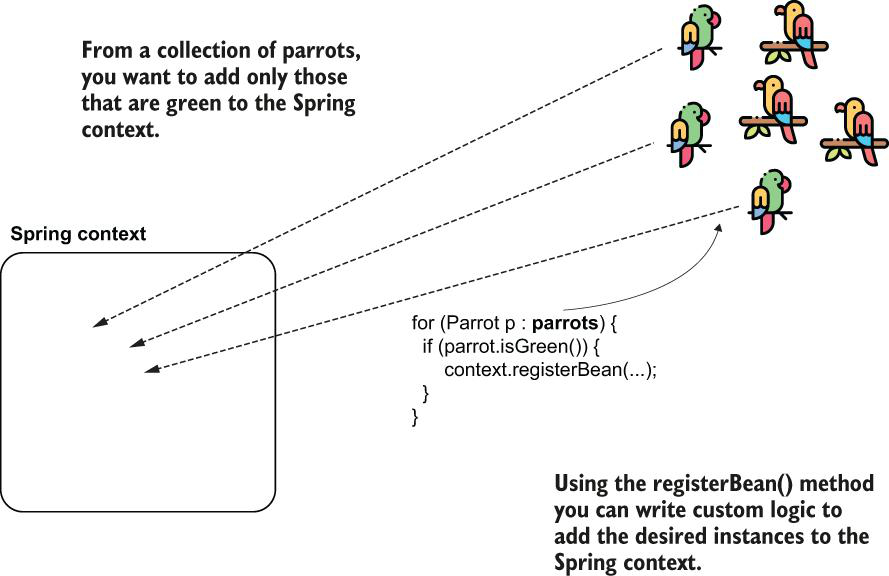
}

**What is use of @PreDestroy annotation?**

With this annotation, you define a method that Spring calls immediately before closing and clearing the context. The @PreDestroy annotation is also described in JSR-250 and borrowed by Spring. 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.

**Programmatically Adding Beans to the Spring Context?**

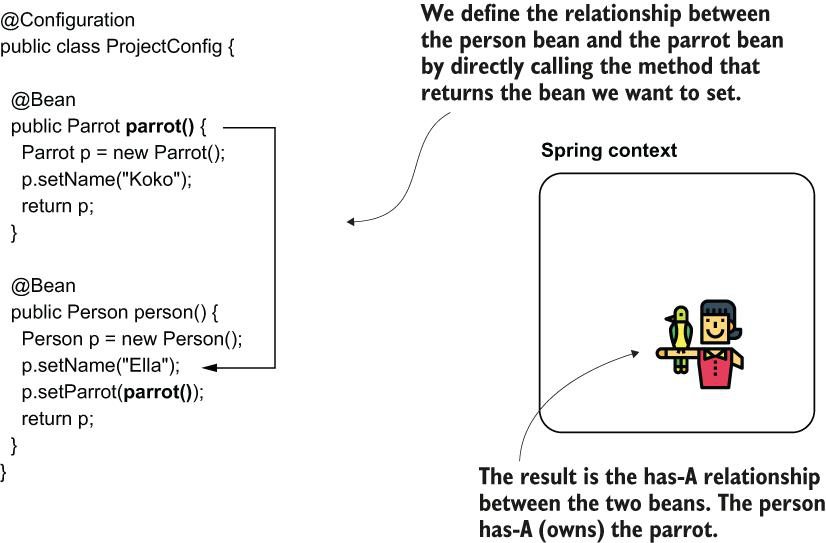
Using the registerBean() method to add specific object instances to the Spring context



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.

**Wiring the Beans Using a Direct Method Call between the @Bean Methods?**

We establish the relationship between the beans using direct wiring. 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



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?

A diagram of a parrot

Description automatically generated

When two methods annotated with @Bean call each other, Spring knows you want to create a link between the beans. If the bean already exists in the context (3A), Spring returns the existing bean without forwarding the call to the @Bean method. If the bean doesn't exist (3B), Spring creates the bean and returns its reference

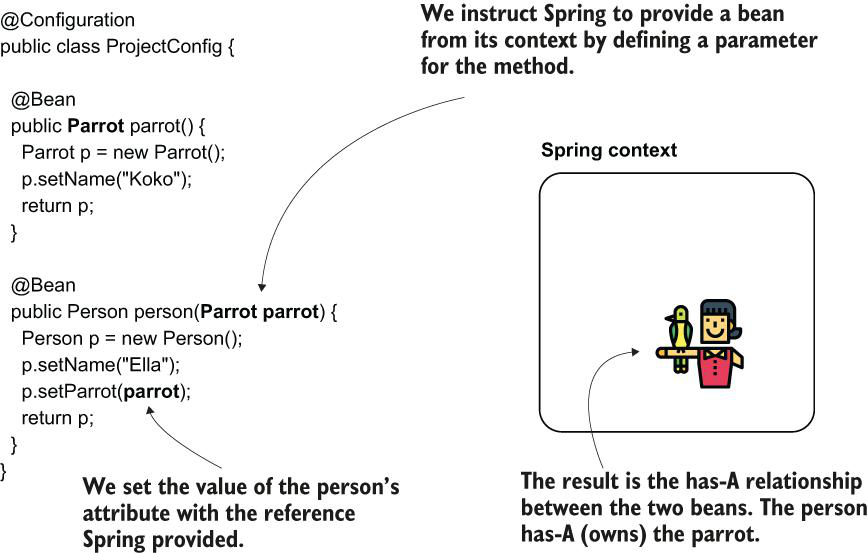
A diagram of a parrot method

Description automatically generated

**Wiring the Beans Using the @Bean Annotated Method's Parameters?**

Instead of directly calling the method that defines the bean we wish to refer to, 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 (figure 3.7). This approach is a bit more flexible than the one we discussed in section 3.1.1. 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 (discussed in chapter 2). In my experience, however, I have observed it's not necessarily this flexibility that makes developers use this approach; it's mostly the taste of each developer that determines which approach they use when working with beans. I wouldn't say one is better than the other, but you'll encounter both approaches in real-world scenarios, so you need to understand and be able to use them.

By defining a parameter to the method, we instruct Spring to provide us a bean of the type of that parameter from its context. We can then use the provided bean (parrot) when creating the second one (person). This way we establish the has-A relationship between the two beans.



@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;

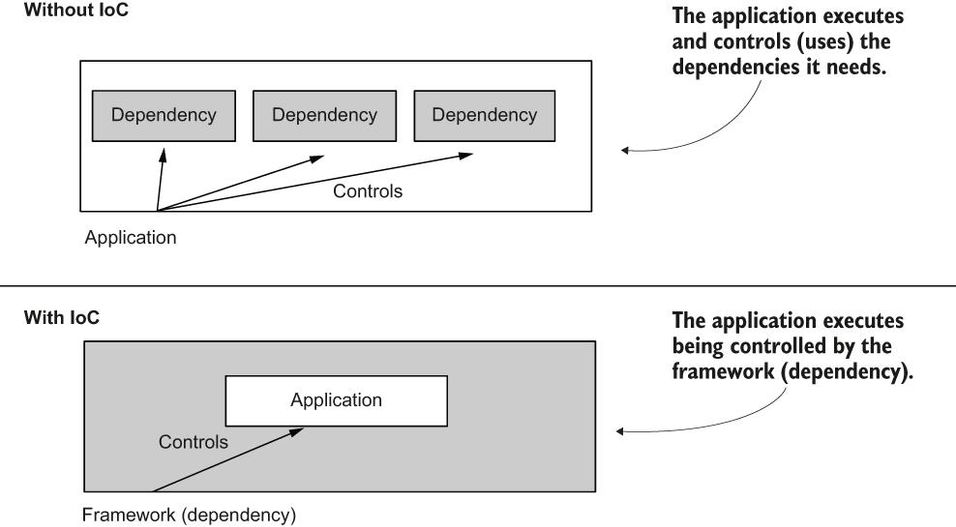
}

}

❶ Spring injects the parrot bean into this parameter.

In the previous paragraph, I used the word “inject.” I refer here to what we will from now on call dependency injection (DI). As its name suggests, 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

An application that's not using the IoC principle controls the execution and makes use of various dependencies. An application using the IoC principle allows a dependency to control its execution. The DI is such an example of control. The framework (a dependency) sets a value into a field of an object of the app.



**What are two approaches in Spring for establishing link between two beans?**

Wiring:

@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.

Auto-wiring:

@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.

**What are different ways to @Autowired annotation?**

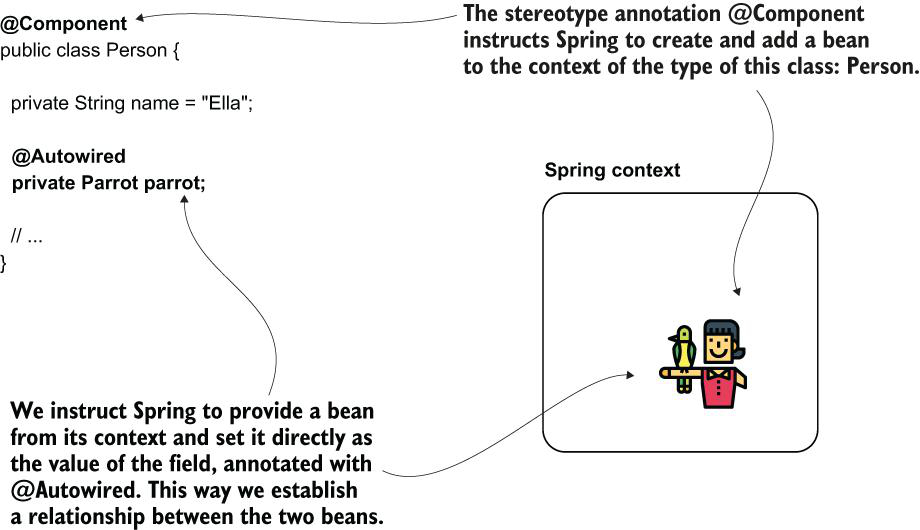
there are three ways we can use the @Autowired annotation:

* **Field Injection:** Injecting the value in the field of the class, which you usually find in examples and proofs of concept.
* **Constructor Injection:** Injecting the value through the constructor parameters of the class approach that you'll use most often in real-world scenarios.
* **Setter Injection:** Injecting the value through the setter, which you'll rarely use in production-ready code.

**Field Injection:**

In this section, we start by discussing the simplest of the three possibilities for using @Autowired, which is also the one developers often use in examples: using the annotation over the field ([figure 3.9](https://cdn2.percipio.com/1692186932.8a8d4f2729bd469477b250849ea583031d5f2d06/eod/books/158139/OEBPS/section-32-23.xhtml#ch03fig09)). As you'll learn, even if this approach is very straightforward, it has its sins, which is why we avoid using it when writing production code. However, you'll see it often used in examples, proofs of concept, and in writing tests, as we'll discuss in [chapter 15](https://cdn2.percipio.com/1692186932.8a8d4f2729bd469477b250849ea583031d5f2d06/eod/books/158139/OEBPS/chapter-15-82.xhtml#ch15), so you need to know how to use this approach.

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



**Note:** I've used stereotype annotations to add the beans in the Spring context for this example. I could have defined the beans using @Bean, but most often, in real-world scenarios, you'll encounter @Autowired used together with stereotype annotations, so let's focus on the approach that's most useful for you.

Why is this approach not desired in production code? It's not totally wrong to use it, but you want to make sure you make your app maintainable and testable in production code. By injecting the value directly in the field:

* you don't have the option to make the field final (see next code snippet), 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

}

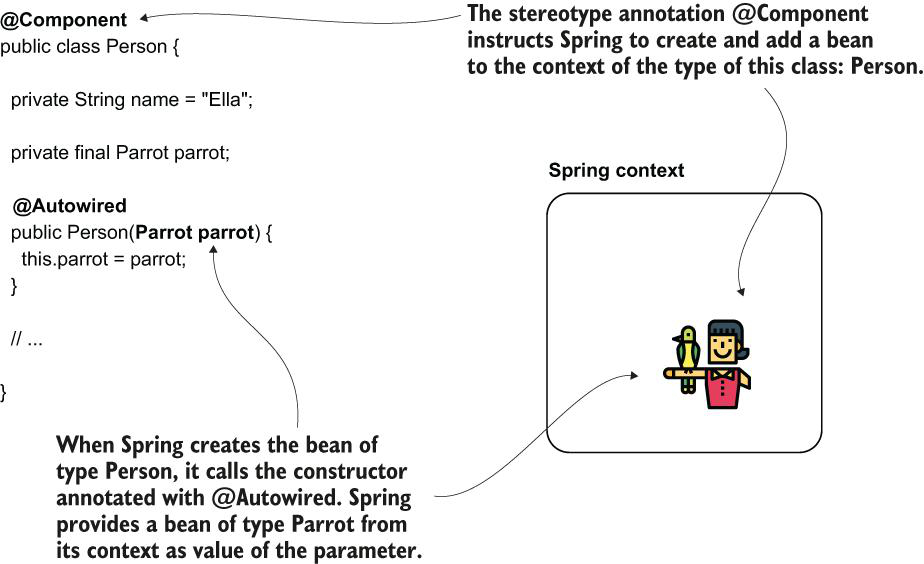
❶ This doesn't compile. You cannot define a final field without an initial value.

* it's more difficult to manage the value yourself at initialization.

**Constructor Injection:**

The second 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 ([figure 3.10](https://cdn2.percipio.com/1692186932.8a8d4f2729bd469477b250849ea583031d5f2d06/eod/books/158139/OEBPS/section-32-23.xhtml#ch03fig10)). 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 (but more on this subject later).

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



#### 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.

#### **Note:**

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

**Setter Injection:**

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. Even so, I wanted to mention this possibility. You might encounter it at some point, and I don't want you to wonder about its existence then. Even if it's not something I recommend, I have seen this used in a couple of old apps.

@Component

public class Person {

private String name = "Ella";

private Parrot parrot;

// Omitted getters and setters

@Autowired

public void setParrot(Parrot parrot) {

this.parrot = parrot;

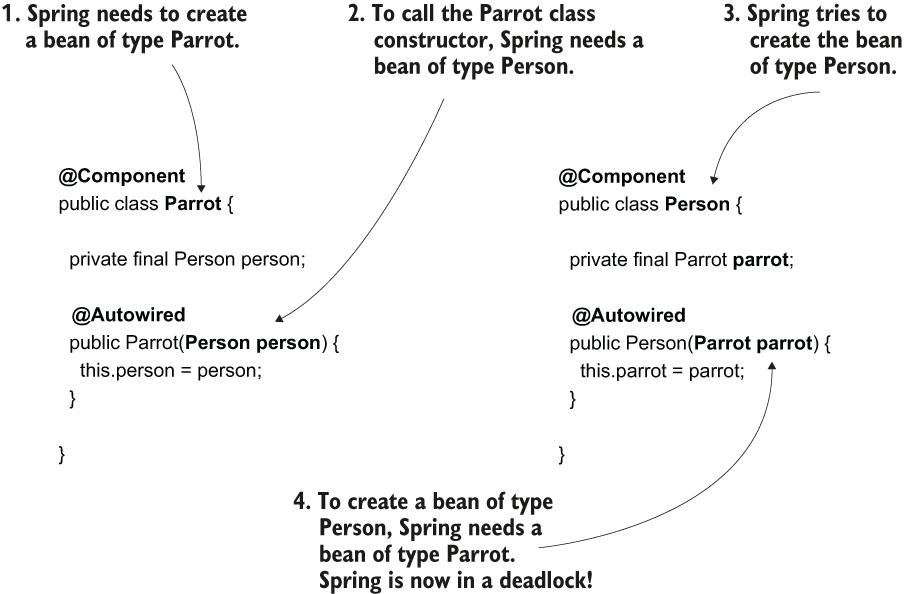
}

}

**What is circular dependency in Spring?**

A circular dependency ([figure 3.11](https://cdn2.percipio.com/1692188922.f7ac792d8f32782be82103f59d2fc42c95a4aefb/eod/books/158139/OEBPS/section-33-24.xhtml#ch03fig11)) 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.

Figure 3.11: A circular dependency. Spring needs to create a bean of type Parrot. But because Parrot has as a dependency a Person, Spring needs first to create a Person. However, to create a Person, Spring already needs to have built a Parrot. Spring is now in a deadlock. It cannot create a Parrot because it needs a Person, and it cannot create a Person because it needs a Parrot



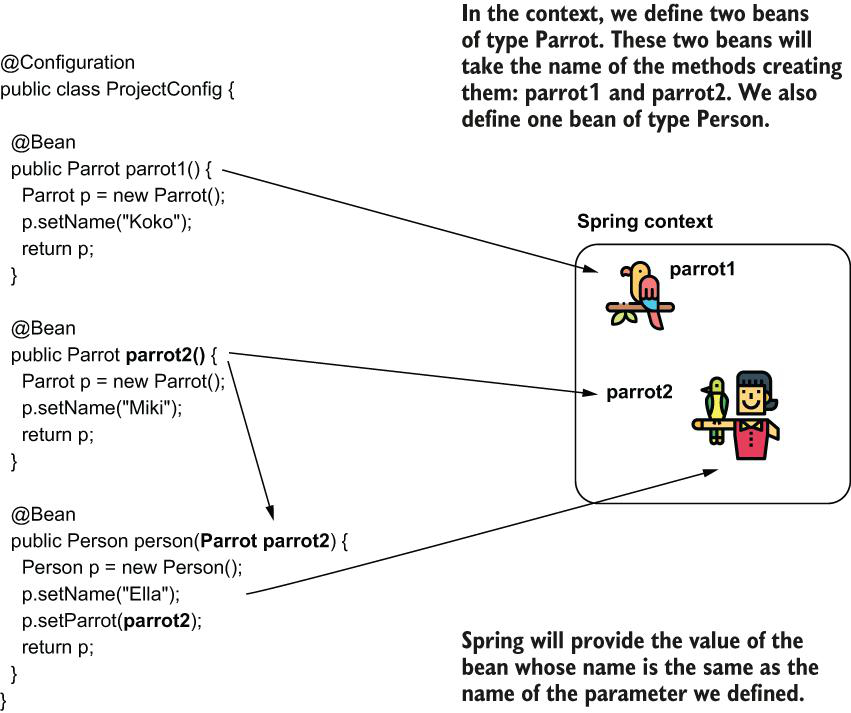
A circular dependency is easy to avoid. You just need to make sure you don't define objects whose creation depends on the other. Having dependencies from one object to another like this is a bad design of classes. In such a case, you need to rewrite your code.

I don't think I know any Spring developer who didn't at least once create a circular dependency in an app. You need to be aware of this scenario so that when you encounter it, you know its cause and you'll solve it fast.

**How do you choose from multiple Beans in the Spring Context during dependency injection?**

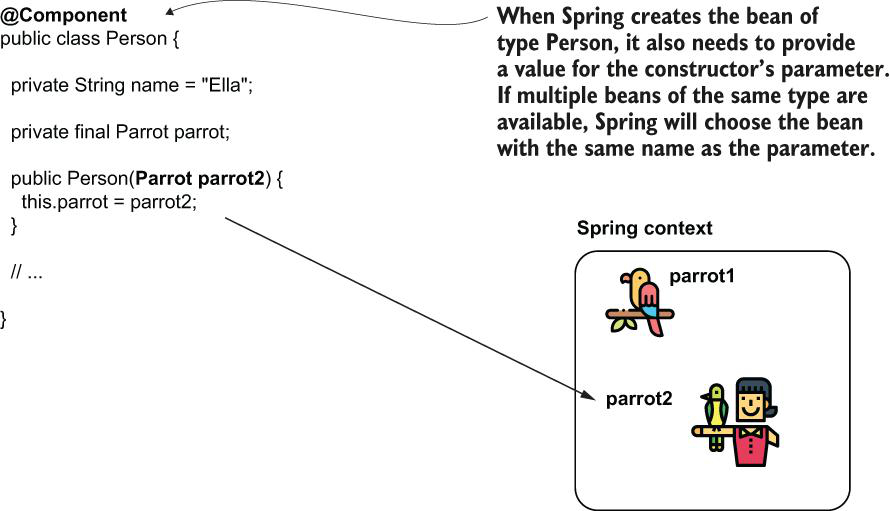
* When Spring has more than one bean of the same type in its context, it can't decide which of those beans need to be injected. You can tell Spring which is the instance it needs to inject by
  + using the @Primary annotation, which marks one of the beans as the default for dependency injection, or
  + naming the beans and injecting them by name using the @Qualifier annotation.

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.



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.

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



As we discussed for the @Bean annotated method parameter, I recommend against relying on the name of the variable. Instead, 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. Look at the change I made to the Person class in the next code snippet. 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 (see the change in the project named “sq-ch3-ex10”):

@Component

public class Person {

private String name = "Ella";

private final Parrot parrot;

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

this.parrot = parrot;

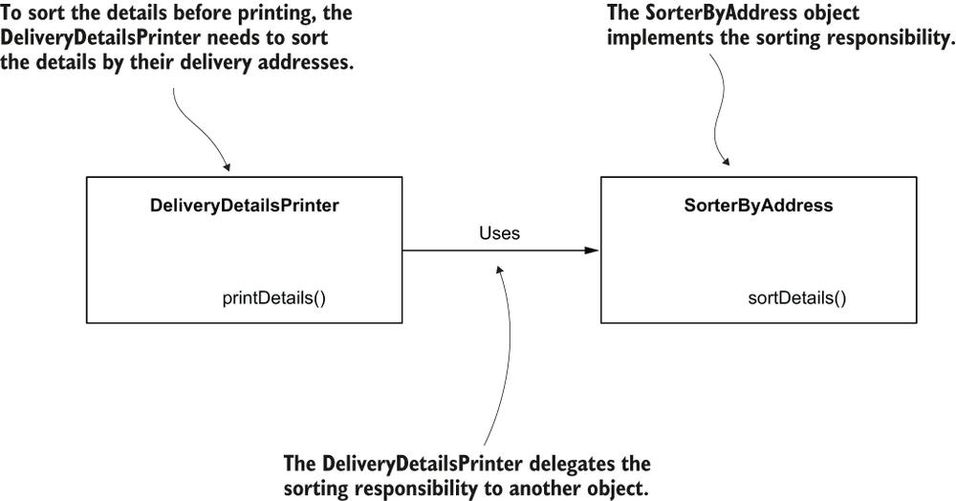
}

// Omitted getters and setters

}

**How can you utilize interfaces for decoupling implementations?**

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

Larger View

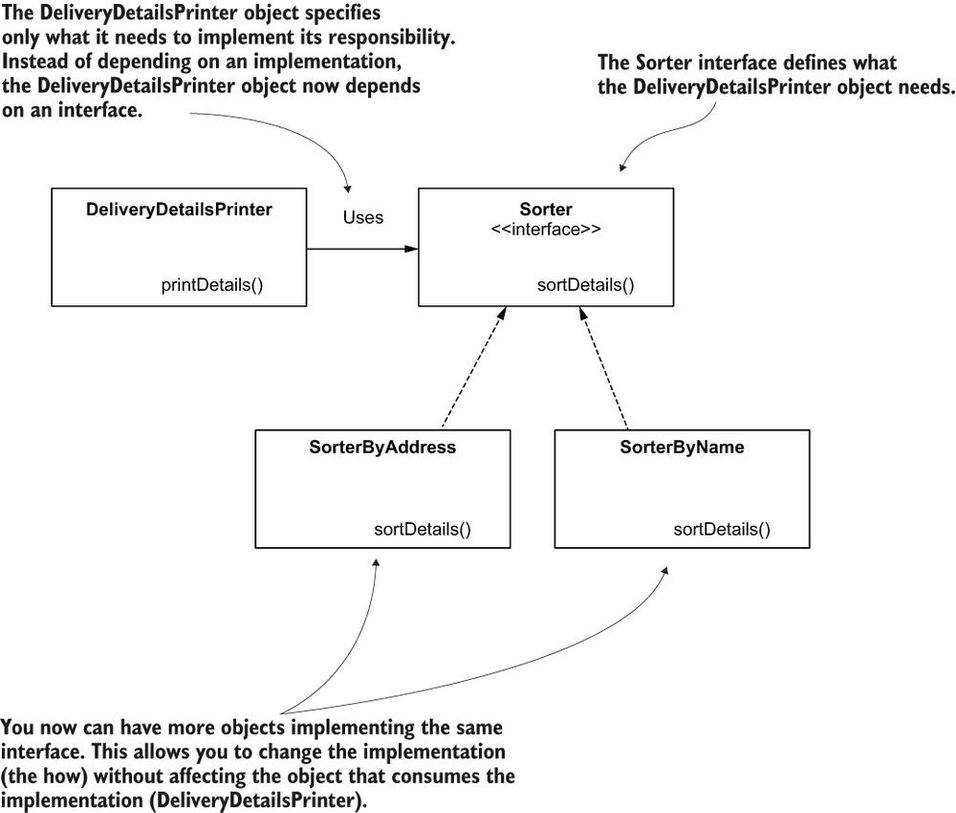
Because the two objects are strongly coupled, if you want to change the sorting responsibility, you also need to change the object using this responsibility. A better design would allow you to change the sorting responsibility without changing the object that uses the responsibility

A diagram of a computer program

Description automatically generated with medium confidence

How can we improve this design? 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. As discussed earlier, an object only needs to specify what it needs and stay completely unaware of how the what is implemented.

Using an interface to decouple the responsibilities. 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

Larger View

In the next code snippet, you find the Sorter interface definition:

public interface Sorter {

void sortDetails();

}

The DeliveryDetailsPrinter object depends on the Sorter interface. You can change the implementation of the Sorter interface and avoid making more changes to the object using this responsibility (DeliveryDetailsPrinter)

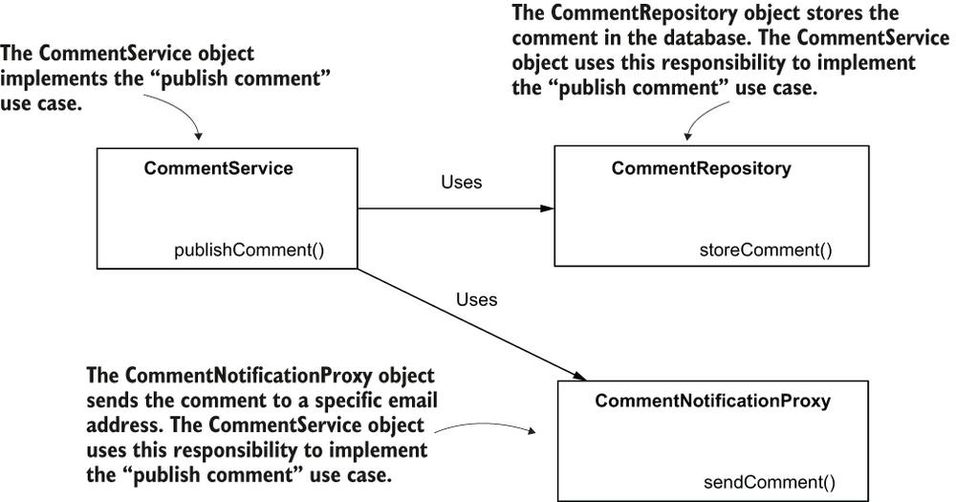
A black and white image of a pointy object

Description automatically generated

**Can you implement code to interface concept without using a framework?**

Finally, in a real-world app, 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

Larger View

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

A diagram of a service

Description automatically generated

public class Comment {

private String author;

private String text;

// Omitted getters and setters

}

public interface CommentRepository {

void storeComment(Comment comment);

}

public class DBCommentRepository implements CommentRepository {

@Override

public void storeComment(Comment comment) {

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

}

}

public interface CommentNotificationProxy {

void sendComment(Comment comment);

}

public class EmailCommentNotificationProxy

implements CommentNotificationProxy {

@Override

public void sendComment(Comment comment) {

System.out.println("Sending notification for comment: "

+ comment.getText());

}

}

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);

}

}

public class Main {

public static void main(String[] args) {

var commentRepository =

new DBCommentRepository(); ❶

var commentNotificationProxy = ❶

new EmailCommentNotificationProxy(); ❶

var commentService =

new CommentService( ❷

commentRepository, commentNotificationProxy);

var comment = new Comment(); ❸

comment.setAuthor("Laurentiu"); ❸

comment.setText("Demo comment"); ❹

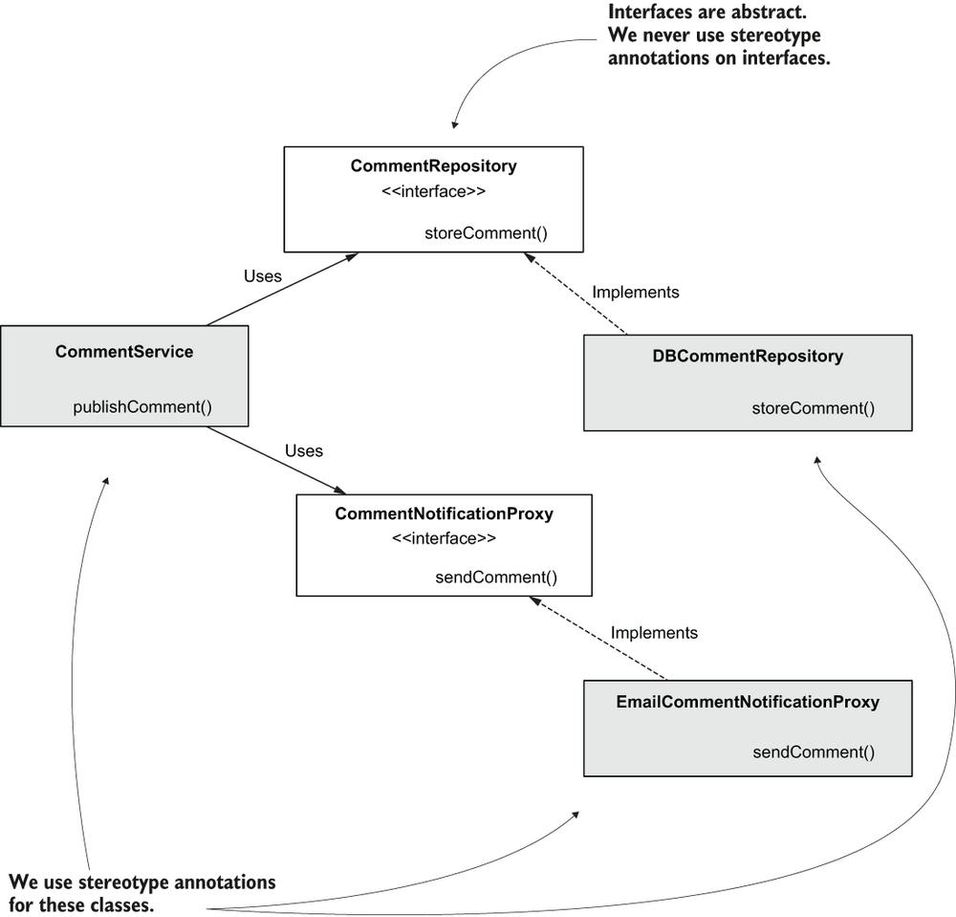
commentService.publishComment(comment); ❹

}

}

**How do you use dependency injection with abstraction?**

 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



@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.

@Component

public class EmailCommentNotificationProxy

implements CommentNotificationProxy {

@Override

public void sendComment(Comment comment) {

System.out.println(

"Sending notification for comment: " +

comment.getText());

}

}

@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.

@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 anno-tations. Observe that the model package is not specified because it doesn't contain classes annotated with stereotype annotations.

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);

}

}

It's a small example, and it might not look like Spring improves a lot the experience, but look again. 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 three classes, letting Spring manage the objects and dependencies among them really makes a difference. It eliminates code that can be implied (which developers also name 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 that we'll discuss in the next chapters.

**When to use attribute basePackages and basePackageClasses?**

In this example, 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. In a production application, you might find both ap-proaches, and, in my experience, one is not better than the other.

**What are different approaches of autowiring a bean?**

1. Using @Autowired annotation:

@Autowired annotation, through which you can make field, constructor, or setter injection

1. Using auto-wiring within the configuration class using the parameters of the methods annotated with @Bean (which Spring uses to create beans in the context).

**Can you give setter injection example?**

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

@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 two dependencies.

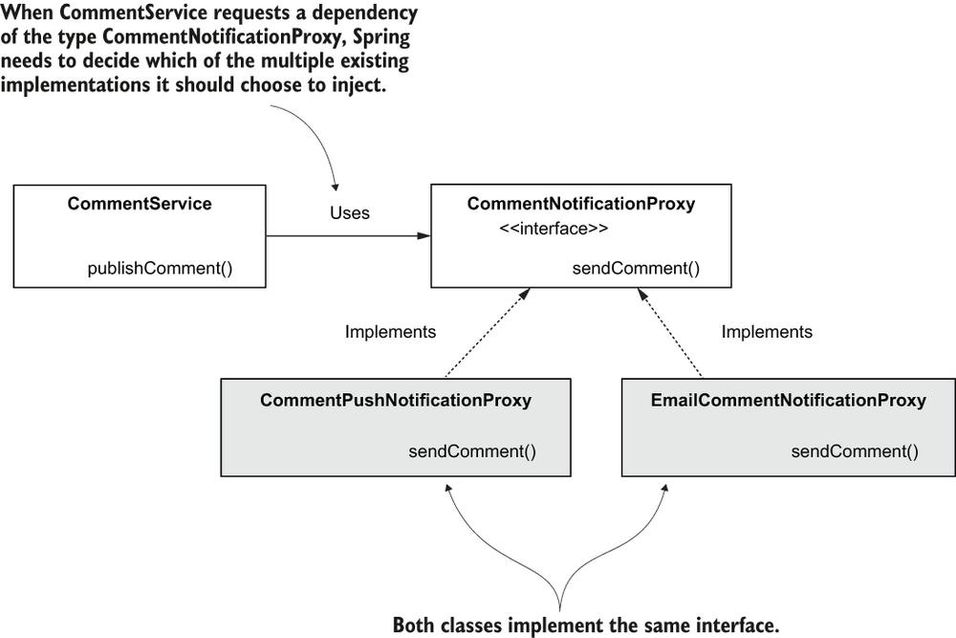
❸ We use parameters of the @Bean method (which are now defined with the interface type) to instruct Spring to provide references for beans from its context, compatible with the type of the parameters.

### Choosing What to Auto-Wire from Multiple Implementations of an Abstraction?

Suppose we have two beans created with two different classes that implement the CommentNotificationProxy interface ([figure 4.9](https://cdn2.percipio.com/1692790663.d593e67911b811d3100fca2347191c522f2cdd7b/eod/books/158139/OEBPS/section-42-29.xhtml#ch04fig09)). Fortunately for us, Spring uses a mechanism for deciding which bean to choose that we discussed in [chapter 3](https://cdn2.percipio.com/1692790663.d593e67911b811d3100fca2347191c522f2cdd7b/eod/books/158139/OEBPS/chapter-3-21.xhtml#ch03). In [chapter 3](https://cdn2.percipio.com/1692790663.d593e67911b811d3100fca2347191c522f2cdd7b/eod/books/158139/OEBPS/chapter-3-21.xhtml#ch03), you learned that 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. You also learned the following approaches:

* Using the @Primary annotation to mark one of the beans for implementation as the default
* Using the @Qualifier annotation to name a bean and then refer to it by its name for DI

Sometimes, in real-world scenarios, we have multiple implementations of the same interface. When using dependency injection on the interface, you need to instruct Spring which is the implementation it should inject



**How do inject a particular bean if multiple beans of same type are available in the Spring context?**

The 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. In the next listing, you find out 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.

**What will happen if we interchange @service and @repository annotation in the spring MVC?**

The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO). Among the uses of this marker is the automatic translation of exceptions, as described in Exception Translation.

Spring provides further stereotype annotations: @Component, @Service, and @Controller. @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases (in the persistence, service, and presentation layers, respectively). Therefore, you can annotate your component classes with @Component, but, by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. @Repository, @Service, and @Controller can also carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated earlier, @Repository is already supported as a marker for automatic exception translation in your persistence layer.

According to [documentaion](https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-stereotype-annotations) @Repository,@Service,@Controller are all synonyms. They all are just specializations of @Component annotation. So, generally, they can be used one instead of other. But ... you should not do this.

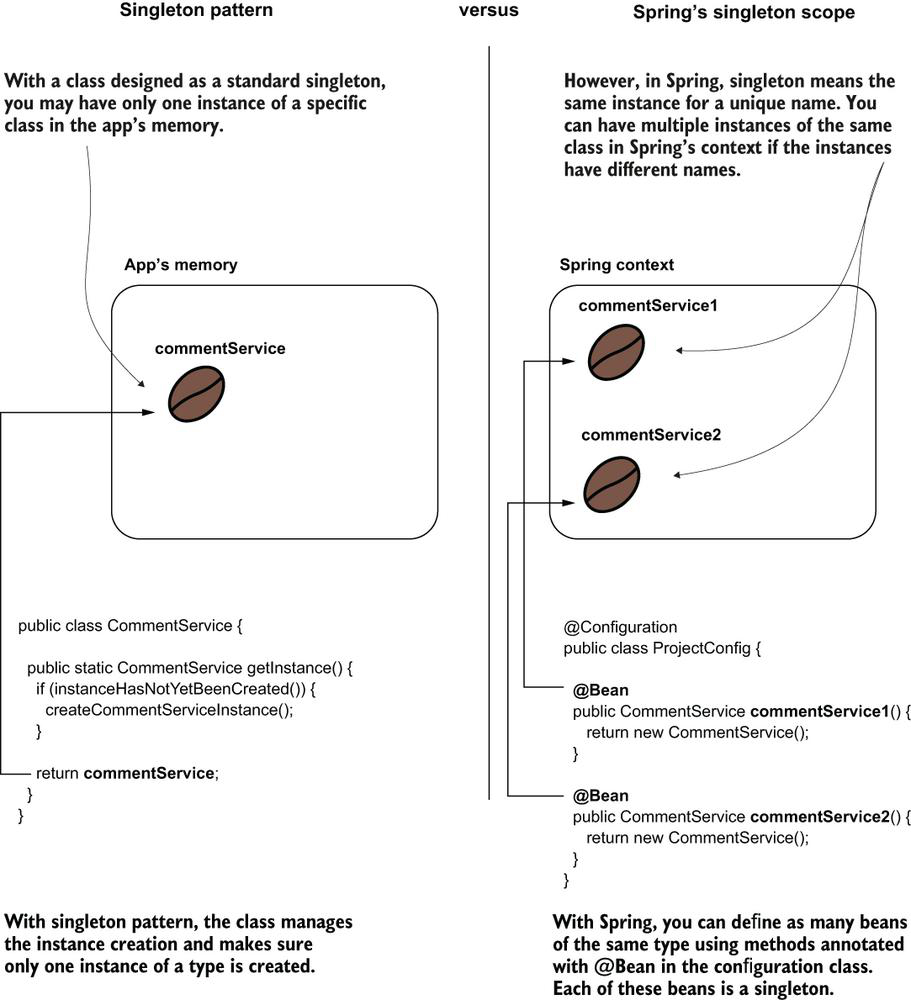
First reason: any of these annotations make clear the role of your component in the application. Shows - is this component belongs to the controller, service, or data layer.

Second reason: some of these annotations processed differently by different Spring modules. For example, Spring Data JPA will process @Repository and will try to replace with implementation any interface marked by this annotation. Spring also will apply automatic exception translation to such classes. Another example: Spring Web MVC processes @Controller, and uses classes marked with it in URL mappings.

Actually, in future versions, some modules of Spring could process @Service in a particular way. Not as simple @Component.

**What is difference between singleton design pattern and singleton bean scope in Spring?**

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

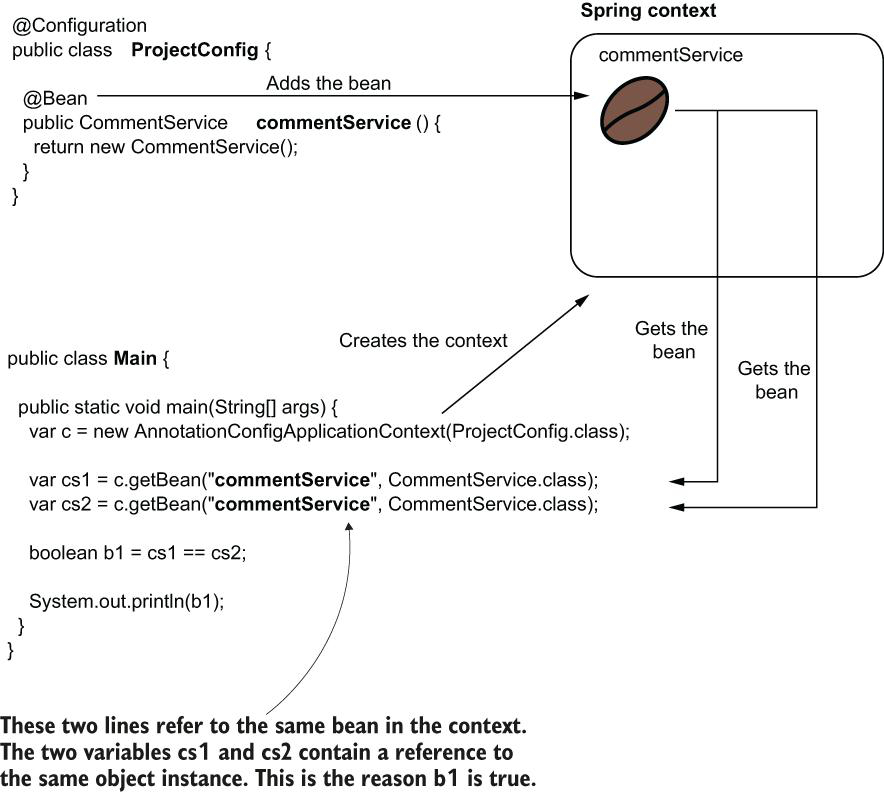


**What would be name of bean annotated with @Bean annotation?**

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.

**Can you explain singleton bean and how do you declare Singleton-Scoped Beans with @Bean?**

A singleton bean. 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.



Also, note that I've explicitly used the bean name when getting the bean from the Spring context in this demonstration. You learned in [chapter 2](https://cdn2.percipio.com/1692892947.9997f9138efc1c8eeb347a59f836fd18ea94cf3c/eod/books/158139/OEBPS/chapter-2-17.xhtml#ch02) 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. As we discussed in [chapter 2](https://cdn2.percipio.com/1692892947.9997f9138efc1c8eeb347a59f836fd18ea94cf3c/eod/books/158139/OEBPS/chapter-2-17.xhtml#ch02), I could have just referred to the type, and in both cases where we get the bean from the context we would get the reference to the same (and only) instance of CommentService in the context.

public class CommentService {

}

@Configuration

public class ProjectConfig {

@Bean ❶

public CommentService commentService() {

return new CommentService();

}

}

❶ Adds the CommentService bean to the Spring context

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);

}

}

**Declaring Singleton Beans Using Stereotype Annotations?**

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



@Repository

public class CommentRepository {

}

@Service

public class CommentService {

@Autowired

private CommentRepository commentRepository;

public CommentRepository getCommentRepository() {

return commentRepository;

}

}

@Service

public class UserService {

@Autowired

private CommentRepository commentRepository;

public CommentRepository getCommentRepository() {

return commentRepository;

}

}

Unlike the first example in this section, the configuration class remains empty in this project. We only need to tell Spring where to find the classes annotated with stereotype annotations. As discussed in [chapter 2](https://cdn2.percipio.com/1692892947.9997f9138efc1c8eeb347a59f836fd18ea94cf3c/eod/books/158139/OEBPS/chapter-2-17.xhtml#ch02), to tell Spring where to find classes annotated with stereotype annotations we use the @ComponentScan annotation. The definition of the configuration class is in the next code snippet:

@Configuration

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

public class ProjectConfig {

}

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 always prints “true.”

**What are points to be taken care of while creating a singleton bean?**

* 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.
* 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.
* If a bean needs to be mutable, an option could be to use the prototype scope, which we discusss.

**Which one to prefer lazy initialization or eager initialization for creating singleton beans?**

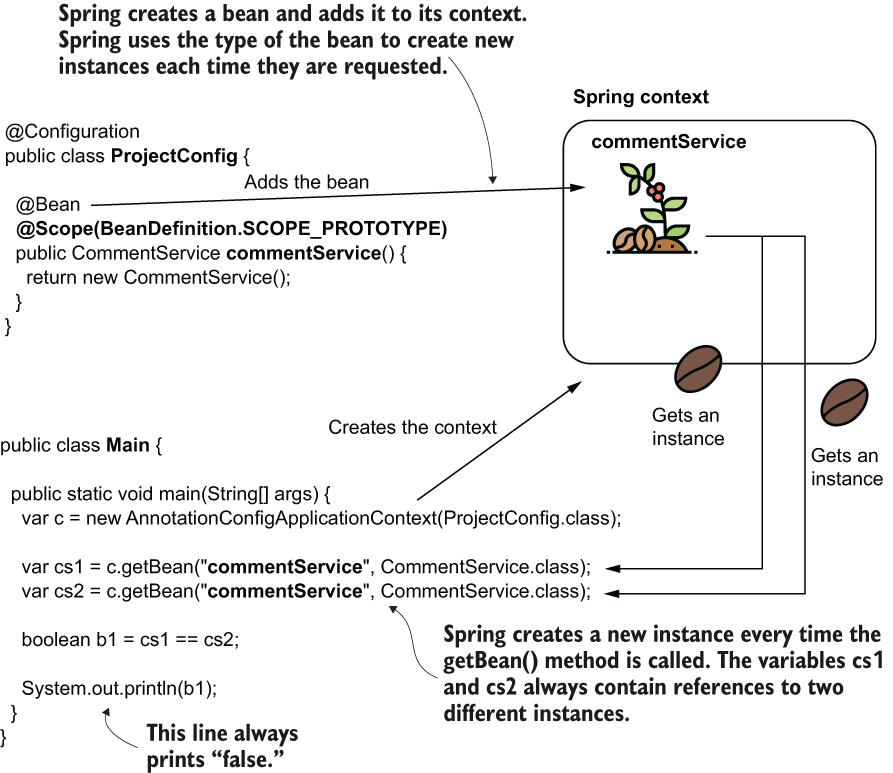
In a lazy instantiation, the framework has to first check if the instance exists and eventually create it if it doesn't, so from the performance point of view, it's better to have the instances in the context already (eager) because it spares some checks the framework needs to do when one bean delegates 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. But in the real world, not everything is possible due to other factors like cost or time. If you cannot treat the real cause of the problem, you can sometimes treat at least some of the symptoms.

**Can you explain prototype bean?**

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

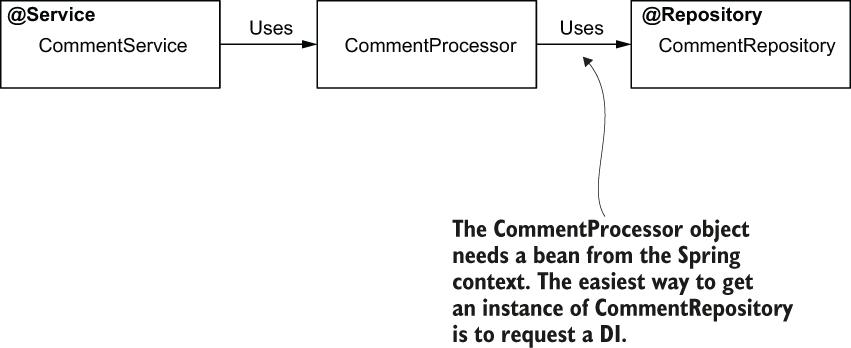
A diagram of coffee beans

Description automatically generated

**Why do you even need a bean?**

To be able to get benefits of DI of Spring framework.

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



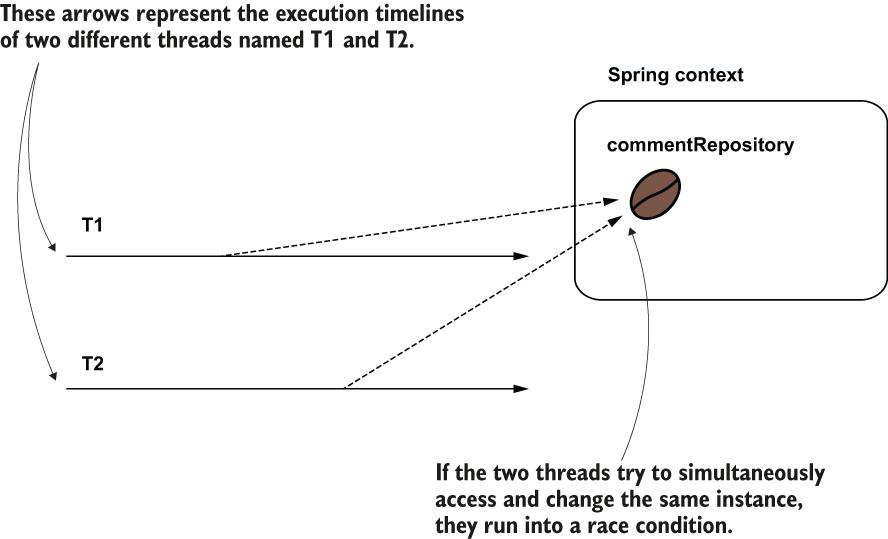
In this scenario, the CommentProcessor bean needs to become a bean to benefit from the DI capability Spring offers. In general, in any case where we want Spring to augment the object with a specific capability, it needs to be a bean.

**What is difference between singleton and prototype bean?**

As a recap, let's have a quick comparison between singleton and prototype scopes. [Table 5.1](https://cdn2.percipio.com/1692975502.860fafeff02c3cb9889eac06378d5064fd2c42fd/eod/books/158139/OEBPS/section-52-34.xhtml#ch05table01) shows their characteristics side-by-side.

| **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. 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. 4. You need to explicitly mark a bean as a prototype. 5. A prototype bean can have mutable attributes. |

**What race condition in singleton bean scope?**

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.

**What are Spring capabilities?**

1. DI
2. AOP
3. Transaction Management
4. Security Management

**Why should I bother to learn aspects or AOP?**

At high level:

1. Transaction management
2. Spring Security

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**, which we'll discuss in [chapter 13](https://cdn2.percipio.com/1693028711.6d4062ad021410993e569cc5864daa29900add5c/eod/books/158139/OEBPS/chapter-13-72.xhtml#ch13). 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. To properly understand what happens in apps using these functionalities, you first need to learn aspects.

**What are 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. It allows the developer to focus only on the relevant details discussed when reading the method logic.

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.



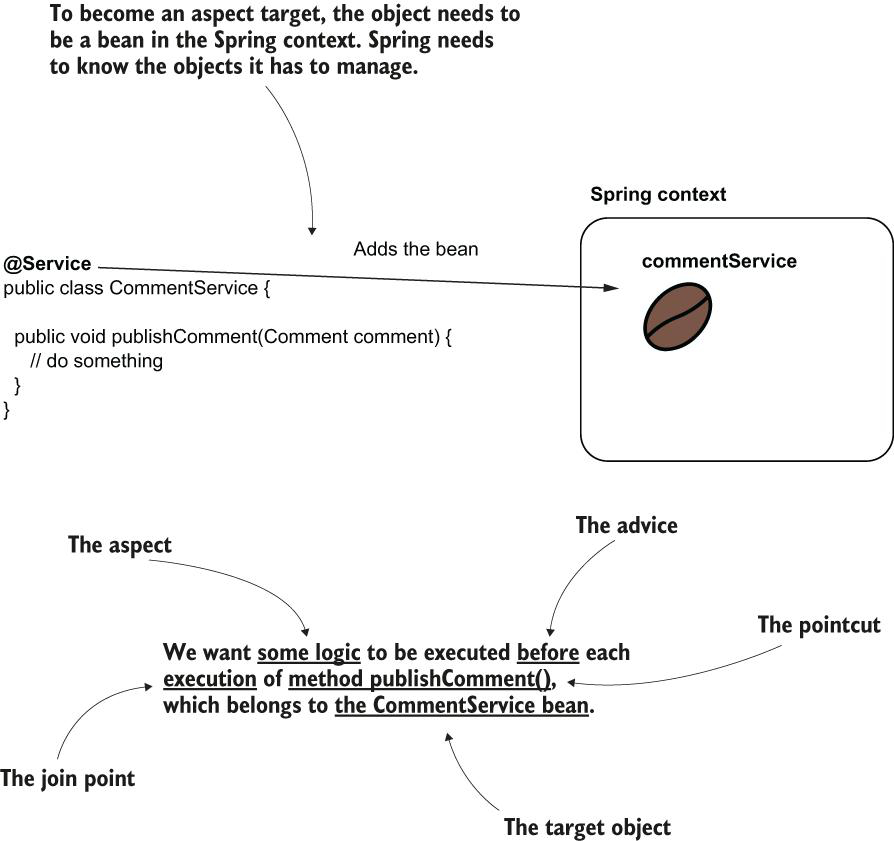
**What are main components you define when designing an aspect?**

An aspect is simply 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**.
* **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**.
* **Which methods the framework needs to intercept** and execute the aspect for them. This is named a **pointcut**.

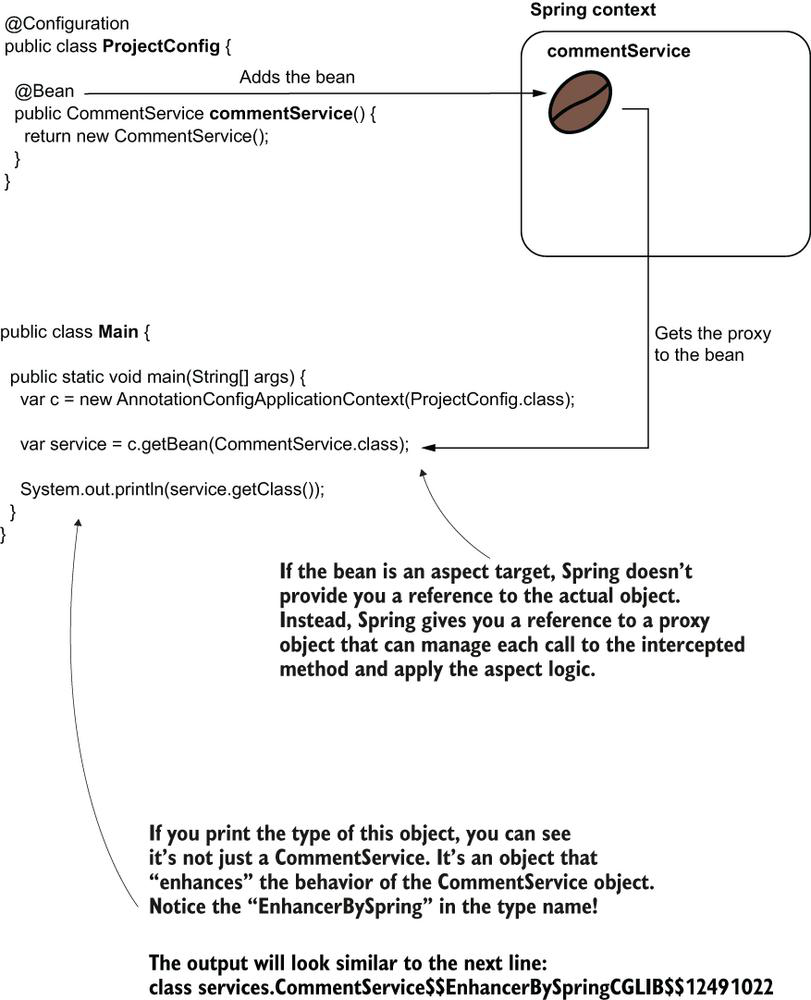
With aspects terminology, you'll also find the concept of a **join point**, which defines the event that triggers the execution of an aspect. But with Spring, **this event(join point) is always a method call.**

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.

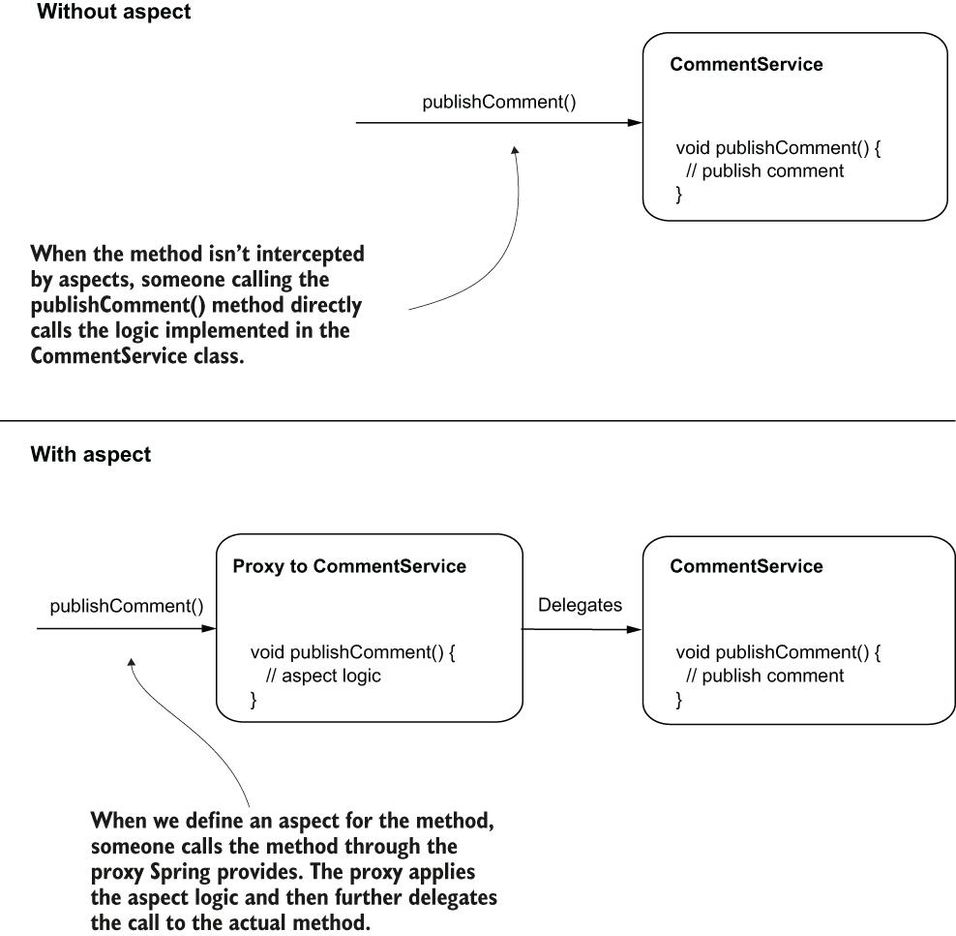


**What is weaving an aspect?**

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.



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.



**What are different logging frameworks?**

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>)
* Slf4j

**Do we have any logger available in JDK itself?**

**How do you tell Spring that you want to use specific mechanism provided by Spring?**

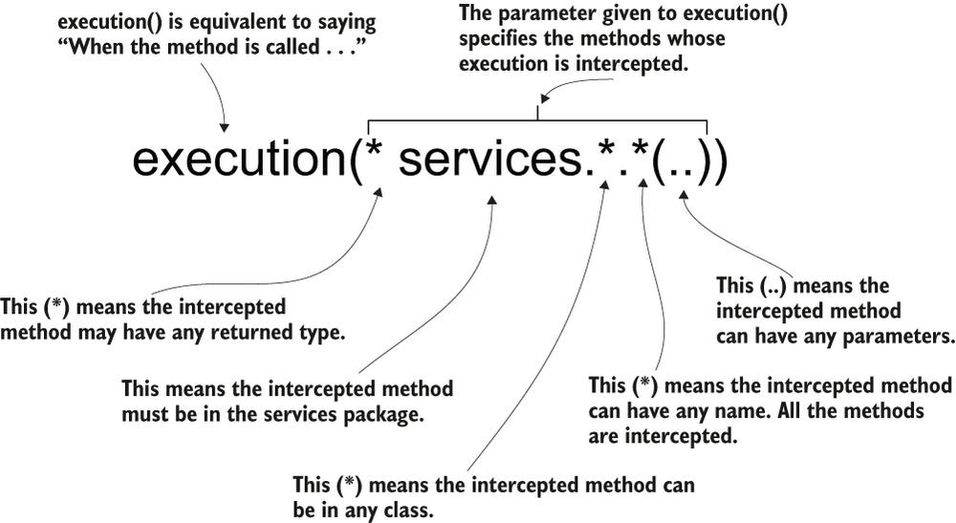
Whenever you use a specific mechanism provided by Spring, you have to explicitly enable it by annotating your configuration class with a particular annotation. In most cases, the names of these annotations start with “Enable.” You will learn more such annotations that enable different Spring capabilities as you progress through the book. In this example, we need to use the @EnableAspectJAutoProxy annotation to enable the aspect capabilities.

**What is AspectJ pointcut language?**

This expression language is called AspectJ pointcut language, and you won't need to learn it by heart to use it. In practice, you don't use complex expressions. When I need to write such an expression, I always refer to the documentation.

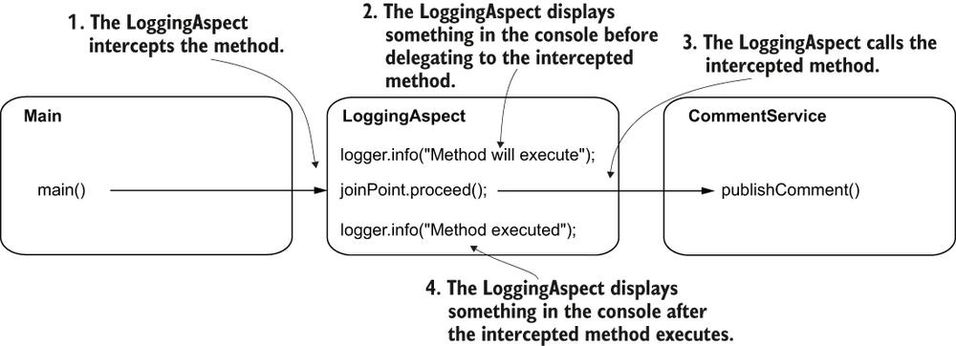
Theoretically, you can write very complex AspectJ pointcut expressions to identify a particular set of method calls to be intercepted. This language is really powerful. But as we'll discuss later in this chapter, it's always better to avoid writing complex expressions. In most cases, you can find simpler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

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.

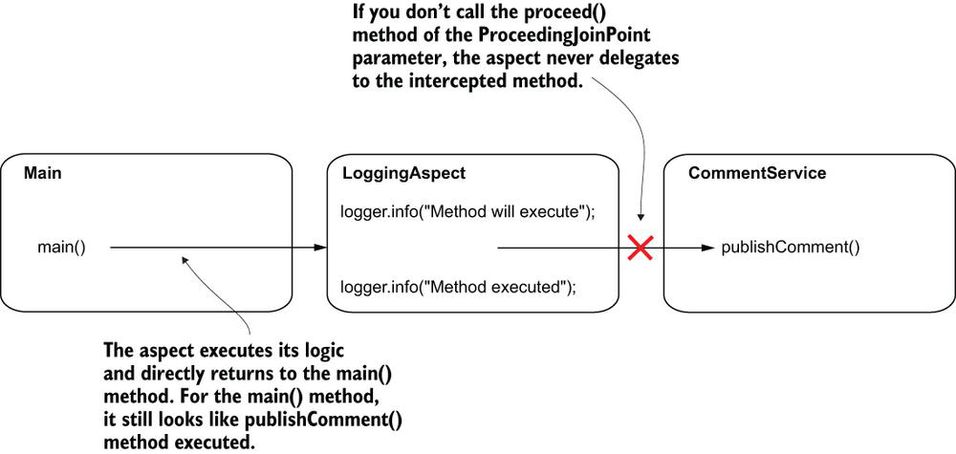


**Can you explain an aspect behaviour in simpler terms?**

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



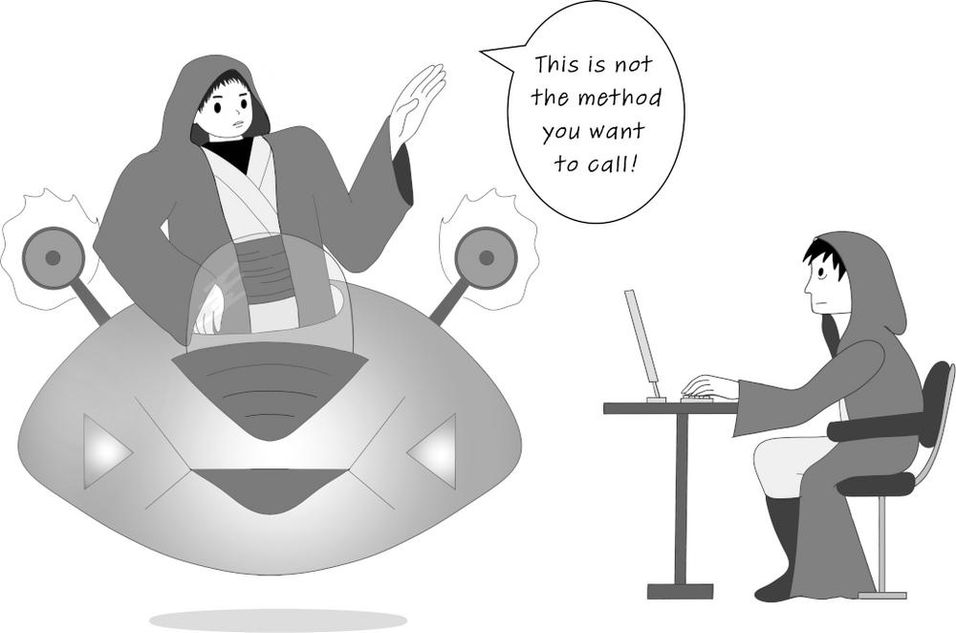
If you don't call the proceed() method of the ProceedingJoinPoint parameter of the aspect, the aspect never delegates further to the intercepted method. In this case, the aspect simply executes instead of the intercepted method. The caller of the method doesn't know that the real method is never executed.



**What is the use case when aspect does not call actual method?**

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](https://cdn2.percipio.com/1693038255.679baff89bdca76a4a07b3eadc533d6b158127b0/eod/books/158139/OEBPS/section-62-38.xhtml#ch06fig09)).

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.



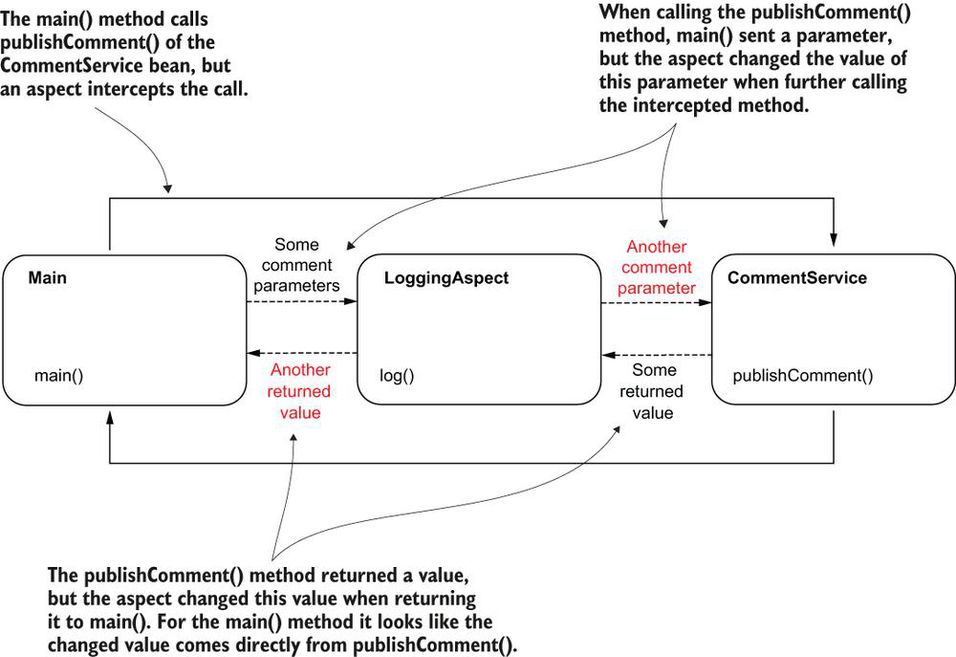
**What are some powerful use cases of Aspects?**

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 behaviour completely. 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. When considering writing an aspect, put yourself in the developer's shoes. Someone who needs to understand the code should easily realize what's happening.

An aspect can change the parameters used to call the intercepted method and the returned value received by the intercepted method's caller. This approach is powerful, and it gives flexible control of the intercepted method.



I know I repeat myself, but this point is quite important. 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!

**Why it is so important to understand how aspects work?**

By first understanding how aspects work you gain a significant advantage in understanding Spring. I often see developers starting to use a framework without understanding what's behind the functionalities they use. Not surprisingly, in many cases, these developers introduce bugs or vulnerabilities to their apps, or they make them less performant and maintainable. My advice is to always learn how things work before using them.

**How do intercept as any annotated method?**

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.



@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

@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

@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

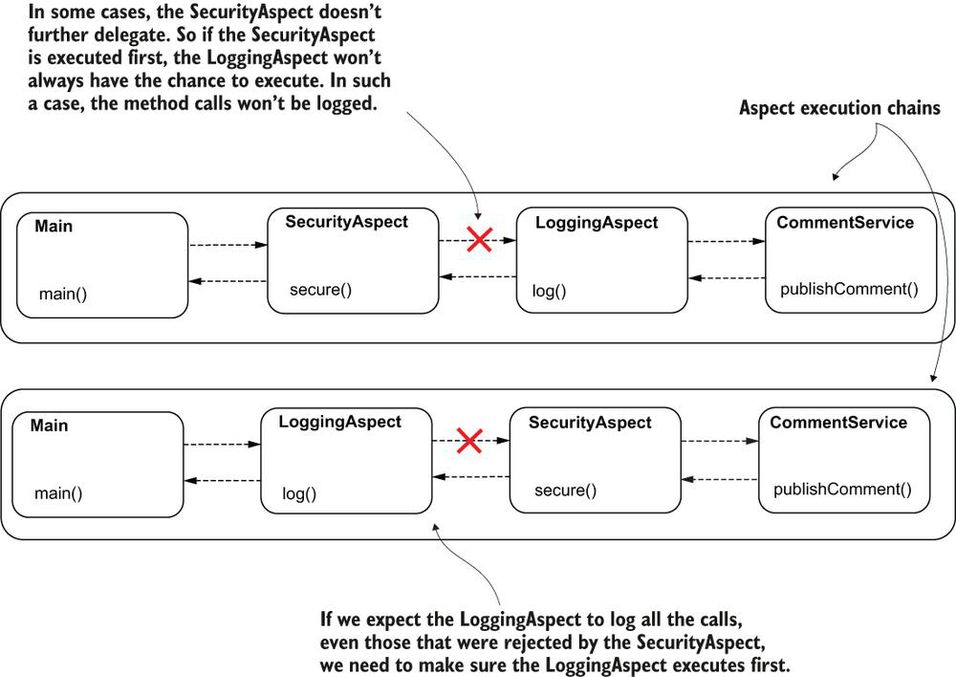
}

}

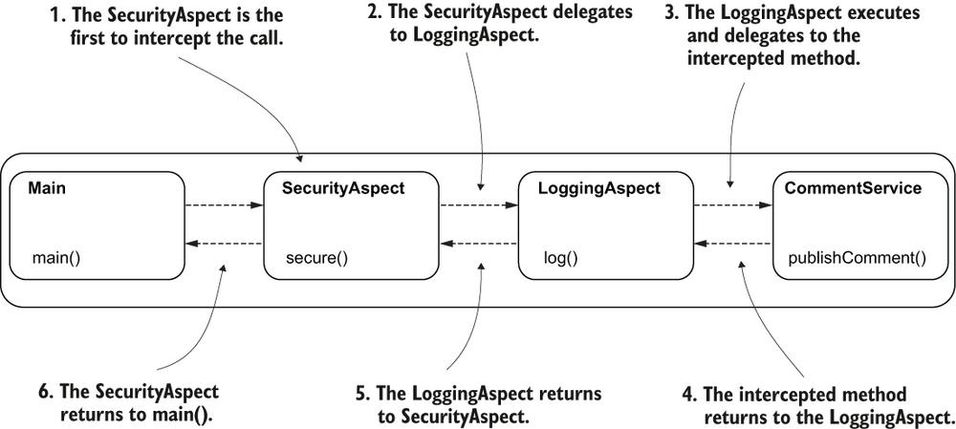
❶ Weaving the aspect to the methods annotated with @ToLog.

**What is aspect execution training?**

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.



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

Larger View

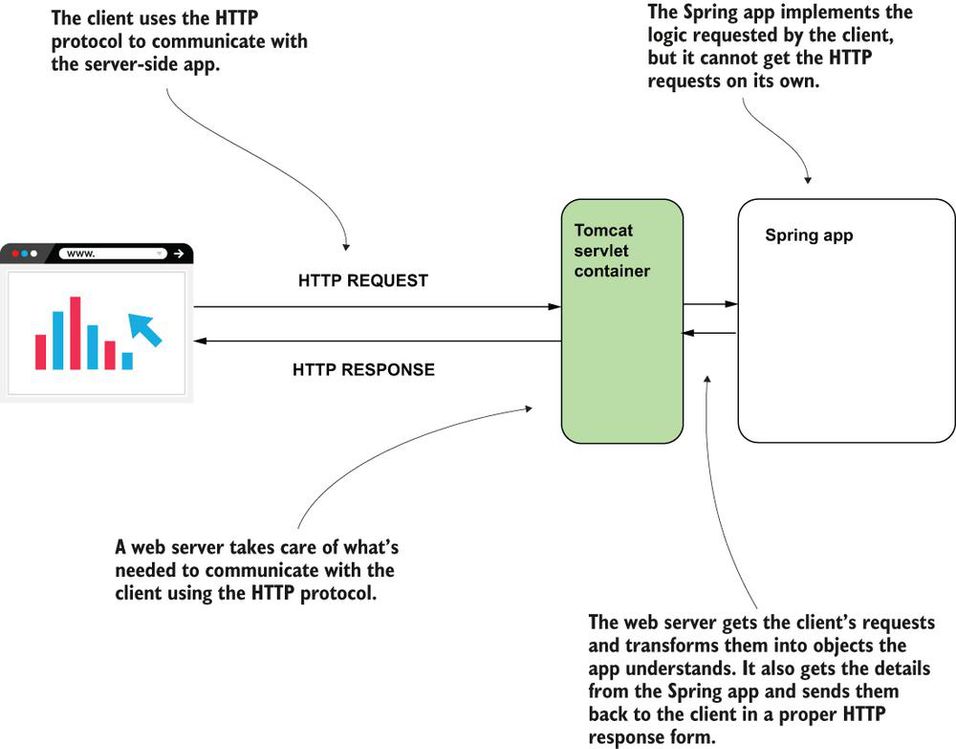
**What is Web App?**

Any app you access through your web browser is a web app.Years ago, we used desktop apps installed on our computers for almost anything we were doing ([figure 7.1](https://cdn2.percipio.com/1693056808.74731d81a2c7670fc1f4814b9f0957246a77f050/eod/books/158139/OEBPS/section-71-43.xhtml#ch07fig01)). With time, most of these apps became accessible via a web browser. Accessing an app in a browser makes it more comfortable to use. You don't have to install anything, and you can use it from any device that has access to the internet, such as a tablet or smartphone.

**What is importance of using servlet container in web development?**

What you need is not only something that understands HTTP, but something that can translate the HTTP request and response to a Java app. This something is a **servlet container (sometimes referred to as a web server)**: a translator of the HTTP messages for your Java app. This way, your Java app doesn't need to take care of implementing the communication layer. One of the most appreciated servlet container implementations is Tomcat.

A servlet container (e.g., Tomcat) speaks HTTP. It translates the HTTP request to our Spring app and the app's response into an HTTP response. This way, we don't need to care about the protocol used for communication on the network, as we simply write everything as Java objects and methods.



The servlet container (Tomcat) registers multiple servlet instances. Each servlet is associated with a path. When the client sends a request, Tomcat calls a method of the servlet associated with the path the client requested. The servlet gets the values on the request and builds the response that Tomcat sends back to the client.

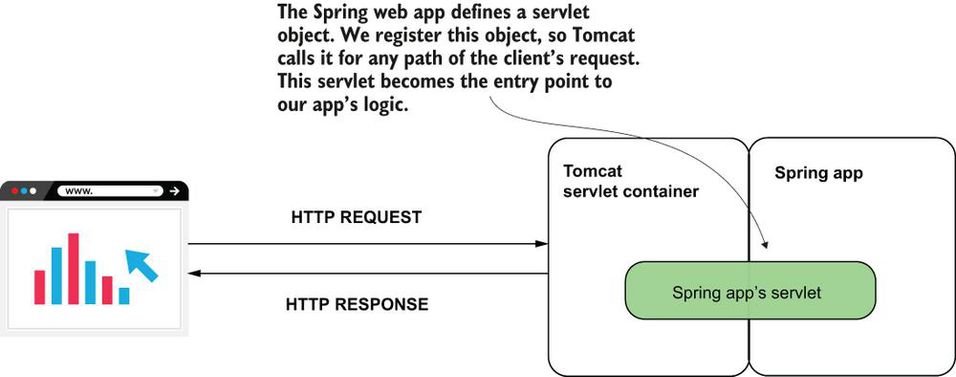
A diagram of a server

Description automatically generated

**How working with servlet changes with Spring web App?**

we don't typically create servlet instances. We'll use a servlet with the Spring apps we develop with Spring, but you won't need to write this yourself, so you don't have to focus on learning to implement servlets. But you do need to remember the servlet is the entry point to your app's logic. It's the component the servlet container (Tomcat, in our case) directly interacts with. It's how the request data enters your app and how the response goes through Tomcat back to the client ([figure 7.8](https://cdn2.percipio.com/1693058676.ca8d277077cd465d8f876b05850aea2128df9e7f/eod/books/158139/OEBPS/section-71-43.xhtml#ch07fig08)).

Figure 7.8: The Spring app defines a servlet object and registers it into the servlet container. Now both Spring and the servlet container know this object and can manage it. The servlet container calls this object for any client request, allowing the servlet to manage the request and the response.



**How Spring boot help in creating Spring web apps?**

To create a Spring web app, we need to configure a servlet container, create a servlet instance, and then make sure we correctly configure this servlet instance such that Tomcat calls it for any client request. What a headache to write so many configurations! Many years ago, when I was teaching Spring 3 (the latest Spring version at that time) and we configured web apps, this was the part both the students and I hated the most. Fortunately, times changed, and today I don't have to bother you by teaching such configurations.

In this section, we'll discuss Spring Boot, a tool for implementing modern Spring apps. Spring Boot is now one of the most appreciated projects in the Spring ecosystem. It helps you create Spring apps more efficiently and focus on the business code you write by eliminating a huge part of the code you used to write for configurations. Especially in a world of service-oriented architectures (SOA) and microservices, where you create apps more often (discussed in [appendix A](https://cdn2.percipio.com/1693058676.ca8d277077cd465d8f876b05850aea2128df9e7f/eod/books/158139/OEBPS/appendix-A-86.xhtml#app01)), avoiding the pain of writing configurations is helpful.

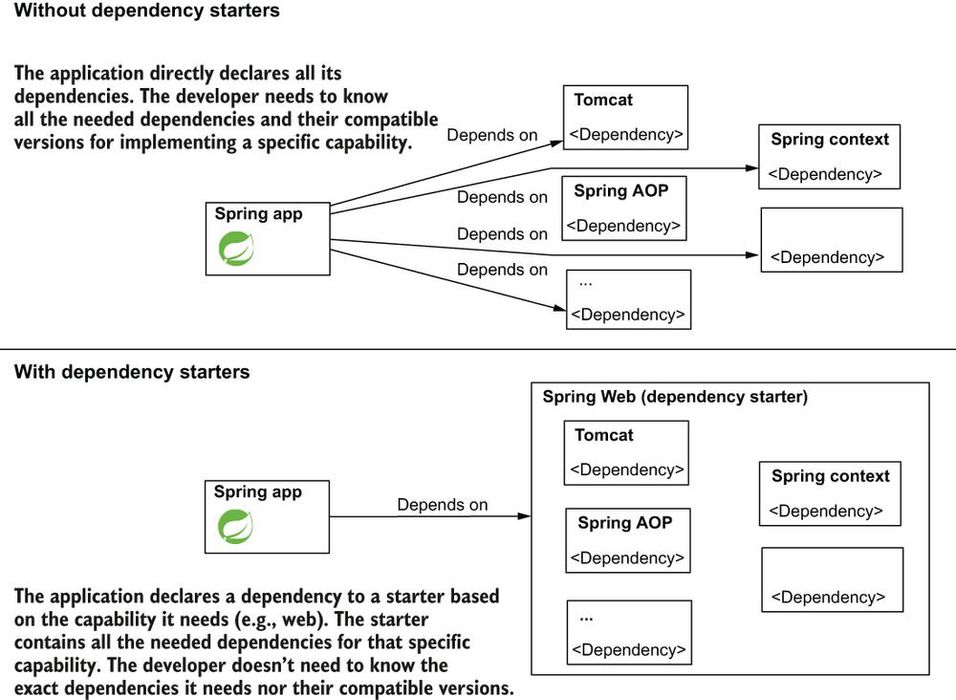
**What are some critical Spring boot features?**

* **Simplified project creation**—You can use a project initialization service to get an empty but configured skeleton app.
* **Dependency starters**—Spring Boot groups certain dependencies used for a specific purpose with dependency starters. You don't need to figure out all the must-have dependencies you need to add to your project for one particular purpose nor which versions you should use for compatibility.
* **Autoconfiguration based on dependencies**—Based on the dependencies you added to your project, Spring Boot defines some default configurations. Instead of writing all the configurations yourself, you only need to change the ones provided by Spring Boot that don't match what you need. Changing the configs likely requires less code (if any).

**What is significance of dependency starters?**

With dependency starters, we don't request dependencies directly. We request capabilities ([figure 7.14](https://cdn2.percipio.com/1693060571.77119d61370e2047fa3e2372c50c723f004514f5/eod/books/158139/OEBPS/section-72-44.xhtml#ch07fig14)). You add a dependency starter for a particular capability you need, say web functionalities, a database, or security. Spring Boot makes sure to add the right dependencies to your app with the proper compatible version for your requested capability. We can say that dependency starters are capability-oriented groups of compatible dependencies.

Using dependency starters. Instead of individually referring to specific dependencies, the app now depends on only a starter. The starter contains all the needed dependencies for implementing a specific capability. The starter also makes sure these dependencies are compatible with one another.



Look at your pom.xml file. You only added the spring-boot-starter-web dependency, no Spring context, no AOP, no Tomcat! But, if you look in the “External Libraries” folder of your app, you'll find JAR archives for all these. Spring Boot knew you would need them and downloaded them with specific versions it knows are compatible.

**What is Autoconfiguration by Convention Based on Dependencies?**

We say that it applies the **convention-over-configuration** principle.

Just start your app, and you'll understand why. Yes, I know, you didn't even write anything yet—only downloaded the project and opened it in your IDE. But you can start the app, and you'll find your app boots a Tomcat instance by default accessible on port 8080. In your console, you find something similar to the next snippet:

Tomcat started on port(s): 8080 (http) with context path '' ❶

Started Main in 1.684 seconds (JVM running for 2.306)

❶ Spring Boot configured Tomcat and starts it by default on port 8080.

Based on the dependencies you added, Spring Boot realizes what you expect from your app and provides you some default configurations. Spring Boot gives you the configurations, which are generally used for the capabilities you requested when adding the dependencies.

For example, Spring knows when you added the web dependency you need for a servlet container and configures you a Tomcat instance because, in most cases, developers use this implementation. For Spring Boot, Tomcat is the convention for a servlet container.

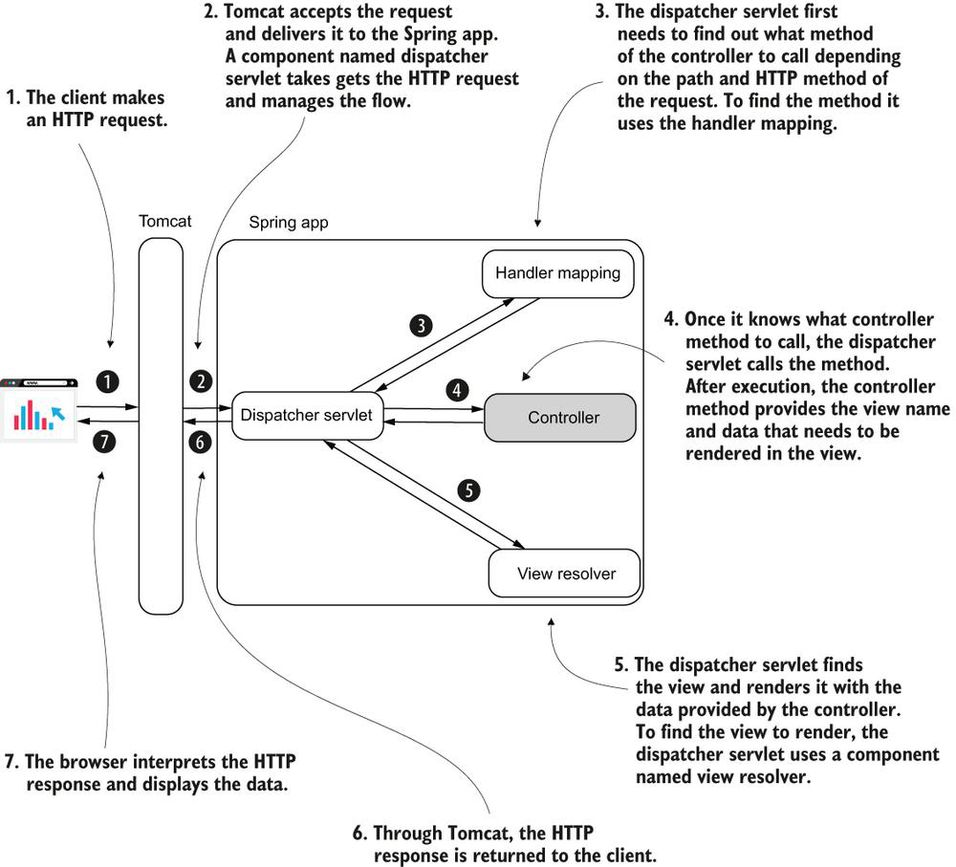
The convention represents the most-used way to configure the app for a specific purpose. Spring Boot configures the app by convention such that you now only need to change those places where your app needs a more particular configuration. With this approach, you'll write less code for configuration (if any).

**What is default folder location where Spring boot looks for static html documents?**

The “resources/static” folder of your Maven project. This folder is the default place where the Spring Boot app expects to find the pages to render.

**Can you explain typical Spring mvc architecture?**

he Spring MVC architecture. In the diagram, you find the main components of Spring MVC. These components and the way they collaborate are responsible for a web app's behavior. The controller (shaded differently) is the only component you implement. Spring Boot configures the other components.



**What is template engine, Thymeleaf?**

Thymeleaf, the template engine is a dependency that allows us to easily send data from the controller to the view and display this data in a specific way. I chose Thymeleaf because it's less complex than others, and I find it easier to understand and learn. As you'll observe in our example, the templates used with Thymleaf are simple HTML static files.

The next code snippet shows the dependency you need to add to the pom.xml file:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId> ❶

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId> ❷

</dependency>

❶ The dependency starter that needs to be added to use Thymeleaf as a template engine.

❷ Though you're building a web app, you still need to add the dependency starter for web apps.

**How does a typical controller look like in Spring MVC?**

@Controller ❶

public class MainController {

@RequestMapping("/home") ❷

public String home(Model page) { ❸

page.addAttribute("username", "Katy"); ❹

page.addAttribute("color", "red"); ❹

return "home.html"; ❺

}

}

❶ The @Controller stereotype annotation marks this class as Spring MVC controller and adds a bean of this type to the Spring context.

❷ We assign the controller's action to an HTTP request path.

❸ The action method defines a parameter of type Model that stores the data the controller sends to the view.

❹ We add the data we want the controller to send to the view.

❺ The controller's action returns the view to be rendered into the HTTP response.

Students sometimes ask me why they get an error if they directly add to the browser's address bar “localhost:8080” without a path like “/home.” It's correct that an error appears. The error is a default page you see displayed by a Spring Boot app when you get an HTTP 404 (Not Found) response status. When you call directly “localhost:8080” you refer to the path “/.” Because you didn't assign any controller action to this path, it's normal to get an HTTP 404. If you wish to see something else instead, assign a controller action to this path as well using the @RequestMapping annotation.

**How do you define a typical view file in Spring MVC?**

To define the view, you need to add a new “home.html” file to your Spring Boot project's “resources/templates” folder. Be attentive to the small difference: in [chapter 7](https://cdn2.percipio.com/1693403024.bb668e6de2e7bff827fba1473cfdad45d1feb35e/eod/books/158139/OEBPS/chapter-7-42.xhtml#ch07), we added the HTML file in the “resources/static” folder because we created a static view. Now that we're using a template engine to create a dynamic view, you need to add the HTML file to the “resources/templates” folder instead.

[Listing 8.2](https://cdn2.percipio.com/1693403024.bb668e6de2e7bff827fba1473cfdad45d1feb35e/eod/books/158139/OEBPS/section-81-48.xhtml#ch08list02) shows the content of the “home.html” file I added to the project. The first important thing to notice in the file's content is the <html> tag where I added the attribute xmlns:th=“http://www.thymeleaf.org”. This definition is equivalent to an import in Java. It allows us further to use the prefix “th” to refer to specific features provided by Thymeleaf in the view.

A little bit further in the view, you find two places where I used this “th” prefix to refer to the controller's data to the view. With the ${attribute\_key} syntax, you refer to any of the attributes you send from the controller using the Model instance. For example, I used the ${username} to get the value of the “username” attribute and ${color} to get the value of the “color” attribute.

#### Listing 8.2: The home.html file representing the dynamic view of the app

<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org"> ❶

<head>

<meta charset="UTF-8">

<title>Home Page</title>

</head>

<body>

<h1>Welcome

<span th:style="'color:' + ${color}" ❷

th:text="${username}"></span>!</h1> ❷

</body>

</html>

❶ Defines the Thymeleaf “th” prefix

❷ Uses the “th” prefix to use the values sent by the controller

**What are different ways by which client can sends data through HTTP request?**

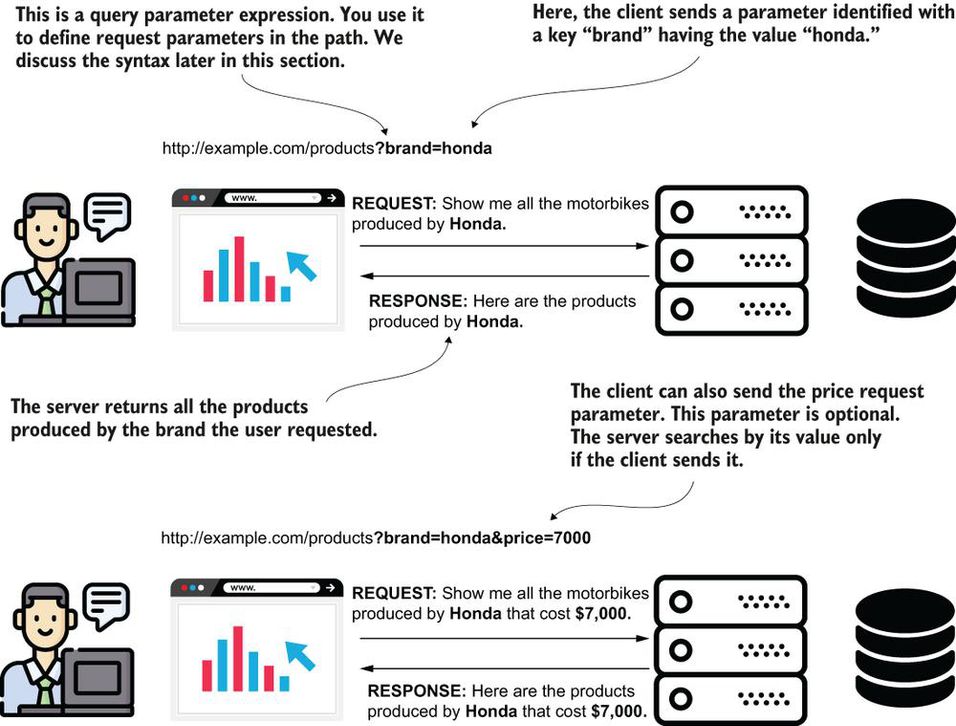
In most cases, to send data through the HTTP request you use one of the following ways:

* An HTTP request parameter represents a simple way to send values from client to server in a key-value(s) pair format. To send HTTP request parameters, you append them to the URI in a request query expression. They are also called query parameters. You should use this approach only for sending a small quantity of data.
* An HTTP request header is similar to the request parameters in that the request headers are sent through the HTTP header. The big difference is that they don't appear in the URI, but you still cannot send large quantities of data using HTTP headers.
* A path variable sends data through the request path itself. It is the same as for the request parameter approach: you use a path variable to send a small quantity of data. But we should use path variables when the value you send is mandatory.
* The HTTP request body is mainly used to send a larger quantity of data (formatted as a string, but sometimes even binary data such as a file).

**What are use cases of using query parameters?**

You use request parameters in the following scenarios:

* The quantity of data you send is not large. You set the request parameters using query variables (as shown in this section's example). This approach limits you to about 2,000 characters.
* You need to send optional data. A request parameter is a clean way to deal with a value the client might not send. The server can expect to not get a value for specific request parameters.

A request parameter can be optional. A common scenario for using request parameters is implementing a search functionality where the search criteria are optional. The client sends only some of the request parameters, and the server knows to use only the values it receives. You implement the server to consider it might not get values for some of the parameters.

**Are request parameters and query parameters same ?**

Yes

An **HTTP request parameter** represents a simple way to send values from client to server in a key-value(s) pair format. To send HTTP request parameters, you append them to the URI in a request query expression. They are **also called query parameters**.

**Can you give typical example of using request parameter?**

@Controller

public class MainController {

@RequestMapping("/home")

public String home(

@RequestParam String color, ❶

Model page) { ❷

page.addAttribute("username", "Katy");

page.addAttribute("color", color); ❸

return "home.html";

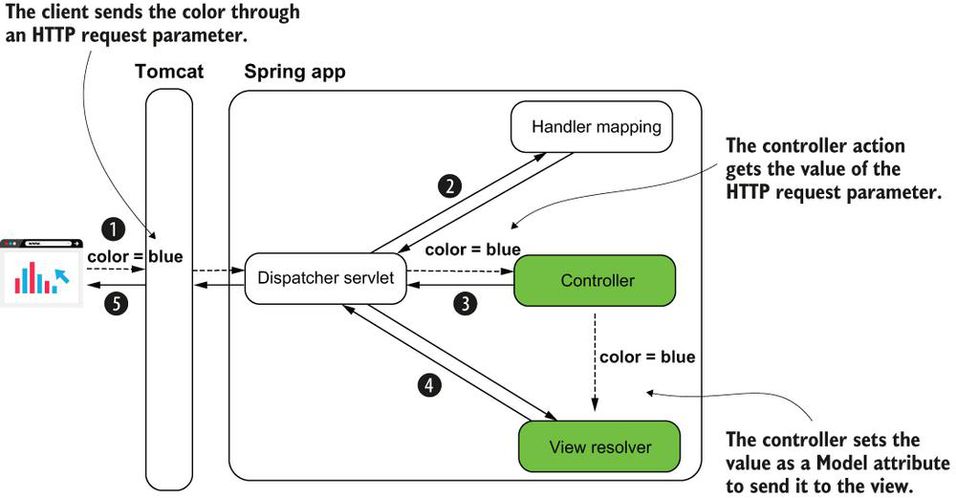
}

}

❶ We define a new parameter for the controller's action method and annotate it with @RequestParam.

❷ We also add the Model parameter that we use to send data from the controller to the view.

❸ The controller passes the color sent by the client to the view.

 The value sent by the client from the Spring MVC perspective. The controller action gets the request parameters the client sends and can use them. In our example, the value is set on the Model and delivered to the view.

**Is request parameter mandatory by default?**

A request parameter is mandatory by default. If the client doesn't provide a value for it, the server sends back a response with the status HTTP “400 Bad Request.” If you wish the value to be optional, you need to explicitly specify this on the annotation using the optional attribute: @RequestParam(optional=true).

**What is difference between request parameter and path variable?**

| **Request parameters** | **Path variables** |
| --- | --- |
| 1.Can be used with optional values.  2.It is recommended that you avoid a large number of parameters. If you need to use more than three, I recommend you use the request body. Avoid sending more than three query parameters for readability.  3.Some developers consider the query expression more difficult to read than the path expression. | 1.Should not be used with optional values.  2.Always avoid sending more than three path variables. It's even better if you keep a maximum of two.  3.Easier to read than a query expression. For **a publicly exposed website, it's also easier for search engines (e.g., Google) to index the pages. This advantage might make the website easier to find through a search engine**. |

**Can you explain typical path variable example in an HTTP request?**

@Controller

public class MainController {

@RequestMapping("/home/{color}") ❶

public String home(

@PathVariable String color, ❷

Model page) {

page.addAttribute("username", "Katy");

page.addAttribute("color", color);

return "home.html";

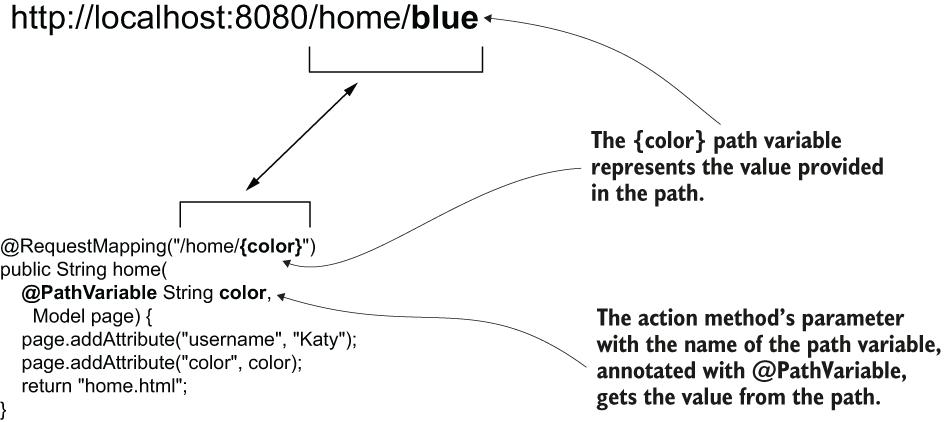
}

}

❶ To define a path variable, you assign it a name and put it in the path between curly braces.

❷ You mark the parameter where you want to get the path variable value with the @PathVariable annotation. The name of the parameter must be the same as the name of the variable in the path.

Using path variables. To get a value from a path variable, you give the variable a name between curly braces when defining the path on the controller action. You use a parameter annotated with @PathVariable to get the value of the path variable.



**Can we use HTTP method again their designed purpose?**

You can use an HTTP method against its designed purpose, but this is incorrect. For example, you could use HTTP GET and implement a functionality that changes data. Technically, this is possible, but it's a bad, bad choice. Never use an HTTP method against its designed purpose.

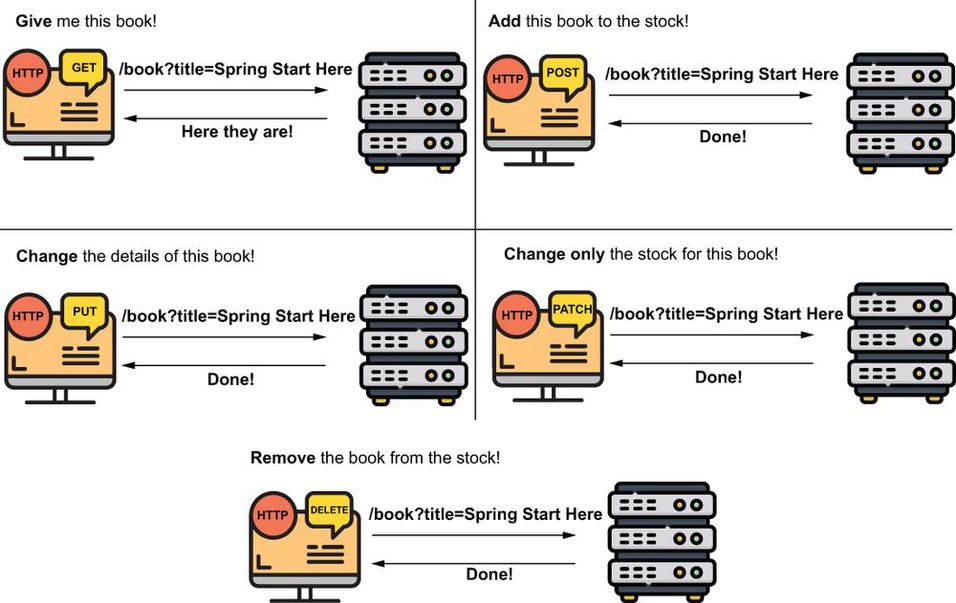
**How can we use request path to do multiples actions?**

In a complex scenario you can assign the same path to multiple actions of the controller as long as you use different HTTP methods

**What are different HTTP methods you would often you in web apps?**

| **HTTP method** | **Description** |
| --- | --- |
| GET | The client's request only retrieves data. |
| POST | The client's request sends new data to be added by the server. |
| PUT | The client's request changes a data record on the server side. |
| PATCH | The client's request partially changes a data record on the server side. |
| DELETE | The client's request deletes data on the server side. |

The basic HTTP methods. You use GET for retrieving data, POST for adding data, PUT for changing a record, PATCH for changing a part of the record, and DELETE to remove data. The client must use the appropriate HTTP method to express the action executed by a specific request.



**What is problem with default singleton scope bean?**

@Service

public class ProductService {

private List<Product> products = new ArrayList<>();

public void addProduct(Product p) {

products.add(p);

}

public List<Product> findAll() {

return products;

}

}

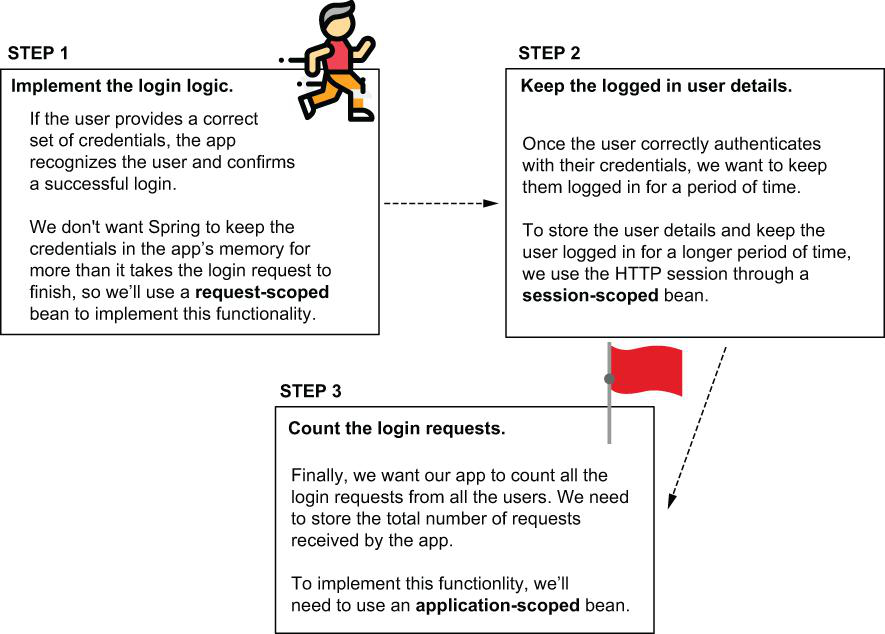
This design is a simplification to allow you to focus on the discussion of the HTTP methods. Remember that a Spring bean's scope by default is singleton, as we discussed in [chapter 5](https://cdn2.percipio.com/1693416467.e6d64506d5c85a1c88a4dbf31ad405bc8b5fda7e/eod/books/158139/OEBPS/chapter-5-32.xhtml#ch05), and a web application implies multiple threads (one for each request). Changing a list defined as an attribute of the bean would cause race condition situations in a real-world app where more clients add products simultaneously. For now, we'll keep our simplification, because in the next chapters we'll replace the list with a database, so this problem will no longer occur. But keep in mind this is a vicious approach, and, as we discussed in [chapter 5](https://cdn2.percipio.com/1693416467.e6d64506d5c85a1c88a4dbf31ad405bc8b5fda7e/eod/books/158139/OEBPS/chapter-5-32.xhtml#ch05), you shouldn't use something similar in a production-ready app. Singleton beans aren't thread-safe!

**What are web scopes of beans in Spring web applications?**

In web apps you can use other bean scopes that are relevant only to web applications. We call them web scopes:

* Request scope—Spring creates an instance of the bean class for every HTTP request. The instance exists only for that specific HTTP request.
* Session scope—Spring creates an instance and keeps the instance in the server's memory for the full HTTP session. Spring links the instance in the context with the client's session.
* Application scope —The instance is unique in the app's context, and it's available while the app is running.

We'll implement the login functionality in three steps. For each step we implement, we'll need to use a different bean scope. In [section 9.1](https://cdn2.percipio.com/1693469774.9bd06c5468a79a1d13733a50470bce548a816d52/eod/books/158139/OEBPS/section-91-52.xhtml#ch09lev1sec2), we'll use a request-scoped bean to implement the login logic without risking storing the credentials for longer than the login request. We'll then decide what details we need to store for the authenticated user in a session-scoped bean. Finally, we'll implement a feature to count all the login requests, and we'll use an application-scoped bean to keep the number.



**What is request scope bean?**

For every HTTP request, Spring provides a new instance for the request-scoped bean. When using a request-scoped bean, you can be sure the data you add on the bean is available only on the HTTP request that created the bean. Spring manages the bean type (the plant) and uses it to get instances (coffee beans) for each new request.

A diagram of a computer process

Description automatically generated

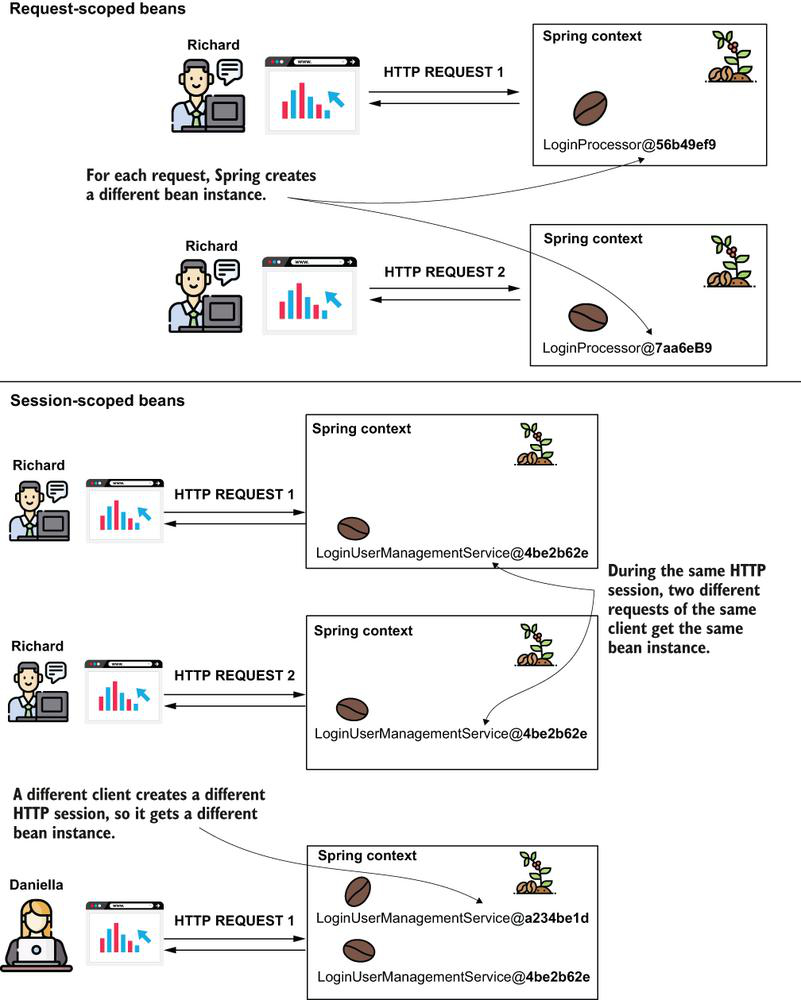
**What are key things to consider while using request scope beans?**

Keep in mind the very relevant aspects of request-scoped beans, explained in the following table.

| **Fact** | **Consequence** | **To consider** | **To avoid** |
| --- | --- | --- | --- |
| Spring creates a new instance for every HTTP request from any client. | Spring creates a lot of instances of this bean in the app's memory during its execution. | The number of instances is usually not a big problem because these instances are short-lived. The app doesn't need them for more than the time the HTTP request needs to complete. Once the HTTP request completes, the app releases the instances, and they are garbage-collected. | However, make sure you don't implement a time-consuming logic Spring needs to execute to create the instance (like getting data from a database or implementing a network call). Avoid writing logic in the constructor or a @PostConstruct method for request-scoped beans. |
| Only one request can use an instance of a request-scoped bean. | Instances of request-scoped beans are not prone to multithread-related issues as only one thread (the one of the request) can access them. | You can use the instance's attributes to store data used by request. | Don't use synchronization techniques for the attributes of these beans. These techniques would be redundant, and they only affect the performance of your app. |

**Can you explain difference between request scope and session scope beans?**

 A comparison between the request-scoped and session-scoped beans to help you visualize the differences between these two web bean scopes. You use request-scoped beans when you want Spring to create a new instance for each request. You use a session-scoped bean when you want to keep the bean (together with any details it holds) throughout the client's HTTP session



**What are JPA and JDBC?**

Like JDBC, JPA is purely a specification, and many commercial and open-source products such as Hibernate and TopLink provide JPA implementations.

Java Persistence API, or JPA, is a standard-based API for accessing, storing, and managing data between Java objects and relational database.

**What is the difference between @Inject and @Autowired in Spring Framework? Which one to use under what condition?**

Assuming here you're referring to the [javax.inject.Inject](http://download.oracle.com/javaee/6/api/javax/inject/Inject.html) annotation. @Inject is part of the Java CDI ([Contexts and Dependency Injection](http://docs.oracle.com/javaee/6/tutorial/doc/giwhb.html)) standard introduced in Java EE 6 (JSR-299), [read more](http://www.oracle.com/technetwork/articles/javaee/javaee6overview-141808.html). Spring has chosen to support using the @Inject annotation synonymously with their own @Autowired annotation.

So, to answer your question, @Autowired is Spring's own annotation. @Inject is part of a Java technology called CDI that defines a standard for dependency injection similar to Spring. In a Spring application, the two annotations works the same way as Spring has decided to support some JSR-299 annotations in addition to their own.

**How do you explain distributed applications?**

For me, it’s an amazing thing to see an application work across dozens (even hundreds) of servers. It’s like watching an orchestra playing a piece of music. While the final product of an orchestra is beautiful, the making of it is often a lot of hard work and requires a significant amount of practice. The same goes for writing a massively distributed application.

Since I entered the software development field 25 years ago, I’ve watched the industry struggle with the “right” way to build distributed applications. I’ve seen distributed service standards such as CORBA rise and fall. Monstrously big companies have tried to push big and, often, proprietary protocols. Anyone remember Microsoft’s Distributed Component Object Model (DCOM) or Oracle’s J2EE’s Enterprise Java Beans 2 (EJB)? I watched as technology companies and their followers rushed to build service-oriented architectures (SOA) using heavy XML-based schemas.

**What is a microservice?**

 A microservice architecture focuses on building small services that use simple protocols (HTTP and JSON) to communicate. That’s it. You can write a microservice with nearly any programming language. There’s beauty in this simplicity.

**Why Spring cloud came into being?**

The Spring Cloud framework provides out-of-the-box solutions for many of the common development and operational problems you’ll run into as a microservice developer. Spring Cloud lets you use only the pieces you need and minimizes the amount of work you need to do to build and deploy production-ready Java micro-services. It does this by using other battle-hardened technologies from companies and groups such as Netflix, HashiCorp, and the Apache foundation.

Building an individual microservice is easy, operationalizing and scaling it is difficult. Getting hundreds of small distributed components to work together and then building a resilient application from them can be incredibly difficult to do. In distributed computing, failure is a fact of life and how your application deals with it is incredibly difficult to get right. To paraphrase my colleagues Chris Miller and Shawn Hagwood: “If it’s not breaking once in a while, you’re not building.”