**Dependency Injection**

Any Java class we write depends on other classes. The other classes a class depends on are its dependencies. If a class directly creates instances of dependencies, a tight coupling is established between them. With Spring, the responsibility of creating and wiring objects is taken over by a new component called the **IoC container**. Classes define dependencies and the Spring **Inversion of Control** (**IoC**) container creates objects and wires the dependencies together. This revolutionary concept, where the control of creating and wiring dependencies is taken over by the container, is famously called IoC or **dependency** **injection** (**DI**).

In this chapter, we start with exploring the need for DI. We use a simple example to illustrate the use of DI. We will understand the important advantages of DI--easier maintainability, less coupling and improved testability. We will explore the DI options in Spring. We will end the chapter by looking at the standard DI specification for Java **Contexts and Dependency Injection** (**CDI**) and how Spring supports it.

This chapter will answer the following questions:

* What is dependency injection?
* How does proper use of dependency injection make applications testable?
* How does Spring implement DI with annotations?
* What is a component scan?
* What is the difference between Java and XML application contexts?
* How do you create unit tests for Spring contexts?
* How does mocking make unit testing simpler?
* What are the different bean scopes?
* What is CDI and how does Spring support CDI?

**Understanding dependency injection**

We will look at an example to understand dependency injection. We will write a simple business service that talks to a data service. We will make the code testable and see how proper use of DI makes the code testable.

The following is the sequence of steps we will follow:

1. Write a simple example of a business service talking to a data service. When a business service directly creates an instance of a data service, they are tightly coupled to one another. Unit testing will be difficult.
2. Make code loosely coupled by moving the responsibility of creating the data service outside the business service.
3. Bring in the Spring IoC container to instantiate the beans and wire them together.
4. Explore the XML and Java configuration options that Spring provides.
5. Explore Spring unit testing options.
6. Write real unit tests using mocking.

**Understanding dependencies**

We will start with writing a simple example; a business service talking to another data service. Most Java classes depend on other classes. These are called **dependencies** of that class.

Take a look at an example class BusinessServiceImpl, as follows:

public class BusinessServiceImpl {   
 public long calculateSum(User user) {   
 DataServiceImpl dataService = new DataServiceImpl();   
 long sum = 0;   
 for (Data data : dataService.retrieveData(user)) {   
 sum += data.getValue();   
 }   
 return sum;   
 }  
 }

Typically, all well-designed applications have multiple layers. Every layer has a well-defined responsibility. The business layer contains the business logic. The data layer talks to the external interfaces and/or databases to get the data. In the preceding example, the DataServiceImpl class gets some data related to the user from the database. BusinessServiceImpl class is a typical business service, talking to the data service DataServiceImpl for data and adding business logic on top of it (in this example, the business logic is very simple: calculate the sum of data returned by the data service).

BusinessServiceImpl depends on DataServiceImpl. So, DataServiceImpl is a dependency of BusinessServiceImpl.

Focus on how BusinessServiceImpl creates an instance of DataServiceImpl.

DataServiceImpl dataService = new DataServiceImpl();

BusinessServiceImpl creates an instance by itself. This is tight coupling.

Think for a moment about unit testing; how do you unit test the BusinessServiceImpl class without involving (or instantiating) the DataServiceImpl class? It's very difficult. One might need to do complicated things such as reflection to write a unit test. So, the preceding code is not testable.

*A piece of code (a method, a group of methods, or a class) is testable when you can easily write a simple unit test for it. One of the approaches used in unit testing is to mock the dependencies. We will discuss mocking in more detail later.*

Here's a question to think about: how do we make the preceding code testable? How do we reduce tight coupling between BusinessServiceImpl and DataServiceImpl?

The first thing we can do is to create an interface for DataServiceImpl. Instead of using the direct class, we can use the newly created interface of DataServiceImpl in BusinessServiceImpl.

The following code shows how to create an interface:

public interface DataService {   
 List<Data> retrieveData(User user);   
 }

Let's update the code in BusinessServiceImpl to use the interface:

DataService dataService = new DataServiceImpl();

*Using interfaces helps in creating loosely coupled code. We can replace the wire with any implementation of an interface into a well-defined dependency.  
  
For example, consider a business service that needs some sorting.  
  
The first option is to use the sorting algorithm directly in the code, for example, bubble sort. The second option is to create an interface for the sorting algorithm and use the interface. The specific algorithm can be wired in later. In the first option, when we need to change the algorithm, we will need to change the code. In the second option, all that we need to change is the wiring.*

We are now using the DataService interface, but BusinessServiceImpl is still tightly coupled as it is creating an instance of DataServiceImpl. How can we solve that?

How about BusinessServiceImpl not creating an instance of DataServiceImpl by itself? Can we create an instance of DataServiceImpl elsewhere (we will discuss who will create the instance later) and give it to BusinessServiceImpl?

To enable this, we will update the code in BusinessServiceImpl to have a setter for DataService. The calculateSum method is also updated to use this reference. The updated code is as follows:

public class BusinessServiceImpl {   
 private DataService dataService;   
 public long calculateSum(User user) {   
 long sum = 0;   
 for (Data data : dataService.retrieveData(user)) {   
 sum += data.getValue();   
 }   
 return sum;   
 }   
 public void setDataService(DataService dataService) {   
 this.dataService = dataService;   
 }   
 }

*Instead of creating a setter for the data service, we could have also created a BusinessServiceImplconstructor accepting a data service as an argument. This is called a***constructor injection***.*

You can see that BusinessServiceImpl can now work with any implementation of DataService. It is not tightly coupled with a specific implementation: DataServiceImpl.

To make the code even more loosely coupled (as we start writing the tests), let's create an interface for BusinessService and have BusinessServiceImpl updated to implement the interface:

public interface BusinessService {   
 long calculateSum(User user);   
 }   
 public class BusinessServiceImpl implements BusinessService {   
 //.... Rest of code..   
 }

Now that we have reduced coupling, one question remains still; who takes the responsibility for creating instance of the DataServiceImpl class and wiring it to the BusinessServiceImpl class?

That's exactly where the Spring IoC container comes into the picture.

**The Spring IoC container**

The Spring IoC container creates the beans and wires them together according to the configuration setup created by the application developer.

The following questions need to be answered:

* **Question 1**: How does the Spring IoC container know which beans to create? Specifically, how does the Spring IoC container know to create beans for the BusinessServiceImpl and DataServiceImpl classes?
* **Question 2**: How does the Spring IoC container know how to wire beans together? Specifically, how does the Spring IoC container know to inject the instance of the DataServiceImpl class into the BusinessServiceImpl class?
* **Question 3**: How does the Spring IoC container know where to search for beans? It is not efficient to search all packages in the classpath.

Before we can focus on creating a container, let's focus on questions 1 and 2; how to define what beans need to be created and how to wire them together.

# Defining beans and wiring

Let's address the first question; how does the Spring IoC container know which beans to create?

We need to tell the Spring IoC container which beans to create. This can be done using @Repository or @Component or @Service annotations on the classes for which beans have to be created. All these annotations tell the Spring Framework to create beans for the specific classes where these annotations are defined.

A @Component annotation is the most generic way of defining a Spring bean. Other annotations have more specific context associated with them. @Service annotation is used in business service components. @Repository annotation is used in **Data Access Object** (**DAO**) components.

We use @Repository annotation on DataServiceImpl because it is related to getting data from the database. We use @Service annotation on the BusinessServiceImpl class as follows, since it is a business service:

@Repository   
 public class DataServiceImpl implements DataService   
 @Service   
 public class BusinessServiceImpl implements BusinessService

Let's shift our attention to question 2 now--how does the Spring IoC container know how to wire beans together? The bean of the DataServiceImpl class needs to be injected into that of the BusinessServiceImpl class.

We can do that by specifying an @Autowired annotation on the instance variable of the DataService interface in the BusinessServiceImpl class:

public class BusinessServiceImpl {   
 @Autowired   
 private DataService dataService;

Now that we have defined the beans and their wiring, to test this, we need an implementation of DataService. We will create a simple, hardcoded implementation. DataServiceImpl returns a couple of pieces of data:

@Repository   
 public class DataServiceImpl implements DataService {   
 public List<Data> retrieveData(User user) {   
 return Arrays.asList(new Data(10), new Data(20));   
 }   
 }

Now that we have our beans and dependencies defined, let's focus on how to create and run a Spring IoC container.

**Creating a Spring IoC container**

There are two ways to create a Spring IoC container:

* Bean factory
* Application context

*Bean factory is the basis for all Spring IoC functionality--bean life cycle and wiring. Application context is basically a superset of Bean factory with the additional functionality typically needed in an enterprise context. Spring recommends that you use the application context in all scenarios, except when the additional few KBs of memory that the application context consumes are critical.*

Let's use an application context to create a Spring IoC container. We can have either a Java configuration or an XML configuration for an application context. Let's start with using a Java application configuration.

# Java configuration for the application context

The following example shows how to create a simple Java context configuration:

@Configuration   
 class SpringContext {   
 }

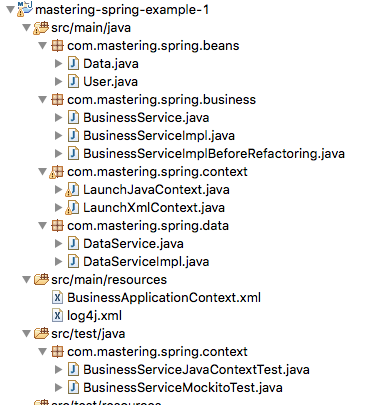
The key is the @Configuration annotation. This is what defines this as a Spring configuration.

One question remains; how does Spring IoC container know where to search for beans?

We need to tell the Spring IoC container the packages to search for by defining a component scan. Let's add a component scan to our earlier Java configuration definition:

@Configuration   
 @ComponentScan(basePackages = { "com.mastering.spring" })   
 class SpringContext {   
 }

We have defined a component scan for the com.mastering.spring package. It shows how all the classes we discussed until now are organized. All the classes we have defined until now are present in this package as follows:



**A quick review**

Let's take a moment and review all the things we have done until now to get this example working:

* We have defined a Spring configuration class SpringContext with the @Configuration annotation with a component scan for the com.mastering.spring package
* We have a couple of files (in the preceding package):
  + BusinessServiceImpl with the @Service annotation
  + DataServiceImpl with the @Repository annotation
* BusinessServiceImpl has the @Autowired annotation on the instance of DataService

When we launch up a Spring context, the following things will happen:

* It will scan the com.mastering.spring package and find the BusinessServiceImpl and DataServiceImpl beans.
* DataServiceImpl does not have any dependency. So, the bean for DataServiceImpl is created.
* BusinessServiceImpl has a dependency on DataService. DataServiceImpl is an implementation of the DataServiceinterface. So, it matches the autowiring criteria. So, a bean for BusinessServiceImpl is created and the bean created for DataServiceImpl is autowired to it through the setter.

# Launching the application context with Java configuration

The following program shows how to launch a Java context; we use the main method to launch the application context using AnnotationConfigApplicationContext:

public class LaunchJavaContext {   
 private static final User DUMMY\_USER = new User("dummy");   
 public static Logger logger =   
 Logger.getLogger(LaunchJavaContext.class);   
 public static void main(String[] args) {   
 ApplicationContext context = new   
 AnnotationConfigApplicationContext(   
 SpringContext.class);   
 BusinessService service =   
 context.getBean(BusinessService.class);   
 logger.debug(service.calculateSum(DUMMY\_USER));   
 }   
 }

The following lines of code create the application context. We want to create an application context based on the Java configuration. So, we use AnnotationConfigApplicationContext:

ApplicationContext context = new   
 AnnotationConfigApplicationContext(   
 SpringContext.class);

Once the context is launched, we will need to get the business service bean. We use the getBean method that passes the type of the bean (BusinessService.class) as an argument:

BusinessService service = context.getBean(BusinessService.class );

We are all set to launch the application context by running the LaunchJavaContext program.

# The console log

The following are some of the important statements from the log once the context is launched using LaunchJavaContext. Let's quickly review the log to get a deeper insight into what Spring is doing:

The first few lines show the component scan in action:

**Looking for matching resources in directory tree [/target/classes/com/mastering/spring]**  
  
**Identified candidate component class: file [/in28Minutes/Workspaces/SpringTutorial/mastering-spring-example-1/target/classes/com/mastering/spring/business/BusinessServiceImpl.class]**  
  
**Identified candidate component class: file [/in28Minutes/Workspaces/SpringTutorial/mastering-spring-example-1/target/classes/com/mastering/spring/data/DataServiceImpl.class]**  
  
**defining beans [\*\*\*\*\*\*OTHERS\*\*\*\*\*,businessServiceImpl,dataServiceImpl];**

Spring now starts to create the beans. It starts with businessServiceImpl, but it has an autowired dependency:

**Creating instance of bean 'businessServiceImpl'Registered injected element on class [com.mastering.spring.business.BusinessServiceImpl]: AutowiredFieldElement for private com.mastering.spring.data.DataService com.mastering.spring.business.BusinessServiceImpl.dataService**   
  
**Processing injected element of bean 'businessServiceImpl': AutowiredFieldElement for private com.mastering.spring.data.DataService com.mastering.spring.business.BusinessServiceImpl.dataService**

Spring moves on to dataServiceImpl and creates an instance for it:

**Creating instance of bean 'dataServiceImpl'**  
**Finished creating instance of bean 'dataServiceImpl'**

Spring autowires dataServiceImpl into businessServiceImpl:

**Autowiring by type from bean name 'businessServiceImpl' to bean named 'dataServiceImpl'**  
**Finished creating instance of bean 'businessServiceImpl'**

**The XML configuration for the application context**

In the previous example, we used a Spring Java configuration to launch an application context. Spring also supports XML configuration.

The following example shows how to launch an application context with an XML configuration. This will have two steps:

* Defining the XML Spring configuration
* Launching the application context with the XML configuration

# Defining the XML Spring configuration

The following example shows a typical XML Spring configuration. This configuration file is created in the src/main/resources directory with the name BusinessApplicationContext.xml:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>   
 <beans> <!-Namespace definitions removed-->   
 <context:component-scan base-package ="com.mastering.spring"/>   
 </beans>

The component scan is defined using context:component-scan.

# Launching an application context with the XML configuration

The following program shows how to launch an application context using the XML configuration. We use the main method to launch the application context using ClassPathXmlApplicationContext:

public class LaunchXmlContext {   
 private static final User DUMMY\_USER = new User("dummy");   
 public static Logger logger =   
 Logger.getLogger(LaunchJavaContext.class);   
 public static void main(String[] args) {   
 ApplicationContext context = new  
 ClassPathXmlApplicationContext(   
 "BusinessApplicationContext.xml");   
 BusinessService service =  
 context.getBean(BusinessService.class);   
 logger.debug(service.calculateSum(DUMMY\_USER));   
 }   
 }

The following lines of code create the application context. We want to create an application context based on the XML configuration. So, we use ClassPathXmlApplicationContext to create an application context: AnnotationConfigApplicationContext.

ApplicationContext context = new   
 ClassPathXmlApplicationContext (SpringContext.class);

Once the context is launched, we will need to get a reference to the business service bean. This is very similar to what we did with the Java configuration. We use the getBean method, passing the type of the bean (BusinessService.class) as an argument.

We can go ahead and run the LaunchXmlContext class. You will notice that we get output very similar to that we get when run the context with the Java configuration.

**Writing JUnit using the Spring context**

In the previous sections, we looked at how to launch a Spring context from the main method. Now let's shift our attention to launching a Spring context from a unit test.

We can use SpringJUnit4ClassRunner.class as a runner to launch a Spring context:

@RunWith(SpringJUnit4ClassRunner.class)

We would need to provide the location of the context configuration. We will use the XML configuration that we created earlier. Here's how you can declare this:

@ContextConfiguration(locations = {   
 "/BusinessApplicationContext.xml" })

We can autowire a bean from the context into the test using the @Autowired annotation. BusinessService is autowired by the type:

@Autowired   
 private BusinessService service;

As of now, DataServiceImpl, which is wired in, returns Arrays.asList(new Data(10), new Data(20)). BusinessServiceImplcalculates the sum 10+20 and returns 30. We will assert for 30 in the test method using assertEquals:

long sum = service.calculateSum(DUMMY\_USER);   
 assertEquals(30, sum);

*Why do we introduce unit testing so early in the book?  
  
Actually, we believe we are already late. Ideally, we would have loved to use***Test-driven development***(***TDD***) and write tests before code. In my experience, doing TDD leads to simple, maintainable, and testable code.*

Unit testing has a number of advantages:

* A safety net against future defects
* Defects are caught early
* Following TDD leads to a better design
* Well-written tests act as documentation of code and functionality--especially those written using the BDD Given-When-Then style

The first test we will write is not really a unit test. We will load up all the beans in this test. The next test, written using mocking, will be a real unit test, where the functionality being unit tested is the specific unit of code being written.

The complete list of the test is as follows; it has one test method:

@RunWith(SpringJUnit4ClassRunner.class)   
 @ContextConfiguration(locations = {  
 "/BusinessApplicationContext.xml" })   
 public class BusinessServiceJavaContextTest {   
 private static final User DUMMY\_USER = new User("dummy");   
 @Autowired   
 private BusinessService service;   
  
 @Test   
 public void testCalculateSum() {   
 long sum = service.calculateSum(DUMMY\_USER);   
 assertEquals(30, sum);   
 }   
 }

There is one problem with the **JUnit** that we wrote. It is not a true unit test. This test is using the real (almost) implementation of DataServiceImpl for the JUnit test. So, we are actually testing the functionality of both BusinessServiceImpl and DataServiceImpl. That's not unit testing.

The question now is this; how do we unit test BusinessServiceImpl without using a real implementation of DataService?

There are two options:

* Create a stub implementation of the data service, providing some dummy data in the src\test\java folder. Use a separate test context configuration to autowire the stub implementation instead of the real the DataServiceImplclass.
* Create a mock of DataService and autowire the mock into BusinessServiceImpl.

Creating a stub implementation would mean the creation of an additional class and an additional context. Stubs become more difficult to maintain, as we need more variations in data for the unit test.

In the next section, we will explore the second option of using a mock for unit testing. With the advances in mocking frameworks (especially **Mockito**) in the last few years, you will see that we would not even need to launch a Spring context to execute the unit test.

# Unit testing with mocks

Let's start with understanding what mocking is. Mocking is creating objects that simulate the behavior of real objects. In the previous example, in the unit test, we would want to simulate the behavior of DataService.

Unlike stubs, mocks can be dynamically created at runtime. We will use the most popular mocking framework, Mockito. To understand more about Mockito, we recommend the Mockito FAQ at <https://github.com/mockito/mockito/wiki/FAQ>.

We will want to create a mock for DataService. There are multiple approaches to creating mocks with Mockito. Let's use the simplest among them--annotations. We use the @Mock annotation to create a mock for DataService:

@Mock   
 private DataService dataService;

Once we create the mock, we will need to inject it into the class under test, BusinessServiceImpl. We do that using the @InjectMocks annotation:

@InjectMocks   
 private BusinessService service =   
 new BusinessServiceImpl();

In the test method, we will need to stub the mock service to provide the data that we want it to provide. There are multiple approaches. We will use the BDD style methods provided by Mockito to mock the retrieveData method:

BDDMockito.given(dataService.retrieveData(  
 Matchers.any(User.class)))   
 .willReturn(Arrays.asList(new Data(10),   
 new Data(15), new Data(25)));

What we are defining in the preceding code is called stubbing. As with anything with Mockito, this is extremely readable. When the retrieveData method is called on the dataService mock with any object of type User, it returns a list of three items with values specified.

When we use Mockito annotations, we would need to use a specific JUnit runner, that is, MockitoJunitRunner. MockitoJunitRunner helps in keeping the test code clean and provides clear debugging information in case of test failures. MockitoJunitRunner initializes the beans annotated with @Mock annotation and also validates the usage of framework after execution of each test method.

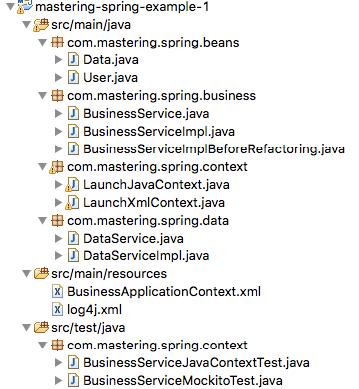
@RunWith(MockitoJUnitRunner.class)

The complete list of the test is as follows. It has one test method:

@RunWith(MockitoJUnitRunner.class)   
 public class BusinessServiceMockitoTest {   
 private static final User DUMMY\_USER = new User("dummy");  
 @Mock   
 private DataService dataService;   
 @InjectMocks   
 private BusinessService service =   
 new BusinessServiceImpl();   
 @Test   
 public void testCalculateSum() {   
 BDDMockito.given(dataService.retrieveData(   
 Matchers.any(User.class)))   
 .willReturn(   
 Arrays.asList(new Data(10),   
 new Data(15), new Data(25)));   
 long sum = service.calculateSum(DUMMY\_USER);   
 assertEquals(10 + 15 + 25, sum);   
 }   
 }

**Container managed beans**

Instead of a class creating its own dependencies, in the earlier example, we looked at how the Spring IoC container can take over the responsibility of managing beans and their dependencies. The beans that are managed by the container are called **Container Managed Beans**.



Delegating the creation and management of beans to the container has many advantages. Some of them are listed as follows:

* Since classes are not responsible for creating dependencies, they are loosely coupled and testable. This leads to good design and fewer defects.
* Since the container manages the beans, a few hooks around the beans can be introduced in a more generic way. Cross-cutting concerns, such as logging, caching, transaction management, and exception handling can be woven around these beans using **Aspect-Oriented Programming** (**AOP**). This leads to more maintainable code.

**Dependency injection types**

In the previous example, we used a setter method to wire in the dependency. There are two types of dependency injections that are used frequently:

* The setter injection
* The constructor injection

# The setter injection

The setter injection is used to inject the dependencies through setter methods. In the following example, the instance of DataService uses the setter injection:

public class BusinessServiceImpl {   
 private DataService dataService;   
 @Autowired   
 public void setDataService(DataService dataService) {   
 this.dataService = dataService;   
 }   
 }

Actually, in order to use the setter injection, you do not even need to declare a setter method. If you specify @Autowired on the variable, Spring automatically uses the setter injection. So, the following code is all that you need for the setter injection for DataService:

public class BusinessServiceImpl {   
 @Autowired   
 private DataService dataService;   
 }

# The constructor injection

The constructor injection, on the other hand, uses a constructor to inject dependencies. The following code shows how to use a constructor for injecting in DataService:

public class BusinessServiceImpl {   
 private DataService dataService;   
 @Autowired   
 public BusinessServiceImpl(DataService dataService) {   
 super();   
 this.dataService = dataService;   
 }   
 }

When you run the code with the preceding implementation of BusinessServiceImpl, you will see this statement in the log, asserting that autowiring took place using the constructor:

Autowiring by type from bean name 'businessServiceImpl' via   
 constructor to bean named 'dataServiceImpl'

# Constructor versus setter injection

Originally, in XML-based application contexts, we used the constructor injection with mandatory dependencies and the setter injection with nonmandatory dependencies.

However, an important thing to note is that when we use @Autowired on a field or a method, the dependency is required by default. If no candidates are available for an @Autowired field, autowiring fails and throws an exception. So, the choice is not so clear anymore with Java application contexts.

Using the setter injection results in the state of the object changing during the creation. For fans of immutable objects, the constructor injection might be the way to go. Using the setter injection might sometimes hide the fact that a class has a lot of dependencies. Using the constructor injection makes it obvious, since the size of the constructor increases.

# Spring bean scopes

Spring beans can be created with multiple scopes. The default scope is a singleton.

Since there is only one instance of a singleton bean, it cannot contain any data that is specific to a request.

The scope can be provided with the @Scope annotation on any spring bean:

@Service   
 @Scope("singleton")   
 public class BusinessServiceImpl implements BusinessService

The following table shows the different types of scopes available for beans:

|  |  |
| --- | --- |
| **Scope** | **Use** |
| Singleton | By default, all beans are of the scope singleton. Only one instance of such beans is used per instance of the Spring IoC container. Even if there are multiple references to a bean, it is created only once per container. The single instance is cached and used for all subsequent requests using this bean. It is important to specify that the Spring singleton scope is one object per one Spring container. If you have multiple spring containers in a single JVM, then there can be multiple instances of the same bean. So, the Spring singleton scope is a little different from the typical definition of a singleton. |
| Prototype | A new instance is created every time a bean is requested from the Spring container. If a bean contains a state, it is recommended that you use the prototype scope for it. |
| request | Available only in Spring web contexts. A new instance of bean is created for every HTTP request. The bean is discarded as soon as the request processing is done. Ideal for beans that hold data specific to a single request. |
| session | Available only in Spring web contexts. A new instance of bean is created for every HTTP session. Ideal for data specific to a single user, such as user permissions in a web application. |
| application | Available only in Spring web contexts. One instance of bean per web application. Ideal for things such as application configuration for a specific environment. |

**Java versus XML configuration**

With the advent of annotations in Java 5, there is widespread use of Java configuration for Spring based applications. What is the right choice to make if you have to choose between a Java-based configuration as opposed to an XML-based configuration?

Spring provides equally good support for Java and XML-based configuration. So, it's left to the programmer and their team to make the choice. Whichever choice is made, it is important to have consistency across teams and projects. Here are some things you might need to consider when making a choice:

* Annotations lead to shorter and simpler bean definitions.
* Annotations are closer to the code they are applicable on than the XML-based configuration.
* Classes using annotations are no longer simple POJOs because they are using framework-specific annotations.
* Autowiring problems when using annotations might be difficult to solve because the wiring is no longer centralized and is not explicitly declared.
* There might be advantages of more flexible wiring using Spring context XML if it is packaged outside the application packaging--WAR or EAR. This will enable us to have different setup for integration tests, for example.

**The @Autowired annotation in depth**

When @Autowired is used on a dependency, the application context searches for a matching dependency. By default, all dependencies that are autowired are required.

Possible results are as follows:

* **One match is found**: This is the dependency you are looking for
* **More than one match is found**: Autowiring fails
* **No match is found**: Autowiring fails

Cases where more than one candidate is found can be resolved in two ways:

* Use the @Primary annotation to mark one of the candidates as the one to be used
* Use @Qualifier to further qualify autowiring

# The @Primary annotation

When the @Primary annotation is used on a bean, it becomes the primary one to be used when there is more than one candidate available to autowire a specific dependency.

In the case of the following example , there are two sorting algorithms available: QuickSort and MergeSort. If the component scan finds both of them, QuickSort is used to wire any dependencies on SortingAlgorithm because of the @Primary annotation:

interface SortingAlgorithm {   
 }   
 @Component   
 class MergeSort implements SortingAlgorithm {   
 // Class code here   
 }   
 @Component   
 @Primary   
 class QuickSort implements SortingAlgorithm {   
 // Class code here   
 }

# The @Qualifier annotation

The @Qualifier annotation can be used to give a reference to a Spring bean. The reference can be used to qualify the dependency that needs to be autowired.

In the case of the following example, there are two sorting algorithms available: QuickSort and MergeSort. But since @Qualifier("mergesort") is used in the SomeService class, MergeSort, which also has a mergesort qualifier defined on it, becomes the candidate dependency selected for autowiring:

@Component   
 @Qualifier("mergesort")   
 class MergeSort implements SortingAlgorithm {   
 // Class code here   
 }   
 @Component   
 class QuickSort implements SortingAlgorithm {   
 // Class code here   
 }   
 @Component   
 class SomeService {   
 @Autowired   
 @Qualifier("mergesort")   
 SortingAlgorithm algorithm;   
 }

**Other important Spring annotations**

Spring provides a great deal of flexibility in defining beans and managing the life cycle of a bean. There are a few other important Spring annotations that we will discuss in the table, as follows:

|  |  |
| --- | --- |
| **Annotations** | **Use** |
| @ScopedProxy | Sometimes, we will need to inject a request or a session-scoped bean into a singleton-scoped bean. In such situations, the @ScopedProxy annotation provides a smart proxy to be injected into singleton-scoped beans. |
| @Component, @Service, @Controller, @Repository | @Component is the most generic way of defining a Spring bean. Other annotations have more specific contexts associated with them.   * @Service is used in the business service layer * @Repository is used in the **data access object** (**DAO**) * @Controller is used in presentation components |
| @PostConstruct | On any spring bean, a post construct method can be provided using the @PostConstructannotation. This method is called once the bean is fully initialized with dependencies. This will be invoked only once during a bean lifecycle. |
| @PreDestroy | On any spring bean, a predestroy method can be provided using the @PreDestroy annotation. This method is called just before a bean is removed from the container. This can be used to release any resources that are held by the bean. |

# Exploring Contexts and dependency injection

CDI is Java EE's attempt at bringing DI into Java EE. While not as fully-fledged as Spring, CDI aims to standardize the basics of how DI is done. Spring supports the standard annotations defined in JSR-330. For the most part, these annotations are treated the same way as Spring annotations.

Before we can use CDI, we will need to ensure that we have dependencies for CDI jars included. Here's the code snippet:

<dependency>   
 <groupId>javax.inject</groupId>   
 <artifactId>javax.inject</artifactId>   
 <version>1</version>   
 </dependency>

In this table, let's compare the CDI annotations with the annotations provided by Spring Framework. It should be noted that @Value, @Required, and @Lazy Spring annotations have no equivalent CDI annotations.

|  |  |
| --- | --- |
| **CDI annotation** | **Comparison with Spring annotations** |
| @Inject | Similar to @Autowired. One insignificant difference is the absence of the required attribute on @Inject. |
| @Named | @Named is similar to @Component. Identifies named components. In addition, @Named can be used to qualify the bean with a name similar to the @Qualifier Spring annotation. This is useful in situations when multiple candidates are available for the autowiring of one dependency. |
| @Singleton | Similar to the Spring annotation @Scope("singleton"). |
| @Qualifier | Similar to a similarly named annotation in Spring--@Qualifier |

# An example of CDI

When we use CDI, this is what the annotations on the different classes would look like. There is no change in how we create and launch the Spring application context.

CDI marks no differentiation between @Repository, @Controller, @Service, and @Component. We use @Named instead of all the preceding annotations.

In the example, we use @Named for DataServiceImpl and BusinessServiceImpl. We use @Inject to inject dataService into BusinessServiceImpl (instead of @Autowired):

@Named //Instead of @Repository   
 public class DataServiceImpl implements DataService   
 @Named //Instead of @Service   
 public class BusinessServiceImpl {   
 @Inject //Instead of @Autowired   
 private DataService dataService;

# Summary

Dependency injection (or IoC) is the key feature of Spring. It makes code loosely coupled and testable. Understanding DI is the key to making the best use of Spring Framework.

In this chapter, we took a deep look at DI and the options Spring Framework provides. We also looked at examples of writing testable code and wrote a couple of unit tests.

In the next chapter, we will shift our attention toward Spring MVC, the most popular Java web MVC framework. We will explore how Spring MVC makes the development of web applications easier.