9
Inversion of Control in Spring – Using Annotation

In this chapter, we will configure Spring beans and the Dependency Injection using annotations. Spring provides support for annotation-based container configuration. We will go through bean management using stereotypical annotations and bean scope using annotations. We will then take a look at an annotation called `@Required`, which allows us to specify which dependencies are actually required. We will also see annotation-based dependency injections and life cycle annotations. We will use the autowired annotation to wire up dependencies in the same way as we did using XML in the previous chapter. You will then learn how to add dependencies by type and by name. We will also use qualifier to narrow down Dependency Injections. We will also understand how to perform Java-based configuration in Spring. We will then try to listen to and publish events in Spring. We will also see how to reference beans using **Spring Expression Language** (**SpEL**), invoke methods using SpEL, and use operators with SpEL. We will then discuss regular expressions using SpEL. Spring provides text message and internationalization, which we will learn to implement in our application.

Here's a list of the topics covered in this chapter:

- Container configuration using annotations
- Java-based configuration in Spring
- Event handling in Spring
- Text message and internationalization

Container configuration using annotation

Container configuration using Spring XML sometimes raises the possibility of delays in application development and maintenance due to size and complexity. To solve this issue, the Spring Framework supports container configuration using annotations without the need of a separate XML definition.

From Spring 2.5, a complete set of configuration annotations has been introduced that is related to Spring configuration XML such as Spring bean definition or Dependency Injection. In addition, Spring 3 has also provided JSR-330 (Dependency Injection for Java) annotation for DI as well as Spring-specialized annotation.

In this chapter, we have divided annotations into bean management, annotation-based dependency injections, and the bean life cycle based on its usage. Let's understand how to use each annotation.

For a Spring IoC container to recognize annotations, we need to add the following definition to the `beans.xml` configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-
beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-
context-3.2.xsd">

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

We can start annotating our code once `<context:annotation-config/>` is configured. Spring can autoscan, detect, and instantiate beans from the project package, and thus saves us the tedious declaration of beans or components in XML. It activates annotations in beans, which is already registered in the `ApplicationContext`.

XML versus annotation

Let's consider an example. We have a code snippet for the `Employee` class and the configuration file that uses XML and an annotation-based configuration. For an annotation-based configuration, we have the `@Service` annotation, which registers the bean class to the container. And we eliminate the definition of the bean in the configuration file. However, in the XML-based configuration without annotations, we have added a definition of the bean to the configuration file.

- **Annotation-based configuration**: The following table shows this configuration with annotations:

File	Syntax

`Employee.java`	`package org.packt.spring.chapter3.annotation;`
	`import org.springframework.stereotype.Service;`
	
	`@Service`
	`public class Employee {`
	`  public String toString() {`
	`  return "Annotation Based Configuration";`
	`  }`
	`}`

`beans.xml`	`<context:annotation-config />`
	
	`<context:component-scan base-package="org.packt.spring.chapter3.annotation"/>`

- **XML-based configuration**: The following table shows this configuration without annotations:

File	Syntax

`Employee.java`	`package org.packt.spring.chapter3.xml;`
	
	`public class Employee {`
	`  public String toString() {`
	`  return "XML Based Configuration";`
	`  }`
	`}`

File	Syntax
`beans.xml`	`<bean id="employeeBean" class="org.packt.Spring.chapter3.xml.Employee ">` `</bean>`

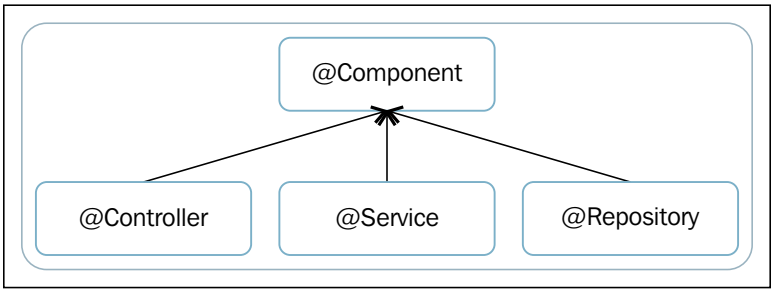
For large applications, there would be more XML tags and that increases the application's complexity. To eliminate this, we can use annotation in the Java class and remove a few XML tags from the configuration file.

Bean management

Beans that need to be managed by Spring can be defined using the stereotype annotation. Classes that are annotated with one of these stereotype annotations will automatically be registered in the Spring application context if `<context:component-scan>` is in the Spring XML configuration.

Stereotype annotation

`@Component` is a parent stereotype annotation and usually defines all beans. Components are divided on the basis of layers by the Spring Framework and it is recommended to use annotation as shown in the following figure:



There are four types of autocomponent scan annotations. They are as follows:

Annotation	Use	Description
`@Component`	Type	This is a generic stereotype annotation that can be used for the Spring-manages component
`@Repository`	Type	This annotation is used to define a component as a repository, and this class will handle data access logic

Annotation	Use	Description
`@Service`	Type	This annotation is used to define a component as a service, and this class will handle business logic
`@Controller`	Type	This annotation is used to define a component as a Spring MVC controller

@Component

The `@Component` annotation is a generic stereotype annotation that defines a class as a bean.

Employee.java

The following code snippet shows the `Employee` class:

```
import org.springframework.stereotype.Component;

@Component
public class Employee {
//…
}
```

Here, the `@Component` annotation defines the `Employee` class as a Spring bean. Whenever Spring detects a class annotated with `@Component`, it does the equivalent of defining the class as a bean in the configuration XML file and creating a name for this bean, which is actually the name of the class `Employee`.

@Repository

We must annotate all your repository classes with the `@Repository` annotation, which is a marker for a class. A repository class serves as a **Data Access Object** (**DAO**) in the persistence layer of the application. All your database access logic should go in these DAO classes, as shown here:

```
import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDAO
{
   // database access logic
}
```

Here, the `@Repository` annotation defines the `EmployeeDAO` class as a repository class that contains database access logic.

@Service

We must annotate all your service classes with the `@Service` annotation, which is a class-level annotation. All your business logic should go in the service classes:

```
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {
    // business logic
}
```

In the preceding example, the `EmployeeService` class is annotated with the Stereotype annotation called `@Service`, which represents a component of the service layer.

@Controller

The `@Controller` annotation indicates that the annotated class is a Spring component of type "controller". It is a class-level annotation indicating that an annotated class serves the role of a controller in Spring MVC:

```
import org.springframework.stereotype.Controller;

@Controller
public class EmployeeController {
    //...
}
```

In the preceding example, the `EmployeeController` class is annotated with `@Controller`, which represents a controller component in Spring MVC.

Naming detected components

When naming components that Spring detects, it lowercases the first character of the nonqualified class name. For example, the `EmployeeService` class will be named `employeeService` and we can receive this component with the name `employeeService`.

The `EmployeeService.java` file is shown in the following code snippet:

```
@Service
public class EmployeeService {

       // . . .
}
```

The `PayrollSystem.java` file is shown in the following code snippet:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

// get 'employeeService' bean from application context
EmployeeService employeeService = (EmployeeService)
context.getBean("employeeService");
```

Custom components name

We can also explicitly define a custom component's name by specifying the desired name in the stereotype annotation's value.

The `EmployeeService.java` file is shown in the following code snippet:

```
@Service("employeeServiceBean")
public class EmployeeService {
       // . . .
}
```

Here, we have specified the `EmployeeService` component's name as `employeeServiceBean` in the `@Service` annotation's value. Now, we can retrieve this component with the name `employeeServiceBean`.

The P`ayrollSystem.java` file is shown in the following code snippet:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

EmployeeService employeeService = (EmployeeService)
context.getBean("employeeServiceBean");
```

Auto detecting classes and registering bean definitions

Spring can automatically detect stereotyped classes and register their corresponding bean definitions with the `ApplicationContext`. To auto detect stereotyped classes, we need to define `<context :component-scan/>` in the configuration XML file. The Spring container can identify classes with the stereotype annotation and manage beans without separately defining them in the configuration XML file.

The `beans.xml` file is shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-
beans.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-
context-3.2.xsd">

    <context:component-scan/>

</beans>
```

If you add this definition in a Spring configuration XML file, then the container will scan the classes that exist in the classpath, detects stereotyped classes, and automatically register them to the Spring Container. It scans packages to find and register beans within the `ApplicationContext`. The `base-package` property can be used to define the search target package as follows:

```
<context:component-scan base-package="..."/>
```

Let's consider an example to enable the Spring autoscanning feature.
The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.annotationbasedconfig;

import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @Override
   public String toString() {
        return "Annotation Based Configuration";
   }
}
```

Here, the `EmployeeService` class is annotated with `@Service` to indicate that this class is an autoscan component.

The `PayrollSystem.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.annotationbasedconfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {

          ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

          EmployeeService employeeService =
(EmployeeServiceImp)context.getBean("employeeService");
}
```

In the preceding code, we have used the name defined as property `employeeService` to find the relevant bean.

The `beans.xml` file is shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-
beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-
context-3.2.xsd">

     <context:component-scan base-
package="org.packt.springframework.chapter3.
annotationbasedconfig"/>

</beans>
```

We have included this definition in `beans.xml` to autodetect the class and register the corresponding bean, where the `base-package` element is a package for the classes. We have added the package `org.packt.springframework.chapter3.annotationbasedconfig` as a value of the `base-package` attribute in `beans.xml`. Spring will automatically scan and detect stereotyped classes inside this package and register that bean definition to the container. Since the `EmployeeService` class is annotated with `@Service` and resides inside this package, the `employeeService` bean definition will be registered in the application context and we can get this bean from `ApplicationContext` using the `getBean` method.

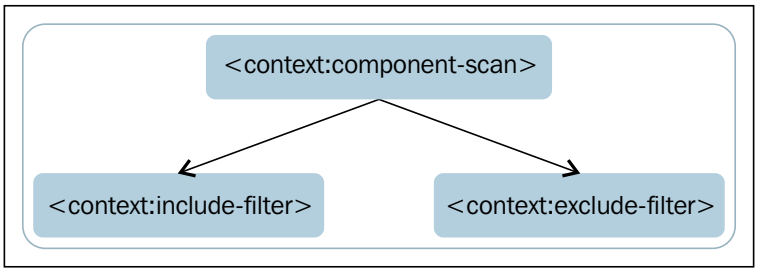
Using filters to customize scanning

Stereotyped classes are automatically detected within the search range using various properties of `<context:component-scan base-package="...">`. It is also possible to filter classes registered as beans in the container.

The `<context:component-scan>` can have subelements such as:

- `<context:include-filter>`
- `<context:exclude-filter>`

The following diagram illustrates these:



Each filter element needs the `type` and `expression` attributes to define the component-scanning strategy. The filter type can be any one of these:

Filter type	Expression attribute
`annotation`	The expression attribute takes the annotation to be scanned
`assignable`	The expression attribute takes the assignable to be scanned
`regex`	The expression attribute takes the regular expression to be matched by the target components' class names
`aspectj`	The expression attribute takes the AspectJ-type expression to be matched by the target components' class names.

Filter component – include (<context:include-filter>)

When the include filter is used with type `regex`, the Spring Container scans and registers the components' name that matches the defined regular expression. Even those classes that are not stereotyped (that is, not annotated with annotations such as `@Component`, `@Service`, `@Repository`, or `@Controller`) are scanned and registered.

The following example code snippet is trying to load the `EmployeeDAO` and `EmployeeService` classes that match the regular expression specified in the expression attribute of `include-filter`. The `EmployeeDAO` and `EmployeeService` classes have not been annotated with the stereotype annotation.

The `EmployeeDAO.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.includefilter;

public class EmployeeDAO {

    public String toString() {
         return "Implements data access logic in DAO class";
   }
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.includefilter;

public class EmployeeService {

   public String toString() {
          return "Implements business logic in Service class";
   }
}
```

The `beans.xml` file is shown in the following code snippet:

```
   <context:component-scan base-
package="org.packt.springframework.chapter3.includefilter">

         <context:include-filter type="regex"
expression="org.packt.springframework.chapter3.includefilter.*DAO.
*" />
          <context:include-filter type="regex" expression="org.packt.
springframework.chapter3.includefilter.*
Service.*" />

   </context:component-scan>
```

Here, in this XML, all filenames contain the DAO or Service (`*DAO.*` or `*Services.*`) syntax, which will be detected and registered in the Spring Container, as we have provided type as `regex` along with the corresponding expressions.

The `PayrollSystem.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.includefilter;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {

         ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

         EmployeeDAO employeeDao = (EmployeeDAO) context.
getBean("employeeDAO");
         System.out.println("Employee Dao: " + employeeDao);

         EmployeeService employeeService = (EmployeeService) context.
getBean("employeeService");
         System.out.println("Employee Service: " +
employeeService);
   }
}
```

Here, we have called the `getBean` method to load the `EmployeeDAO` and `EmployeeService` beans from the `ApplicationContext`. When we run the preceding main method, the output will be generated as follows, which indicates that the filter has scanned and registered the bean definition in the container.

The expected output will be as follows:

```
Employee Dao: Implements data access logic in DAO class
Employee Service: Implements business logic in Service class
```

Filter component – exclude (<context:exclude-filter>)

When the exclude filter is used with type regex, the Spring container scans and prevents Spring from registering component names that match the defined regular expression.

The `beans.xml` file is shown in the following code snippet:

```
   <context:component-scan base-
package="org.packt.springframework.chapter3.excludefilter">

          <context:exclude-filter type="regex"
expression="org.packt.springframework.chapter3.
excludefilter.*DAO.*" />

   </context:component-scan>
```

Here, in the configuration file, the entire matched component with the regular expression will be detected and prevent Spring from registering the component names in the container. So, when we call the `getBean("employeeDAO")` method, it will result in the following:

```
org.springframework.beans.factory.NoSuchBeanDefinitionException:
No bean named 'employeeDAO' is defined
```

Scope definition – @Scope

Singleton is the most common scope for autodetected components by default. We can use other scopes after providing the name of scope within a new `@Scope` annotation. Let's say for prototype scope, use `@Scope("prototype")`.

The `EmployeeService.java` file is shown in the following code snippet:

```
import org.springframework.context.annotation.Scope;

@Scope("prototype")
public class EmployeeService {
   //business logic
}
```

Dependency checking with the @Required annotation

Since Spring 2.5, annotation-based Spring configuration has been an alternative to XML-based Spring configurations. XML injection is performed after annotation injection into Spring. For properties wired through both the XML and annotation approach, the XML-based configuration will override the annotation-based configuration.

The `@Required` annotation enforces the required properties of the bean. Before the bean is used, the setter method annotated with the `@Required` annotation will be called. The `@Required` annotation indicates that the affected bean property must be populated in the configuration file during Spring configuration, otherwise the Spring Container throws an exception, `BeanInitializationException`.

Let's take an example of the `EmployeeDAO` class to understand the `@Required` annotation. We have imported a package named `org.springframework.beans.factory.annotation.Required` to annotate the method `setDatabaseUrl()` with the `@Required` annotation, which indicates that the affected bean property (`databaseUrl`) must be populated at configuration time.

Check `EmployeeDAO.java` for the following code snippet:

```
package org.packt.spring...chapter3.requiredannotation;

import org.springframework.beans.factory.annotation.Required;

public class EmployeeDAO {

   String databaseUrl;

   @Required
   public void setDatabaseUrl(String databaseUrl) {
          this.databaseUrl = databaseUrl;
   }
}
```

The affected bean property must be populated either through an explicit property value in the bean definition or through autowiring; if not populated at configuration time, this will throw an exception and avoid `NullPointerException` or the like later on.

The `beans.xml` file is shown in the following code snippet:

```
<context:annotation-config />

<bean id="employeeDao"
class="org.packt.Spring.chapter3.requiredannotation.EmployeeDAO">

          <!-- try without passing value for property and check
result -->
          <!-- property name="databaseUrl" value="..."></property
-->

</bean>
```

Here, in `beans.xml`, we are not passing a value for the property `databaseUrl`. When we try to use the bean `employeeDao`, the following error message will be throw, because the `databaseUrl` property has not been set:

```
org.springframework.beans.factory.BeanInitializationException:
   Property 'databaseUrl' is required for bean 'employeeDao'
```

Annotation-based Dependency Injection

Since Spring 2.5, annotations have been introduced to automatically wire bean properties. Annotation-based autowiring is not very different from XML-based autowiring. Annotation wiring in the Spring Container is not turned on by default; we need to enable it in the Spring configuration before using annotation-based autowiring. Spring has added annotation-based configuration support for Dependency Injection:

Annotation	Description	Target
`@Autowired`	This is used to define dependencies provided in the Spring framework	Constructor, field, or method
`@Resource`	This can be considered to be related to a JNDI resource (datasource, Java messaging service destination, or environment entry)	Field or method
`@Inject`	Constructor, field, or method	Constructor, field, or method

@Autowired

The `@Autowired` annotation is used to define the dependency provided in the Spring Framework. It can be used to autowire beans on the setter method or on a constructor, property, or method. We can autowire one or more parameters to a constructor or method. More than one constructor can be set as autowired even though only one constructor can be marked as required. If more than one constructor has been marked as autowired, then Spring will use the constructor that has the most arguments matched based on the beans in the Spring Container. When `@Autowired` is unable to find at least one match, an exception will be thrown, `BeanCreationException`.

@Autowired on setter methods – byType

The `@Autowired` annotation can be used on setter methods to get rid of the `<property>` element in a Spring XML configuration file. When Spring finds a setter method annotated with `@Autowired`, it autowires the bean `byType`. This bean allows a property to be autowired if exactly one bean of the property type exists in the container.

Let's consider an example. We have an `EmployeeService` class annotated with `@Service` and an `EmployeeDao` class annotated with `@Repository`. The `EmployeeService` class has the property `employeeDao` and the setter and getter methods. We have annotated the `setEmployeeDao()` method of `EmployeeService` with `@Autowired` from `org.springframework.beans.factory.annotation.Autowired`. So we don't need to specify the property in XML for this bean.

In the `PayrollSystem` class, we are instantiating the bean using the `getBean()` method. We have done the annotation-based configuration in `beans.xml`.

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onsetter;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   private EmployeeDao employeeDao;

   @Autowired
   public void setEmployeeDao(EmployeeDao employeeDao) {
          this.employeeDao = employeeDao;
   }
   public EmployeeDao getEmployeeDao() {
         return employeeDao;
   }
}
```

The `EmployeeDao.java` file is shown in the following code snippet:

```
package org.packt.Spring.chapter3.annotation...onsetter;

import org.springframework.stereotype.Repository;

@Repository
public class EmployeeDao {

   @Override
   public String toString() {
         return "Data Access logic will reside in DAO";
   }
}
```

The `beans.xml` file is shown in the following code snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-
beans.xsd
          http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-
context-3.2.xsd">

   <context:annotation-config />

   <context:component-scan base-
package="org.packt.spring.chapter3.annotation...onsetter" />

</beans>
```

The `PayrollSystem.java` file is shown in the following code snippet:

```
package org.packt.Spring.chapter3.annotation...onsetter;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {

          ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

          EmployeeService employeeService =
(EmployeeService)context.getBean("employeeService");
          System.out.println(employeeService.getEmployeeDao());
   }
}
```

The output is as follows:

```
Data Access logic will reside in DAO
```

We can easily observe that there is no relation between the two beans `employeeService` and `employeeDao`. The `@Autowired` annotation on the setter method `setEmployeeDao()` in the `EmployeeService` class will automatically wire the `employeeDao` bean on the basis of type.

Ambiguity – BeanCreationException

If more than one bean match is found in `ApplicationContext` when matching by type, and they aren't the `Collection` class or an array, an exception will be thrown, `BeanCreationException`.

Let's understand this with an example. We have an `Employee` class and more definitions in `beans.xml` results in more than one bean in `ApplicationContext`. The `EmployeeService` class will have the `setEmployee()` method annotated with `@Autowired`.

The `Employee.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onsetter;

public class Employee {

   @Override
   public String toString() {
         return "From employee class";
   }
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onsetter;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   private Employee employee;
   @Autowired
   public void setEmployeeDao(EmployeeDao employeeDao) {
          this.employee = employee;
   }
}
```

The `beans.xml` file is shown in the following code snippet:

```
    <context:annotation-config />
    <context:component-scan base-
package="org.packt.Spring.chapter3.Annotation.onsetter" />

    <bean id="employeeA"
class="org.packt.Spring.chapter3.Annotation.onsetter.Employee">
    </bean>


    <bean id="employeeB"
class="org.packt.spring.chapter3.annotation.onsetter.Employee">
    </bean>
```

The `PayrollSystem.java` file is shown in the following code snippet:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
```

The output is as follows:

```
Error creating bean with name 'employeeService': Injection of
autowired dependencies failed; nested exception is
org.springframework.beans.factory.BeanCreationException
```

Since there is more than one bean definition available in `ApplicationContext`, Spring cannot know which beans need to be injected.

Read on to see the solution to this problem.

The `beans.xml` file is shown in the following code snippet:

```
<bean id="employeeA"
class="org.packt.Spring.chapter3.Annotation.onsetter.Employee">
    </bean>
    <bean id="employeeB"
class="org.packt.Spring.chapter3.Annotation.onsetter.Employee">
    </bean>

    <bean id="employeeC"
class="org.packt.Spring.chapter3.Annotation.onsetter.Employee">
    </bean>

    <bean id="employee"
class="org.packt.Spring.chapter3.Annotation.onsetter.Employee">
    </bean>
```

In the preceding `beans.xml` file, we have multiple definitions for the bean `Employee` with different IDs: `employeeA`, `employeeB`, `employeeC`, and `employee`. When one and only one of these IDs matches with the property name in the `EmployeeService` class (in this case, only `id=employee` matches the property name), the dependency will be successfully injected.

The `PayrollSystem.java` file is shown in the following code snippet:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
```

The output is as follows:

```
From employee class
```

So, the `@Autowired` annotation first looks for type, and if it finds that there is only one bean of that type, even though the name is different, it will autowire it as there is only one bean. If there are multiple beans of the same type, then Spring will look to see whether there are any beans whose names match the name of the member variable (in our case, `id=employee` matches employee). If any do, it will inject the dependency of that matched bean.

But let's assume that we are restricted to changing the name of a bean defined in XML, maybe because it was referred by other beans in XML with that name. Another approach to solve this ambiguity is to use `@Qualifier`. We will see that in the latter part of this chapter.

@Autowired on properties

When the `@Autowired` annotation is used on properties or fields, values are automatically assigned to the corresponding attributes.

Let's take an example of the classes `EmployeeService` and `Employee`. The `EmployeeService` class has the property `employee` and a method `getEmployee()`, and this class has been annotated with `@Service`. The property `employee` of the `EmployeeService` class has been annotated with `@Autowired`.

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onproperty;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
```

```
public class EmployeeService {

   @Autowired
   private Employee employee;

   public Employee getEmployee() {
         return employee;
   }
}
```

The `Employee.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onproperty;

public class Employee {

   @Override
   public String toString() {
          return "From employee class";
   }
}
```

The `beans.xml` file is shown in the following code snippet:

```
   <context:annotation-config />
   <context:component-scan base-package="org.packt.Spring.chapter3.
Annotation.onproperty" />

   <bean id="employee"

   class="org.packt.spring.chapter3.
annotation.onproperty.Employee">
   </bean>
```

Here, we see that the `@Autowired` annotation is used for the `type` property without any setter method.

@Autowired on constructors

The `@Autowired` annotation on a constructor specifies that that constructor should be autowired when creating the Spring bean, even if no `<constructor-arg>` elements are used in the configuration XML file while configuring the bean in it. Let's consider an example. We have an `Employee` class annotated with `@Component`, an `EmployeeService` class that has an `Employee` property, and a constructor that is annotated with the `@Autowired` annotation and takes an argument of type `Employee`.

The `Employee.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onconstruction;

import org.springframework.stereotype.Component;

@Component
public class Employee {

   @Override
   public String toString() {
         return "From employee class";
   }
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...onconstruction;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   private Employee employee;

   @Autowired
   EmployeeService(Employee employee) {
          System.out.println("Inside Employee Service
Constructor");
          this.employee = employee;
   }
}
```

Here, the `EmployeeService` class contains a constructor that is annotated with `@Autowired`.

The `beans.xml` file is shown in the following code snippet:

```
<context:annotation-config />

<context:component-scan base-
package="org.packt.spring.chapter3.annotation...onconstruction" />
```

The `PayrollSystem.java` file is shown in the following code snippet:

```
EmployeeService employeeService = (EmployeeService)context.
getBean("employeeService");
System.out.println(employeeService.getEmployee());
```

Let's run the application. If everything goes fine with your application, the code will print the following message:

```
Inside Employee Service Constructor
From employee class
```

Optional autowiring – @Autowired(required=false)

By default, `@Autowired` requires that the thing it annotates is wired, otherwise autowired fails with the exception `NoSuchBeanDefinitionException`. Many times, it's better to have Spring fail early when autowiring goes bad instead of later with `NullPointerException`.

Sometimes, we need to make the property being wired an optional property that accepts a `null` value. In that case, by setting the `@Autowired` annotation's required attribute to `false`, we configure an optional autowiring.

Let's consider an example to understand this in more detail. We have the `Employee` class annotated with the `@Component` annotation and an `Address` class. We have the `EmployeeService` class annotated with the `@Service` annotation. This class has two properties: `Employee` and `Address`.

The `Employee.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...false;

import org.springframework.stereotype.Component;

@Component
public class Employee {

   @Override
   public String toString() {
         return "From employee class";
   }
}
```

The `Address.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...false;

public class Address {

   @Override
   public String toString() {
         return "From to address";
   }
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation.false;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @Autowired
   private Employee employee;

   private Address address;

   public Address getAddress() {
          return address;
   }

   @Autowired(required = false)
   public void setAddress(Address address) {
          this.address = address;
   }

   public Employee getEmployee() {
         return employee;
   }
}
```

Here, in the `EmployeeService` class, we have the annotated property `Employee` with `@Autowired`. Also, the `setAddress()` method has been annotated with `@autowired(required=false)`, which will prevent runtime exceptions when no bean definitions are found while autowiring.

The `PayrollSystem.java` file is shown in the following code snippet:

```
EmployeeService employeeService = (EmployeeService)
context.getBean("employeeService");

System.out.println(employeeService.getEmployee());

System.out.println(employeeService.getAddress());
```

The `beans.xml` file is shown in the following code snippet:

```
   <context:annotation-config />
   <context:component-scan base-
package="org.packt.Spring.chapter3.annotation.false" />
```

Here, in `bean.xml`, we have not added a definition for `Address`. So the bean will not be available to `ApplicationContext`.

The following is the output:

```
From employee class
null
```

Now, even when Spring is unable to autowire because there is no definition in `ApplicationContext`, no runtime exceptions will be thrown. Here we can see that when the `Address` bean is not available, `null` will be returned.

@Qualifier

While using the `@Autowired` annotation, we observed some ambiguity. `@Autowired` is run in a type-driven injection manner. When, in the same object type, you find several beans, you use the `@Qualifier` annotation for more control over deciding the target bean.

```
@Autowired
@Qualifier("qualifier_name")
```

The `@Qualifier` annotation helps us to fine-tune annotation-based autowiring. Sometimes, we create more than one bean of the same type and wire only one of them with a property. In that event, there is no way for `@Autowired` to choose which one you really want, and hence a `NoSuchBeanDefinitionException` will be thrown and wiring will fail. This can be controlled using the `@Qualifier` annotation along with the `@Autowired` annotation.

Let's demonstrate the use of `@Qualifier` along with `@Autowired`.

The `beans.xml` file is shown in the following code snippet:

```
<bean id="employeeA"
class="org.packt.Spring.chapter3.annotation.onsetter.Employee">
   </bean>
   <bean id="employeeB"
class="org.packt.Spring.chapter3.annotation.onsetter.Employee">
   </bean>
   <bean id="employeeC"
class="org.packt.Spring.chapter3.annotation.onsetter.Employee">
   </bean>
```

The `Employee.java` file is shown in the following code snippet:

```
Public class Employee {
    // . . .
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
Public class EmployeeService {

   @Autowired
      @Qualifier("employeeA")
   private Employee employee;
    // . . .
}
```

As you can see, I've used the `@Qualifier` annotation along with the `@Autowired` annotation. Spring will first check for autowire by type after looking at the configuration file for beans of that type. And, in this case, it will get multiple beans of the same type, that is, `Employee`. Beans whose qualifier property value is `employeeA` are injected as employee member variables. In `EmployeeService.java`, I've used `@Qualifier("employeeA");` it means we want to autowire the `Employee` property of `EmployeeService` with the bean `id="employeeService"` in the XML configuration file.

@Resource

The Spring Framework supports dependency injection using the JSR-250 `@Resource` annotation on the bean property or the property setter methods. The `@Resource` annotation is found in Java EE 5 and Java 6, which the Spring Framework supports for Spring-managed objects as well.

The `@Resource` annotation injects the dependency in the variables of the class. It is from `javax.annotation` and isn't specific to the Spring Framework. `@Resource` takes the name as attribute and Spring interprets the associated value as the bean name to be injected.

`@Resource` has the `name` property. Due to this, a designated search name is used by the name property to inject to the element defined by `@Resource` when the Spring Container searches for the bean. If you do not specify the name attribute in `@Resource`, then Spring will interpret the property name as the bean name to be injected and will search for that bean in the configuration file.

Let's demonstrate `@Resource` with the following example: We have annotated the `EmployeeService` class's property with `@Resource`, which will perform Dependency Injection by name.

The `Employee.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...resource;

public class Employee {

   @Override
   public String toString() {
          return "From employee class";
   }
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...resource;

import javax.annotation.Resource;

import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @Resource(name = "employeeA")
   private Employee employee;

   public Employee getEmployee() {
          return employee;
   }
}
```

The `beans.xml` file is shown in the following code snippet:

```
    <context:annotation-config />

    <context:component-scan base-
package="org.packt.Spring.chapter3.annotation.resource" />

    <bean id="employeeA"
class="org.packt.spring.chapter3.annotation.resource.Employee">
    </bean>

    <bean id="employeeB"
class="org.packt.spring.chapter3.annotation.resource.Employee">
    </bean>
```

As you can see, two beans of the same type have been defined in `beans.xml`. In the `EmployeeService` class, we have used `@Resource(name="employeeA")`, which means we want to autowire the `Employee` property of `EmployeeService` with the bean `id="employeeA"` defined in the XML configuration file.

Life cycle annotation

We saw the Spring bean life cycle in the previous chapter. It is composed of initialization, then activation, and finally destruction. When defining the lifecycle method, the necessary action is performed at the initialization or destruction stage. We can use the following annotation to manage the life cycle of the bean even without an interface inheritance or XML definition:

Annotation	Package	Description
`@PostConstruct`	`javax.annotation.PostConstruct`	In Spring, the `@PostConstruct` annotation is used for a method defined in a bean. The annotated method will be called just after the constructor is called. It is an alternate to an initialization callback.

Annotation	Package	Description
`@PreDestroy`	`javax.annotation.PreDestroy`	In Spring, the `@PreDestroy` annotation is used for a method defined in a bean. The annotated method will be called just before the object is garbage-collected. It is an alternate to a destruction callback.

The `@PostConstruct` and `@PreDestroy` annotations are located in the Java EE library (`common-annotation.jar`) and are not a part of Spring.

Let's take a scenario where the `Properties` file gets loaded as soon as a bean gets initialized and the properties file gets cleared when the bean gets destroyed. If the properties file gets loaded for first client request, it means it will take time on first request. To overcome these issues, the properties file should get loaded on bean initialization itself and get destroyed once the bean is destroyed.

Let's take an example where we implement the `@PostConstruct` and `@PreDestroy` annotations:

The `EmployeeService.java` is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...lifecycle;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @PostConstruct
   public void initializeEmployeeService() {
         System.out.println("Init of EmployeeService");
   }

   @PreDestroy
   public void destroyEmployeeService() {
          System.out.println("Destroy Of EmployeeService");
   }
}
```

Here, we have custom methods named `initializeEmployeeService` and `destroyEmployeeService` annotated with `@PostConstruct` and `@PreDestroy` respectively. We have not configured them in the XML configuration file as we did in previous chapter during `init` and `destroy` method in `beans.xml`.

The method `initializeEmployeeService` gets executed when the bean is just initialized and the method `destroyEmployeeService` gets executed when the bean is about to be destroyed.

The `PayrollSystem.java` is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...lifecycle;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {

         ConfigurableApplicationContext context = new
ClassPathXmlApplicationContext(
                     "beans.xml");
         context.close();
   }
}
```

Here, in the `PayrollSystem` class, we have used `ConfigurableApplicationContext`, which will have a `close()` method to perform shutdown.

The `beans.xml` file is shown in the following code snippet:

```
   <context:annotation-config />

   <context:component-scan base-
package="org.packt.spring.chapter3.annotation...lifecycle" />
```

The output will be as follows:

```
Init of EmployeeService
Destroy Of EmployeeService
```

Java-based configuration in Spring

Traditionally, Spring is configured using XML-based Spring configurations to create beans and manage their dependencies. Spring 3.0 introduces a new feature that allows the entire configuration of Spring in Java just by adding some core features of the Spring Java configuration project. The Java-based configuration option enables most of our configurations in Spring without XML. The Java-based configuration also allows bean management using Java code and annotations.

This configuration guarantees type safety. As it is mainly implemented using Java code, if there is any type error in the Dependency Injection, the compile itself is not executed. It also takes out the dependency from Spring and beans can be implemented with pure Java code.

@Configuration annotation

A class annotated with `@Configuration` is the central artifact of Java-based Spring configuration. A class annotated with the `@Configuration` annotation can be used by a Spring Container as a source of bean definitions. This class will contain one or more Spring bean declarations. Let's see how to use the `@Configuration` annotation.

The `SpringConfig.java` is shown in the following code snippet:

```
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringConfig {
   // bean declaration methods
}
```

In the preceding code snippet, we have annotated a class with the `@Configuration` annotation:

- An application can use one or more class that has been annotated with `@Configuration`.
- The `@Configuration` annotation is meta-annotated as a `@Component` annotation.
- Classes annotated with `@Configuration` are candidates for component scanning. These classes can also take advantage of `@Autowired` annotations at the field and method levels, but not at the constructor level.
- Classes annotated with `@Configuration` must also have a default constructor.
- The `@Value` annotation can be used to wire externalized values into classes annotated with `@Configuration`.

@Bean annotation

The `@Bean` annotation is a method-level annotation that is a direct analog of the XML `<bean/>` element and is used to wire beans using Spring's Java-based configuration. This annotation supports some of the attributes offered by the `<bean/>` tag in XML, such as `init-method`, `destroy-method`, `autowiring`, and `name`. It can be used in a class either annotated with `@Configuration` or `@Component`.

Declaring a bean

We can declare a bean by annotating a method with `@Bean`. This annotated method is used to register a bean definition of the type specified as the method's return value within an `ApplicationContext`.

The bean name, by default, will be taken from the method name. Let's understand how to use the `@Bean` annotation to wire beans using Spring's Java-based configuration. The object returned from this annotated method will be registered as a bean in the Spring `ApplicationContext`:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringConfig {

   @Bean
   public Employee employee() {
          return new Employee();
   }
}
```

The preceding method is the Java configuration equivalent of the following Spring XML:

```
<beans>
    <bean name="employee"
class="org.packt.Spring.chapter3.Employee"/>
</beans>
```

Both declarations register a bean named `employee` in `ApplicationContext`, bound to an object instance of type `Employee`.

Customizing bean naming

By default, the bean name will be taken from a method's name annotated with `@Bean`. This functionality can be overridden with the `name` attribute. Here, the bean named `employeeBean` will be registered in `ApplicationContext`.

The `SpringConfig.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.javabased;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringConfig {

   @Bean(name="employeeBean")
   public Employee employee() {
          return new Employee();
   }
}
```

The `MainApp.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.javabased;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
AnnotationConfigApplication
Context;

public class MainApp {

   public static void main(String[] args) {

          ApplicationContext context = new
AnnotationConfigApplicationContext(
                       SpringConfig.class);
          Employee employee = (Employee)
context.getBean("employeeBean");
   }
}
```

Injecting dependencies

Injecting dependencies using Java-based Spring configuration is as simple as having one bean method call another:

```
@Configuration
public class SpringConfig {

    @Bean
    public EmployeeService employeeService() {
        return new EmployeeDao(employeeDao());
    }

    @Bean
    public EmployeeDao employeeDao() {
        return new EmployeeDao();
    }
}
```

In this example, the `employeeService` bean receives a reference to `employeeDao` via constructor injection.

@Import annotation

To load `@Bean` definitions from another configuration class, we can use the `@Import` annotation.

Let's consider a configuration class `ConfigA`:

```
@Configuration
public class ConfigA {

   @Bean
   public EmployeeService employeeService() {
      return new EmployeeService();
   }
}
```

You can import the bean declaration from the `ConfigA` class to another configuration class as shown here:

```
@Configuration
@Import(ConfigA.class)
public class ConfigB {

   @Bean
```

```
    public HrService hrService() {
       return new HrService();
    }
}
```

Only `ConfigB` needs to be supplied when instantiating the context:

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

   ApplicationContext context =
   new AnnotationConfigApplicationContext(ConfigB.class);

   // now both beans A and B will be available...
   ConfigA a = context.getBean(ConfigA.class);
   ConfigB b = context.getBean(ConfigB.class);
}
```

Life cycle callbacks

The regular life cycle callbacks are supported by beans created in a class annotated with `@Configuration`. The `@PostConstruct` and `@PreDestroy` annotations from JSR-250 can be used by any class defined within the `@Bean` annotation. The Java-based configuration fully supports Spring life cycle callbacks.

The initialization and destruction callback methods are supported by the `@Bean` annotation, which is same as XML's `init-method` and `destroy-method` attributes to the bean element:

```
public class EmployeeService {

    public void init() {
        // initialization logic
    }

    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class SpringConfig {
    @Bean(initMethodName = "init" , destroyMethod = "cleanup")
    public EmployeeService employeeService() {
        return new EmployeeService();
    }
}
```

@Scope annotation

The beans defined with the `@Bean` annotation is by default singleton, but you can override this with the `@Scope` annotation as follows:

```
@Configuration
public class AppConfig {

    @Bean
    @Scope("prototype")
    public EmployeeService employeeService() {
        // ...
    }
}
```

Event handling in Spring

Sometimes, it requires reacting to an event that happened in the Spring Container. Spring provides a way to listen to the events that may allow you to react on the context. Spring allows you to handle events triggered by the framework. An event is triggered because of a state change or some activities within the container.

`ApplicationContext` manages the complete life cycle of the beans of singleton type and publishes certain types of events on bean loading. For example, when the application context is started, then the `ContextStartedEvent` is published. Similarly, when the context is stopped, then the `ContextStoppedEvent` is published.

Event handling is the important feature provided by `ApplicationContext`. It consists of three core components:

- `ApplicationListener`: A class that listens to an event has to implement the `ApplicationListener` interface. Every time the `ApplicationEvent` gets published, the bean that implemented the `ApplicationListener` will be notified. It is required to implement the `onApplicationEvent()` method while implementing this interface.
- `ApplicationEventPublisher`: A class that publishes an event has to implement the `ApplicationEventPublisher` interface. By calling an application event publisher's `publishEvent()` method, a bean can publish an event.
- `ApplicationEvent`: This class is used when we are writing our own custom event and adding additional functionality and additional metadata about the event.

Standard events in Spring

The following table shows the standard events in Spring:

Event	Description
`ContextRefreshedEvent`	When `ApplicationContext` is either initialized or refreshed, `ContextRefreshedEvent` is published. Here, initialized means when all beans are loaded into the container and the `ApplicationContext` object is ready for use. A refresh using the `refresh()` method can be triggered multiple times as long as the context has not been closed and the chosen `ApplicationContext` actually supports refreshes. For example, `GenericApplicationContext` does not support refreshes whereas `XmlWebApplicationContext` does.
`ContextStartedEvent`	This is published on starting `ApplicationContext` using the `start()` method on the `ConfigurableApplicationContext` interface. When `ApplicationContext` starts, the life cycle of beans receive an explicit start signal. This event can be used to restart beans after an explicit stop. This event can also be used to start components that have not been configured for autostart.
`ContextStoppedEvent`	When `ApplicationContext` is stopped using the `stop()` method on the `ConfigurableApplicationContext` interface, this event is published and the life cycle of beans receive an explicit stop signal.
`ContextClosedEvent`	This is published when `ApplicationContext` is closed using the `close()` method on the `ConfigurableApplicationContext` interface. It indicates that all singleton beans are destroyed and cannot be refreshed or restarted as it reaches the end of its life.
`RequestHandledEvent`	This is an event specific to the Web, telling all beans that an HTTP request has been serviced. After the request is complete, this event is published. Only web applications that use Spring's `DispatcherServlet` can apply this event.

Listening to standard events in Spring

To listen to all events generated by `ApplicationContext`, a class needs to implement the `ApplicationListener` interface that is provided by the Spring Framework. The `ApplicationListener` interface can be found in the Spring Context package.

We need to implement the `onApplicationEvent()` method while implementing this interface, which is called by Spring when the application event is published. This method takes a parameter of type `ApplicationContextEvent`, and Spring will send the `ApplicationEvent` object to this method.

Creating listeners

The first step to creating listeners is to create some beans that can listen to events.

The `MyApplicationEventListener.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...listenevent;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStartedEvent;
import org.springframework.stereotype.Component;

@Component
public class MyApplicationEventListener implements
          ApplicationListener<ContextStartedEvent> {

   @Override
   public void onApplicationEvent(ContextStartedEvent arg0) {
          System.out.println("Context Started Event Received");
   }
}
```

Here, when an `ApplicationContext` will start, an event will be published in the Spring Framework and the `onApplicationEvent()` method will get executed. The presence of generics in the `ApplicationListener` interface allows us to listen for very specific events. In this case, our listener listens to `ContextStartedEvent` of type `ApplicationEvent`.

The `beans.xml` is shown in the following code snippet:

```
   <context:annotation-config />
   <context:component-scan base-
package="org.packt.spring.chapter3.annotation...listenevent" />
```

The `MainApp.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...listenevent;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;
public class MainApp {

   public static void main(String[] args) {

            ConfigurableApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
            context.start();
   }
}
```

Here, the `start()` method will propagate the start signal to component.

Publishing an event in Spring

Spring allows us to create a bean that can publish an event. Events may be framework events (such as `RequestHandledEvent`) or application-specific events.

Creating a custom event

To enable event-based communication, let's first define an event. In Spring, an event is created by extending the abstract class `ApplicationEvent` and passing the event source as the constructor argument.

The `LoginEvent.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...publishevent;

import org.springframework.context.ApplicationEvent;

public class LoginEvent extends ApplicationEvent{

   public LoginEvent(Object source) {
         super(source);
   }

   public String toString() {
          return "Hello World";
   }
}
```

Publishing an event

To publish an event, a class has to implement the `ApplicationEventPublisherAware` interface. The `ApplicationEventPublisher` interface notifies all registered listeners of this application. A class that implements the `ApplicationListener` interface can handle published events. Pass an instance of the event to the `publishEvent()` method.

The `LoginEventPublisher.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...publishevent;

import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.stereotype.Component;

@Component
public class LoginEventPublisher implements
ApplicationEventPublisherAware {

   private ApplicationEventPublisher publisher;

   @Override
   public void
setApplicationEventPublisher(ApplicationEventPublisher publisher)
{
           this.publisher = publisher;
   }

   public void publish() {
         LoginEvent loginEvent = new LoginEvent(this);
         publisher.publishEvent(loginEvent);
   }
}
```

Handling custom event

The `LoginEventListener.java` is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...publishevent;

import org.springframework.context.ApplicationListener;
```

```
import org.springframework.stereotype.Component;

@Component
public class LoginEventListener implements
ApplicationListener<LoginEvent> {

   @Override
   public void onApplicationEvent(LoginEvent event) {
         if (event instanceof LoginEvent) {
                System.out.println(event);
         }
   }
}
```

We need to filter the events that the listener needs to handle. Here, in the `LoginEventListener` class, we use an instance of check to filter on the non-generic.

The `MainApp.java` file is shown in the following code snippet:

```
package org.packt.spring.chapter3.annotation...publishevent;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class MainApp {

   public static void main(String[] args) {

          ApplicationContext context = new
ClassPathXmlApplicationContext(
                       "beans.xml");

          LoginEventPublisher loginEventPublisher =
(LoginEventPublisher) context
                      .getBean("loginEventPublisher");

          loginEventPublisher.publish();
   }

}
```

Benefits of event in Spring

The following are the benefits of using events in Spring:

- The `ApplicationContext` in Spring supports event-based communication based between its beans.
- In event-based communication, without knowing who the receiver will be, the sender component just publishes the event.
- The receiver doesn't need to know who is publishing the event. It can listen to multiple events from different senders at the same time.
- Sender and receiver are loosely coupled.

Spring Expression Language

Spring Expression Language (**SpEL**) is a powerful expression language that supports features to query and manipulate an object graph at runtime. SpEL can be used to dynamically evaluate properties and use them as a value configured in the IoC controller. SpEL supports mathematical, logical, and relational operators. Regular expressions are also supported in SpEL using the `matches` operator.

SpEL provides dynamic bean wiring at runtime. SpEL picks the right bean or value to `dependency-inject` at runtime. We can also use SpEL to inject a bean or a bean property in another bean, or even to invoke a bean method in another bean.

Here are the features of SpEL:

- To reference beans using a bean's ID
- To inject methods and properties on beans
- To perform mathematical, logical, and relational operations on values
- To match regular expressions
- To manipulate collections

SpEL expressions can be used either with XML- or annotation-based configuration metadata for defining bean definitions. In both cases, the syntax to define the expression is of the form `#{<expression string>}`, which is evaluated when the bean is created and not when the `ApplicationContext` is initialized. The `#{}` part of the syntax indicates that it contains a SpEL expression. For example, to wire a string literal into a bean property, express it like this:

```
<property name="publisher" value="#{'Packt'}">
</property>
```

Bean reference using SpEL

Let's consider an example to get an idea how the wiring of beans can become very dynamic using SpEL. SpEL allows you to reference an already defined bean or one of its properties and dependency-inject it into another object. Here, we will inject `addressBean` into another `customerBean`. The `#{addressBean}` expression is used to inject `addressBean` into the `customerBean` bean's `address` property, whereas the `# {addressBean.country}` is used to inject the country property of `addressBean` into the country property of `customerBean`. Use dot notation to reference properties of a bean in SpEL.

XML-based configuration

The `beans.xml` file is shown in the following code snippet, which will define the `addressBean` and `customerBean` beans, and then we will inject `addressBean` into another `customerBean`:

```
<bean id="addressBean" class="Address">
   <property name="id" value="12345"></property>
   <property name="streetName" value="24th Main"></property>
   <property name="country" value="India"></property>
</bean>

<bean id="customerBean" class="Customer">
   <property name="custName" value="Ravi"></property>
   <property name="address"
           value="#{addressBean}">
   </property>
   <property name="country"
           value="#{addressBean.country}">
   </property>
</bean>
```

Annotation-based configuration

The `Address.java` is shown in the following code snippet, which will define the `addressBean` bean using annotation:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("addressBean")
public class Address {
```

```
    @Value("12345")
    private Long id;

    @Value("24th Main")
    private String streetName;

    @Value("India")
    private String country;

    // setter & getter
}
```

The `Customer.java` is shown in the following code snippet, which will define the `customerBean` bean using annotation:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {

    @Value("Ravi")
    private String custName;

    @Value("#{addressBean}")
    private Address address;

    @Value("#{addressBean.country}")
    private String country;

    // setter & getter
}
```

Method invocation using SpEL

You can invoke a method with SpEL to execute the method and inject the method's returned value into a property. We'll use the expression `Language` to execute a method in `addressBean` and inject the result into `customerBean`. The `#{addressBean.getFullAddress()}` is used to execute a method of `addressBean` and return the value into the `fullAddress` property of `customerBean`. Use dot notation to reference the method of a bean in SpEL.

XML-based configuration

The `beans.xml` is shown in the following code snippet, which will use the expression `Language` to execute a method in `addressBean` and inject the result into `customerBean`:

```
<bean id="addressBean" class="Address">
   <property name="id" value="12345"></property>
   <property name="streetName" value="24th Main"></property>
   <property name="country" value="India"></property>
</bean>
<bean id="customerBean" class="Customer">

   <property name="custName" value="Ravi"></property>
   <property name="fullAddress"
          value="#{addressBean.getFullAddress()}">
   </property>
</bean>
```

Annotation-based configuration

The `Address.java` file is shown in the following code snippet, which will create the `addressBean` bean using annotation:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("addressBean")
public class Address {

   @Value("12345")
   private Long id;

   @Value("24th Main")
   private String streetName;

   @Value("India")
   private String country;

   public String getFullAddress() {
          return streetName + " " + country;
   }

   // setter & getter
}
```

The `Customer.java` file is shown in the following code snippet, which will create the `customerBean` bean with annotation:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {

   @Value("Ravi")
   private String custName;

   @Value("#{addressBean.getFullAddress()}")
   private String fullAddress;

   // setter & getter
}
```

Operators with SpEL

SpEL supports most of the standard mathematical, logical, or relational operators that you can use to manipulate the values of an expression. Here's a list:

Operator	Function	Sign
Relational operator	equal	`==, eq`
	not equal	`!=, ne`
	less than	`<, lt`
	greater than	`>, gt`
	less than or equal	`<=, le`
	greater than or equal	`>=, ge`
Logical operator	and	
	or	
	not	`!`
Mathematical operator	addition	`+`
	subtraction	`-`
	multiplication	`*`
	division	`/`
	modulus	`%`
	exponential power	`^`

XML-based configuration

In XML, symbols like < are not supported. We must instead use the textual equivalents shown in the table; for example, use `lt` instead of < and `le` instead of <=.

The `beans.xml` file is shown in the following code snippet:

```
<bean id="numberBean"
class="org.packt.springframework.chapter3.SpEL.operator.Number">
   <property name="no" value="999" />
</bean>
<bean id="calculatorBean"
class="org.packt.springframework.chapter3.SpEL.operator.Calculator
">
<!---------------------------------------->
<! -------- Relational Operator --------->
<!---------------------------------------->

   <property name="isEqual" value="#{1 == 1}" />
   <property name="isNotEqual" value="#{1 != 1}" />
   <property name="isLessThan" value="#{1 lt 1}" />
   <property name="isLessThanOrEqual" value="#{1 le 1}" />
   <property name="isGreaterThan" value="#{1 > 1}" />
   <property name="isGreaterThanOrEqual" value="#{1 >= 1}" />


<!---------------------------------------->
<! --------- Logical Operator ----------->
<!---------------------------------------->

   <property name="isAnd" value="#{numberBean.no == 999 and
numberBean.no lt 900}" />
   <property name="isOr" value="#{numberBean.no == 999 or
numberBean.no lt 900}" />
   <property name="isNot" value="#{numberBean.no != 999}"
/>


<!---------------------------------------->
<! -------- Mathematical Operator ------->
<!---------------------------------------->

   <property name="addition" value="#{1 + 1}" />
   <property name="subtraction" value="#{1 - 1}" />
```

```
    <property name="multiplication" value="#{1 * 1}"/>
    <property name="division" value="#{10 / 2}" />
    <property name="modulus" value="#{10 % 10}" />

</bean>
```

Annotation-based configuration

The `Number.java` is shown in the following code snippet:

```
package org.packt.springframework.chapter3.SpEL.operator;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("numberBean")
public class Number {

   @Value("999")
   private int no;

   // setter & getter
}
```

The `Calculator.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.SpEL.operator;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class Calculator {

   /* Relational operators */

   @Value("#{1 == 1}")// true
   private boolean isEqual;

   @Value("#{1 != 1}") // false
   private boolean isNotEqual;

   @Value("#{1 < 1}")  // false
```

```
    private boolean isLessThan;

    @Value("#{1 <= 1}") // true
    private boolean isLessThanOrEqual;

    @Value("#{1 > 1}")// false
    private boolean isGreaterThan;

    @Value("#{1 >= 1}")// true
    private boolean isGreaterThanOrEqual;

    /* Logical operators , numberBean.no == 999 */

    @Value("#{numberBean.no == 999 and numberBean.no < 900}")
// false
    private boolean isAnd;

    @Value("#{numberBean.no == 999 or numberBean.no < 900}")// true
    private boolean isOr;

    @Value("#{!(numberBean.no == 999)}")// false
    private boolean isNot;

    /* Mathematical operators */

    @Value("#{1 + 1}")// 2.0
    private double addition;

    @Value("#{1 - 1}")// 0.0
    private double subtraction;

    @Value("#{1 * 1}")// 1.0
    private double multiplication;

    @Value("#{10 / 2}") // 5.0
    private double division;

    @Value("#{10 % 10}")// 0.0
    private double modulus;

    // setter & getter
}
```

Collections with Spring EL

Spring Expression Language can be used to get the values of a map and a list.

XML-based configuration

The `beans.xml` file is shown in the following code snippet:

```
<bean id="employeeBean"
class="org.packt.springframework.chapter3.SpEL.collection.
Employee">
</bean>

<bean id="employeeServiceBean" class="
org.packt.springframework.chapter3.SpEL.collection.
EmployeeService">

    <property name="mapElement"
   value="#{employeeBean.map['emp1']}"/>
    <property name="listElement"
   value="#{employeeBean.list[0]}"/>

</bean>
```

Annotation-based configuration

The `Employee.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.SpEL.collection;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.stereotype.Component;

@Component("employeeBean")
public class Employee {

   private Map<String, String> map;
   private List<String> list;

   Employee() {
```

```
            map = new HashMap<String, String>();
            map.put("emp1", "Employee 1");
            map.put("emp2", "Employee 2");

            list = new ArrayList<String>();
            list.add("employee-1");
            list.add("employee-2");
    }

    // getter & Setter
}
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.SpEL.collection;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class EmployeeService {

   @Value("#{employeeBean.map['emp1']}")
   private String mapElement;

   @Value("#{employeeBean.list[0]}")
   private String listElement;
   // getter & Setter
}
```

Regular expressions with Spring EL

Spring EL supports regular expressions using the keyword "`matches`". It is used to check whether text matches a certain pattern:

Operator	Result	Condition
`matches`	return `true`	If the given value matches the regular expression.
	return `false`	If the given value doesn't match the regular expression.

Let's consider an example to check whether a string contains a valid e-mail address using the matches operator.

XML-based configuration

The `beans.xml` file is shown in the following code snippet:

```
<bean id="emailBean"
class="org.packt.springframework.chapter3.SpEL.regularexpression
.Email">
   <property name="emailAddress" value="spam@abc.com"/>
</bean>
<bean id="clientBean" class="org.packt.springframework.chapter3.SpEL.
regularexpression
.Client">

   <property name="validEmail"
   value="#{emailBean.emailAddress  matches  '
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.com'  }" />

</bean>
```

Annotation-based configuration

The `Email.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.SpEL.regularexpression;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("emailBean")
public class Email {

   @Value("spam@abc.com")
   String emailAddress;

   //getter and setter methods
}
```

The `Client.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.SpEL.regularexpression;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("clientBean")
```

```
public class Client {

   // email regular expression
   String emailRegEx = "^[_A-Za-z0-9-]+(\\.[_A-Za-z0-9-]+)"
               + "*@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)*
(\\.[A-Za-z]{2,})$";

   // emailBean.emailAddress contains valid email address?
   @Value("#{emailBean.emailAddress matches
clientBean.emailRegEx}")
   private Boolean validEmail;
   //getter and setter methods
}
```

Text message and internationalization

The `ApplicationContext` has an additional functionality over a `BeanFactory` support for messaging and internationalization. The `MessageSource` interface is used to retrieve messages out of property files. If you have messages that need to be displayed in different places in your application UI, you would not want to have those messages inside the code either in the Java class or JSP. You want to take all messages out and keep them in a common properties file and all referred text inside the properties file and print out that text instead of having the text itself reside in the source code.

Spring provides support for text messages and internationalization. You can have a file that consolidates all these key-value pairs and then you can specify the key itself in your application and refer to the text that is assigned to the key in the properties file.

Here, we will see how to implement support for messaging using Spring.
We will create a simple properties file, `messages.properties`, and add all the key-value pairs to store all messages inside the properties file. Let's say we use `user.welcome=Hello, Welcome to Spring!`, where `user.welcome` is the key and `Hello, Welcome to Spring!` is the value. We will then use this key to print the value associated with this key in our program.

The `messages.properties` is shown here:

```
user.welcome=Hello, Welcome to Spring!
```

In order to have messaging support enabled, we need to have a `MessageSource` bean defined in my application:

```
<bean id="messageSource"
class="org.springframework.context.support.
ResourceBundleMessageSource"/>
```

Here, the `org.springframework.context.support.ResourceBundleMessageSource` class provided by Spring helps to pick messages out of the properties files. It needs to map the properties file to the `ResourceBundleMessageSource` class.

For the property named `basename`, we will provide a list of all the properties files where we have stored all the messages. The list will have a value tag that will take the properties filename without extension.

The `beans.xml` file is shown in the following code snippet:

```
<bean id="messageSource"
class="org.springframework.context.support.
ResourceBundleMessageSource">
          <property name="basenames">
                <list>
                      <value>messages</value>
                </list>
           </property>
</bean>
```

So, a `messageSource` bean knows the properties files to read, as the class will look for the file with the extension `.properties`, that is, `messages.properties` in the classpath.

The `ApplicationContext` has a method called `getMessage` with different signatures. Here, we will use a method that takes four arguments:

```
String org.springframework.context.MessageSource.getMessage
( String code, Object[] args, String defaultMessage, Locale
locale )
```

The following are the different arguments covered in the previous line of code:

- `code`: This will take the key from the `messages.properties` file, that is, `user.welcome`.
- `args`: This will take an array of objects in case the message takes some parameter like dynamic value. Since we aren't passing any parameter now, we will set it as `null`.

- `defaultMessage`: This will be returned in case it does not find the message we are looking for, that is, default message. So, when `arg0` does not match the key in the properties file, this default value will be printed.
- `locale`: This indicates the locale for which we need to get the message. Here, we are passing null, as we have not defined any locale now. We will looks at locales later in this chapter.

The `PayrollSystem.java` file is shown in the following code snippet:

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");

System.out.println(context.getMessage("user.welcome", null,
"Default Greeting", null));
```

If the code executes successfully, it will print the value from the properties file, as shown here:

```
Hello, Welcome to Spring!
```

In `beans.xml`, we defined a bean named `messageSource` of the class `ResourceBundleMessageSource` and it implements an interface called `messageSource` that has a method `getMessage`. So, as long as we have a member variable of the message source inside our bean, let's say `EmployeeService.java`, and we can do Dependency Injection and we can get `messageSource` bean inside `EmployeeService` class, and we can perform `getMessage` for that bean.

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.messaging;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @Autowired
   private MessageSource messageSource;

   @Override
   public String toString() {
```

```
            return messageSource.getMessage("user.welcome", null,
                        "Default Greeting", null);
    }

}
```

Internationalization (i18n) in Spring

Internationalization refers to the process in which you design a software application so that it can be adapted, without any engineering changes, to various languages and regions. Spring 3 provides classes for i18n support.

If you have multiple locales to supply messages to, let's say `messages_en.properties` for English and `messages_de.properties` for Deutsch, we can pass the corresponding locale to the last parameter of the method `getMessage`. And the corresponding message picked up from the right properties file.

This is shown here:

```
user.welcome=Hello, Welcome to Spring!
```

The `messages_de.properties` is shown here:

```
user.welcome=Hallo, Herzlich Willkommen Frühling!
```

The `EmployeeService.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.internationalization;

import java.util.Locale;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

   @Autowired
   private MessageSource messageSource;

   public String toGerman() {
          return messageSource.getMessage("user.welcome", null,
                     "Default Greeting", Locale.GERMAN);
```

```
    }

    public String toEnglish() {
           return messageSource.getMessage("user.welcome", null,
                      "Default Greeting", Locale.ENGLISH);
    }
}
```

Here, we have passed `Locale.English`, which will match with the key in the file `messages_en.properties`. We have also passed `Local.German`, which will match with the key in the file `messages_de.properties`.

The `PayrollSystem.java` file is shown in the following code snippet:

```
package org.packt.springframework.chapter3.internationalization;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
ClassPathXmlApplicationContext;

public class PayrollSystem {

   public static void main(String[] args) {

          ApplicationContext context = new
ClassPathXmlApplicationContext(
                       "beans.xml");

          EmployeeService employeeService = (EmployeeService)
context
                      .getBean("employeeService");

           System.out.println("English: " +
employeeService.toEnglish());
           System.out.println("German: " +
employeeService.toGernam());
   }
}
```

When we run the preceding code, the output will printed as follows:

```
English: Hello, Welcome to Spring!
German: Hallo, Herzlich Willkommen Frühling!
```

Exercise

Q1. What are Stereotype annotations?

Q2. Explain the different components of event handling.

Q3. What is Spring Expression Language (SpEL)?

Summary

In this chapter, we minimized the XML configurations in Spring. We configured containers using annotations and compared XML configuration with annotation-based configuration.

We then covered bean management using the stereotype annotation. We understood the different stereotype annotations such as `@Component`, `@Repository`, `@Service`, and `@Controller` to define stereotyped classes. We used the `@Configuration` annotation to define a configuration class that can have methods annotated with the `@Bean` annotation to define beans.

Then we created a custom event and published that event in Spring. We discussed Spring Expression Language (SpEL) to dynamically evaluate properties and use them as values configured in the IoC controller using XML and annotations. Then we implemented text messaging and internationalization (i18n) in Spring.

In the next chapter, we will take a look at Aspect Oriented Programming in Spring. We will define aspect using XML and annotations in Spring. We will also understand proxies in Spring.