# Spring Dependency Injection

<http://www.tutorialspoint.com/spring/spring_dependency_injection.htm>

Every java based application has a few objects that work together to present what the end-user sees as a working application. When writing a complex Java application, application classes should be as independent as possible of other Java classes to increase the **possibility to reuse** these classes and to test them independently of other classes while doing unit testing. Dependency Injection (or sometime called wiring) helps in **gluing these classes together and same time keeping them independent.**

**Real time example:**

1. Parsing log file: Code to parse log file- high level. Log file- low level.

Whatever log file needs to be parsed has to be converted to a type of object which will be accepted by this code. 🡪 so high level (code) controls low level (log file/input) rather than the type of log file determining the code because in that case the code can be used only for a particular case and cannot be reused so for another type of log file, new code will be required.

1. In DAO, when DB connection is made, an instance of the connection object is injected by spring container and DAO class need not worry about the instance creation.

Eg:

In DAOImpl class we will have the below line for DB connection.

**jdbcTemplate = new JdbcTemplate(dataSource);**

In ApplicationContext file, datasource properties will be configured

<bean

id="employeeDAO" class="com.javacodegeeks.snippets.enterprise.dao.impl.EmployeeDAOImpl">

<property name="dataSource" ref="dataSource" />

</bean><bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/test" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

Consider you have an application which has a text editor component and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor() {

spellChecker = new SpellChecker();

}

}

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an inversion of control scenario we would instead do something like this:

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

this.spellChecker = spellChecker;

}

}

Here TextEditor should not worry about SpellChecker implementation. The SpellChecker will be implemented independently and will be provided to TextEditor at the time of TextEditor instantiation and this entire procedure is controlled by the Spring Framework.

Here, we have removed the total control from TextEditor and kept it somewhere else (ie. XML configuration file) and the dependency ( ie. class SpellChecker) is being injected into the class TextEditor through a **Class Constructor**. Thus flow of control has been "inverted" by Dependency Injection (DI) because you have effectively delegated dependances to some external system.

Second method of injecting dependency is through **Setter Methods** of TextEditor class where we will create SpellChecker instance and this instance will be used to call setter methods to initialize TextEditor's properties.

Spring Setter-based Dependency Injection

Here is the content of **TextEditor.java** file:

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

// a setter method to inject the dependency.

public void setSpellChecker(SpellChecker spellChecker) {

System.out.println("Inside setSpellChecker." );

this.spellChecker = spellChecker;

}

// a getter method to return spellChecker

public SpellChecker getSpellChecker() {

return spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Here you need to check naming convention of the setter methods. To set a variable **spellChecker** we are using **setSpellChecker()** method which is very similar to Java POJO classes. Let us create the content of another dependent class file **SpellChecker.java**:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

Following is the content of the **MainApp.java** file:

package com.tutorialspoint;

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** which has configuration for the setter-based injection:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<property name="spellChecker" ref="spellChecker"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

You should note the difference in Beans.xml file defined in constructor-based injection and setter-based injection. The only difference is inside the <bean> element where we have used <constructor-arg> tags for constructor-based injection and <property> tags for setter-based injection.

# Spring Constructor-based Dependency Injection

Here is the content of **TextEditor.java** file:

package com.tutorialspoint;

public class TextEditor {

private SpellChecker spellChecker;

public TextEditor(SpellChecker spellChecker) {

System.out.println("Inside TextEditor constructor." );

this.spellChecker = spellChecker;

}

public void spellCheck() {

spellChecker.checkSpelling();

}

}

Following is the content of another dependent class file**SpellChecker.java**:

package com.tutorialspoint;

public class SpellChecker {

public SpellChecker(){

System.out.println("Inside SpellChecker constructor." );

}

public void checkSpelling() {

System.out.println("Inside checkSpelling." );

}

}

Following is the content of the **MainApp.java** file:

package com.tutorialspoint;

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");

TextEditor te = (TextEditor) context.getBean("textEditor");

te.spellCheck();

}

}

Following is the configuration file **Beans.xml** which has configuration for the constructor-based injection:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Definition for textEditor bean -->

<bean id="textEditor" class="com.tutorialspoint.TextEditor">

<constructor-arg ref="spellChecker"/>

</bean>

<!-- Definition for spellChecker bean -->

<bean id="spellChecker" class="com.tutorialspoint.SpellChecker">

</bean>

</beans>

Once you are done with creating source and bean configuration files, let us run the application. If everything is fine with your application, this will print the following message:

Inside SpellChecker constructor.

Inside TextEditor constructor.

Inside checkSpelling.

**Java Based**

**Spring Configuration with Annotations**

For annotation based configuration, we need to write a Configurator class that will be used to inject the actual implementation bean to the component property.

DIConfiguration.java

package com.journaldev.spring.di.configuration;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import com.journaldev.spring.di.services.EmailService;

import com.journaldev.spring.di.services.MessageService;

@Configuration

@ComponentScan(value={"com.journaldev.spring.di.consumer"})

public class DIConfiguration {

@Bean

public MessageService getMessageService(){

return new EmailService();

}

}

Some important points related to above class are:

@Configuration annotation is used to let Spring know that it’s a Configuration class.

@ComponentScan annotation is used with @Configuration annotation to specify the packages to look for Component classes.

@Bean annotation is used to let Spring framework know that this method should be used to get the bean implementation to inject in Component classes.

**Annotations based DI**

### What is Annotation-based container configuration?

An alternative to XML setups is provided by annotation-based configuration which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. **Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration.**

@Autowired tells the Spring IOC container to find configured object of this type and inject an instance at runtime.

package com.javapapers.spring.ioc;

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

import org.springframework.stereotype.Service;

@Service

public class Zoo {

private WildAnimal wild;

@Autowired

public void setWild(WildAnimal wild) {

this.wild = wild;

}

public void testSound() {

System.out.println(wild.sound());

}

}

spring-context.xml – gives the configuration information to the spring container. In this I describe that we are using annotation based configurations.

<?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:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- Dependency Injection with annotations -->

<context:component-scan base-package="com.javapapers.spring.ioc" />

<mvc:annotation-driven />

<!-- Dependency Injection without annotations -->

<!--

<bean id="wild" class="com.javapapers.spring.ioc.Wolf" />

<bean id="zoo" class="com.javapapers.spring.ioc.Zoo">

<property name="wild" ref="wild" />

</bean>

-->

</beans>

# ****Spring JdbcTemplate Example****

[**http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/**](http://examples.javacodegeeks.com/enterprise-java/spring/jdbc/spring-jdbctemplate-example/)

When we need to interface with databases the Spring JDBC framework provides solutions to all the low-level details, like open/close a connection, prepare and execute SQL statements, process exceptions and handle transactions. Thus, the only thing a developer must do is just define connection parameters and specify the SQL statement to be executed.

Spring JDBC provides several approaches and different classes to form the basis for a JDBC database access. The most popular approach makes use of JdbcTemplateclass. This is the central framework class that manages all the database communication and exception handling.

Once a connection is obtained we can interact with the database. The JDBC *Statement, CallableStatement,* and *PreparedStatement*interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

Following table provides a summary of each interface's purpose to understand how do you decide which interface to use?

|  |  |
| --- | --- |
| **Interfaces** | **Recommended Use** |
| Statement | Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters. |
| PreparedStatement | Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime. |
| CallableStatement | Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters. |

**Spring Configuration:**

In web.xml, we will configure the [front controller](http://en.wikipedia.org/wiki/Front_Controller_pattern) for spring framework that is [DispatcherServlet](http://static.springsource.org/spring/docs/3.1.x/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html" \o "DispatcherServlet).

**web.xml**

 <servlet>

        <servlet-name>spring</servlet-name>

        <servlet-class>

            org.springframework.web.servlet.DispatcherServlet

        </servlet-class>

        <load-on-startup>1</load-on-startup>

    </servlet>

    <servlet-mapping>

        <servlet-name>spring</servlet-name>

        <url-pattern>.ftl</url-pattern>

    </servlet-mapping>

**Hibernate Configuration in Spring:**

## Entity Class

It is a simple class defines the properties mapped to each column in the corresponding DB table.

**DispatcherServlet i.e spring-servlet.xml**

    <bean id="dataSource"

        class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"

        p:driverClassName="${jdbc.driverClassName}"

        p:url="${jdbc.databaseurl}"

p:username="${jdbc.username}"

        p:password="${jdbc.password}"></bean>

    <bean id="sessionFactory"

        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

        <property name="dataSource" ref="dataSource"></property>

        <property name="configLocation">

            <value>classpath:hibernate.cfg.xml</value>

        </property>

        <property name="configurationClass">

            <value>org.hibernate.cfg.AnnotationConfiguration</value>

        </property>

        <property name="hibernateProperties">

            <props>

                <prop key="hibernate.dialect">${jdbc.dialect}</prop>

                <prop key="hibernate.show\_sql">true</prop>

            </props>

        </property>

    </bean>

    <bean id="employeeDAO" class="com.howtodoinjava.dao.EmployeeDaoImpl"></bean>

    <bean id="employeeManager" class="com.howtodoinjava.service.EmployeeManagerImpl"></bean>

    <tx:annotation-driven />

    <bean id="transactionManager"

        class="org.springframework.orm.hibernate3.HibernateTransactionManager">

        <property name="sessionFactory" ref="sessionFactory"></property>

    </bean>

**hibernate.cfg.xml**

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<mapping class="com.howtodoinjava.entity.**EmployeeEntity**"></mapping>

***<!—Here You are referring to the entity class directly because the mapping is done using annotation directly in the component class(i.e entity class) -- >***

</session-factory>

</hibernate-configuration>

**(Or)**

**Below is an example where datasource is configured in hibernate.cfg.xml**

**hibernate.cfg.xml**

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost:3306/test</property>

<property name="connection.username">root</property>

<property name="connection.password">admin</property>

<!-- JDBC connection pool (use the built-in) -->

<property name="connection.pool\_size">1</property>

<!-- SQL dialect -->

<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Echo all executed SQL to stdout -->

<property name="show\_sql">true</property>

<!-- Mapping files -->

<mapping resource="javabeat/net/hibernate/Employee.hbm.xml"/>

***<!—Here You are referring to the hbm.xml file because the mapping is done in XMl file and not directly using annotations in the component class(i.e entity class) –as shown below >***

</session-factory>

</hibernate-configuration>

**Entity Class without annotations** –eg: @Entity, @ Table, @Column

package javabeat.net.hibernate;

public class Employee {

private int id;

private String empName;

private String branch;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getEmpName() {

return empName;

}

public void setEmpName(String empName) {

this.empName = empName;

}

public String getBranch() {

return branch; }

public void setBranch(String branch) {

this.branch = branch;

}

}

**Hibernate Mapping File (Employee.hbm.xml)**

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="javabeat.net.hibernate">

<class name="Employee" table="employee">

<id name="id" column="id">

<generator class="increment"/>

</id>

<property name="empName" column="empname"/>

<property name="branch"/>

</class>

</hibernate-mapping>

AOP with Spring Framework

One of the key components of Spring Framework is the **Aspect oriented programming (AOP)** framework. Aspect Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called **cross-cutting concerns** and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, data validation and caching etc.

The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. **Dependency Injection helps you decouple your application objects from each other and AOP helps you decouple cross-cutting concerns from the objects that they affect**. AOP is like triggers in programming languages such as Perl, .NET, Java and others.

Spring AOP module provides interceptors to intercept an application, for example, when a method is executed, you can add extra functionality before or after the method execution.

## AOP Terminologies:

Before we start working with AOP, let us become familiar with the AOP concepts and terminology. These terms are not specific to Spring, rather they are related to AOP.

|  |  |
| --- | --- |
| **Terms** | **Description** |
| Aspect | A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. |
| Join point | This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework. |
| Advice | This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework. |
| Pointcut | Pointcut are expressions that is matched with join points to determine whether advice needs to be executed or not. |
| Introduction | An introduction allows you to add new methods or attributes to existing classes. |
| Target object | The object being advised by one or more aspects, this object will always be a proxied object. Also referred to as the advised object. |
| Weaving | Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. |

## Types of Advice

Spring aspects can work with five kinds of advice mentioned below:

|  |  |
| --- | --- |
| **Advice** | **Description** |
| before | Run advice before the a method execution. |
| after | Run advice after the a method execution regardless of its outcome. |
| after-returning | Run advice after the a method execution only if method completes successfully. |
| after-throwing | Run advice after the a method execution only if method exits by throwing an exception. |
| around | Run advice before and after the advised method is invoked. |

XML Schema Based AOP with Spring

[**http://www.tutorialspoint.com/spring/schema\_based\_aop\_appoach.htm**](http://www.tutorialspoint.com/spring/schema_based_aop_appoach.htm)

<?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:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

<aop:config>

<aop:aspect id="log" ref="logging">

<aop:pointcut id="selectAll"

expression="execution(\* com.tutorialspoint.\*.\*(..))"/>

<aop:before pointcut-ref="selectAll" method="beforeAdvice"/>

<aop:after pointcut-ref="selectAll" method="afterAdvice"/>

<aop:after-returning pointcut-ref="selectAll"

returning="retVal"

method="afterReturningAdvice"/>

<aop:after-throwing pointcut-ref="selectAll"

throwing="ex"

method="AfterThrowingAdvice"/>

</aop:aspect>

</aop:config>

<!-- Definition for student bean -->

<bean id="student" class="com.tutorialspoint.Student">

<property name="name" value="Zara" />

<property name="age" value="11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id="logging" class="com.tutorialspoint.Logging"/>

</beans>

<aop:pointcut> selects all the methods defined under the package com.tutorialspoint. Let us suppose, you want to execute your advice before or after a particular method, you can define your pointcut to narrow down your execution by replacing stars (\*) in pointcut definition with actual class and method names. Below is a modified XML configuration file to show the concept:

<?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:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">

<aop:config>

<aop:aspect id="log" ref="logging">

<aop:pointcut id="selectAll"

expression="execution(\* com.tutorialspoint.Student.getName(..))"/>

<aop:before pointcut-ref="selectAll" method="beforeAdvice"/>

<aop:after pointcut-ref="selectAll" method="afterAdvice"/>

</aop:aspect>

</aop:config>

<!-- Definition for student bean -->

<bean id="student" class="com.tutorialspoint.Student">

<property name="name" value="Zara" />

<property name="age" value="11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id="logging" class="com.tutorialspoint.Logging"/>

</beans>

@AspectJ Based AOP with Spring

[**http://www.tutorialspoint.com/spring/aspectj\_based\_aop\_appoach.htm**](http://www.tutorialspoint.com/spring/aspectj_based_aop_appoach.htm)

The following example defines a pointcut named 'businessService' that will match the execution of every method available in the classes under the package com.xyz.myapp.service:

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.xyz.myapp.service.\*.\*(..))") // expression

private void businessService() {} // signature

The following example defines a pointcut named 'getname' that will match the execution of getName() method available in Student class under the package com.tutorialspoint:

import org.aspectj.lang.annotation.Pointcut;

@Pointcut("execution(\* com.tutorialspoint.Student.getName(..))")

private void getname() {}

## Declaring advices

You can declare any of the five advices using @{ADVICE-NAME} annotations as given below. This assumes that you already have defined a pointcut signature method businessService():

@Before("businessService()")

public void doBeforeTask(){

...

}

@After("businessService()")

public void doAfterTask(){

...

}

@AfterReturning(pointcut = "businessService()", returning="retVal")

public void doAfterReturnningTask(Object retVal){

// you can intercept retVal here.

...

}

@AfterThrowing(pointcut = "businessService()", throwing="ex")

public void doAfterThrowingTask(Exception ex){

// you can intercept thrown exception here.

...

}

@Around("businessService()")

public void doAroundTask(){

...

}

You can define you pointcut inline for any of the advices. Below is an example to define inline pointcut for before advice:

@Before("execution(\* com.xyz.myapp.service.\*.\*(..))")

public doBeforeTask(){

...

}

**Bean Scopes**

|  |  |
| --- | --- |
| **Scope** | **Description** |
| singleton | This scopes the bean definition to a single instance per Spring IoC container (default). |
| prototype | This scopes a single bean definition to have any number of object instances. |
| request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

**Spring Annotations:**

[**http://www.techferry.com/articles/spring-annotations.html**](http://www.techferry.com/articles/spring-annotations.html)

### @RequestBody and @ResponseBody

If you annotate a method with @ResponseBody, spring will try to convert its return value and write it to the http response automatically. If you annotate a methods parameter with @RequestBody, spring will try to convert the content of the incoming request body to your parameter object on the fly.

**@Controller**

**@RequestMapping**(value = "/bookcase")

**public** **class** **BookCaseController** {

**private** BookCase bookCase;

**@RequestMapping**(method = RequestMethod.GET)

**@ResponseBody**

**public** BookCase **getBookCase**() {

**return** **this**.bookCase;

}

**@RequestMapping**(method = RequestMethod.PUT)

**@ResponseStatus**(HttpStatus.NO\_CONTENT)

**public** **void** **setBookCase**(**@RequestBody** BookCase bookCase) {

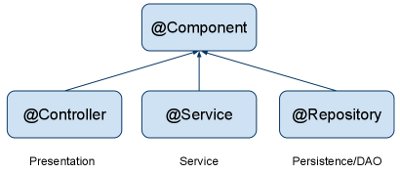
**this**.bookCase = bookCase;

}

}

**@Transactional**

When you use @transactional with the proper Spring configuration, Spring will recognize that the method needs an transaction and will handle the transaction creation, commit and close for you.



@Service

Business logic

@Repository

Database related code

@Controller

For code in the presentation layer.

**@Qualifier**

There may be a situation when you create more than one bean of the same type and want to wire only one of them with a property, in such case you can use **@Qualifier** annotation along with **@Autowired** to remove the confusion by specifying which exact bean will be wired. Below is an example to show the use of @Qualifier annotation.