

Tejaswini Mandar Jog

# Learning Spring 5.0

Build, test, and secure robust enterprise-grade  
applications using the Spring Framework

This book is based on Spring Version 5.0 RC1



Packt

**Contents**

---

- [1: Spring at Glance](#)
  - [b'Chapter 1: Spring at Glance'](#)
  - [b'Introduction to Spring framework'](#)
  - [b'Problems addressed by Spring'](#)
  - [b'Spring Architecture'](#)
  - [b'What more Spring supports underneath?'](#)
  - [b'Spring road map'](#)
  - [b'Container-The heart of Spring'](#)
  - [b'How beans are available from container?'](#)
  - [b'Summary'](#)
- [2: Dependency Injection](#)
  - [b'Chapter 2: Dependency Injection'](#)
  - [b'The life of a bean'](#)
  - [b'Using JSR-250 annotations for bean lifecycle'](#)
  - [b'Instance creation'](#)
  - [b'Dependency Injection'](#)
  - [b'Summary'](#)
- [3: Accelerate with Spring DAO](#)
  - [b'Chapter 3: Accelerate with Spring DAO'](#)
  - [b'How Spring handles database?'](#)
  - [b'Object Relation Mapping'](#)
  - [b'Summary'](#)
- [4: Aspect Oriented Programming](#)
  - [b'Chapter 4: Aspect Oriented Programming'](#)
  - [b'Aspect Oriented Programming \(AOP\)'](#)
  - [b'Part I : Creating application for the core concern\(JDBC\)'](#)
  - [b'PartII: Integration of Log4J'](#)
  - [b'Part III: Writing Logging aspect.'](#)
  - [b'Annotation based aspect.'](#)
  - [b'Introduction'](#)
- [5: Be Consistent: Transaction Management](#)
  - [b'Chapter 5: Be Consistent: Transaction Management'](#)
  - [b'Life cycle of transaction management'](#)
  - [b'Summary'](#)
- [6: Explore Spring MVC](#)
  - [b'Chapter 6: Explore Spring MVC'](#)
  - [b"](#)

- [b"](#)
- [b'Summary'](#)
- [7: Be assured take a test drive](#)
  - [b'Chapter 7: Be assured take a test drive'](#)
  - [b"Testing' an important step'](#)
  - [b'Testing Tools'](#)
  - [b'Pase I Unit testingDAO Unit testing by JUnit'](#)
  - [b'Mock Testing'](#)
  - [b'Pase II Integration testing'](#)
  - [b'Pase III System testing'](#)
  - [b'Summary'](#)
- [8: Explore the Power of Restful Web Services](#)
  - [b'Chapter 8: Explore the Power of Restful Web Services'](#)
  - [b'Web services'](#)
  - [b'Summary'](#)
- [9: Exchange the Message: The Messaging](#)
  - [b'Chapter 9: Exchange the Message: The Messaging'](#)
  - [b'Spring and Messaging'](#)
  - [b'Overview of WebSocket API'](#)
  - [b'SockJS'](#)
  - [b'STOMP'](#)
  - [b'Summary'](#)

# Chapter 1. Spring at Glance

*Spring the fresh new start after the winter of traditional J2EE*, is what Spring framework is in actual. A complete solution to the most of the problems occurred in handling the development of numerous complex modules collaborating with each other in a Java enterprise application. Spring is not a replacement to the traditional Java Development but it is a reliable solution to the companies to withstand in today's competitive and faster growing market without forcing the developers to be tightly coupled on Spring APIs.

In this topic, we will be going through the following points:

- Introduction to Spring framework
- Problems address by Spring in enterprise application development
- Spring road map
- What's new in Spring 5.0

# Introduction to Spring framework

---

Rod Johnson is an Australian computer specialist and co-founder of SpringSource. "Expert One on One J2EE Design and Development" was published in November 2002 by him. This book contains about 30000 lines of code, which contains the fundamental concepts like **Inversion of Control (IoC)**, **Dependency Injection (DI)** of the framework. This code is referred as interface21. He wrote this code with just an intension to be used by developers to simplify their work, or they can use this as basis of their own development. He never thought of any framework development or anything like that. There happened to be a long discussion at Wrox Forum about the code, its improvement and lot many things. Juregen Holler and Yann Caroffa, were the two readers of the forum who proposed the thought of making the code a base of a new framework. This is the reasoning of Yann, *Spring the fresh new start after Winter of traditional J2EE* who names the framework as The Spring framework. The project went in public in June 2003 and powered towards 1.0. Then onwards lots of changes and up gradations took place to withstand and support the technologies in market. We aim in this book about the latest version 5.0. In couple of pages we will cover what are the new features added in this version. In subsequent pages we will cover how to use the latest features in your application and how as a developer you can take advantages of.

# **Problems addressed by Spring**

---

Java Platform is long term, complex, scalable, aggressive, and rapidly developing platform. The application development takes place on a particular version. The applications need to keep on upgrading to the latest version in order to maintain recent standards and cope up with them. These applications have numerous classes which interact with each other, reuse the APIs to take their fullest advantage so as to make the application is running smoothly. But this leads to some very common problems of as.

## **Scalability**

The growth and development of each of the technologies in market is pretty fast both in hardware as well as software. The application developed, couple of years back may get outdated because of this growth in these areas. The market is so demanding that the developers need to keep on changing the application on frequent basis. That means whatever application we develop today should be capable of handling the upcoming demands and growth without affecting the working application. The scalability of an application is handling or supporting the handling of the increased load of the work to adapt to the growing environment instead of replacing them. The application when supports handling of increased traffic of website due to increase in numbers of users is a very simple example to call the application is scalable. As the code is tightly coupled, making it scalable becomes a problem.

## **Plumbing code**

Let's take an example of configuring the DataSource in the Tomcat environment. Now the developers want to use this configured DataSource in the application. What will we do? Yes, we will do the JNDI lookup to get the DataSource. In order to handle JDBC we will acquire and then release the resources in `try catch`. The code like `try catch` as we discuss here, inter computer communication, collections too necessary but are not application

specific are the plumbing codes. The plumbing code increases the length of the code and makes debugging complex.

## **Boiler plate code**

How do we get the Connection while doing JDBC? We need to register Driver class and invoke the `getConnection()` method on DriverManager to obtain the connection object. Is there any alternative to these steps? Actually NO! Whenever, wherever we have to do JDBC these same steps have to repeat every time. This kind of repetitive code, block of code which developer write at many places with little or no modification to achieve some task is called as Boilerplate code. The boiler plate code makes the Java development unnecessarily lengthier and complex.

## **Unavoidable non-functional code**

Whenever application development happens, the developer concentrate on the business logic, look and feel and persistency to be achieved. But along with these things the developers also give a rigorous thought on how to manage the transactions, how to handle increasing load on site, how to make the application secure and many more. If we give a close look, these things are not core concerns of the application but still these are unavoidable. Such kind of code which is not handling the business logic (functional) requirement but important for maintenance, trouble shooting, managing security of an application is called as non-functional code. In most of the Java application along with core concerns the developers have to write down non-functional code quite frequently. This leads to provide biased concentration on business logic development.

## **Unit testing of the application**

Let's take an example. We want to test a code which is saving the data to the table in database. Here testing the database is not our motive, we just want to be sure whether the code which we have written is working fine or not. Enterprise Java application consists of many classes, which are

interdependent. As there is dependency exists in the objects it becomes difficult to carry out the testing.

Spring, mainly addresses these problematic areas and provide a very powerful yet easy solution with,

## **POJO based development**

The class is a very basic structure of application development. If the class is getting extended or implementing an interface of the framework, reusing it becomes difficult as they are tightly coupled with API. The **Plain Old Java Object (POJO)** is very famous and regularly used terminology in Java application development. Unlike Struts and EJB Spring doesn't force developers to write the code which is importing or extending Spring APIs. The best thing about Spring is that developers can write the code which generally doesn't have any dependencies on framework and for this, POJOs are the favorite choice. POJOs support loosely coupled modules which are reusable and easy to test.

### **Note**

The Spring framework is called to be non-invasive as it doesn't force the developer to use API classes or interfaces and allows to develop loosely coupled application.

## **Loose coupling through DI**

Coupling, is the degree of knowledge in class has about the other. When a class is less dependent on the design of any other class, the class will be called as loosely coupled. Loose coupling can be best achieved by **interface programming**. In the Spring framework, we can keep the dependencies of the class separated from the code in a separate configuration file. Using interfaces and dependency injection techniques provided by Spring, developers can write loosely coupled code (Don't worry, very soon we will discuss about Dependency Injection and how to achieve it). With the help of

loose coupling one can write a code which needs a frequent change, due to the change in the dependency it has. It makes the application more flexible and maintainable.

## **Declarative programming**

In declarative programming, the code states what is it going to perform but not how it will be performed. This is totally opposite of imperative programming where we need to state stepwise what we will execute. The declarative programming can be achieved using XML and annotations. Spring framework keeps all configurations in XML from where it can be used by the framework to maintain the lifecycle of a bean. As the development happened in Spring framework, the 2.0 onward version gave an alternative to XML configuration with a wide range of annotations.

## **Boilerplate code reduction using aspects and templates**

We just have discussed couple of pages back that repetitive code is boilerplate code. The boiler plate code is essential and without which providing transactions, security, logging etc will become difficult. The framework gives solution of writing Aspect which will deal with such cross cutting concerns and no need to write them along with business logic code. The use of Aspect helps in reduction of boilerplate code but the developers still can achieve the same end effect. One more thing the framework provides, is the templates for different requirements. The JDBCTemplate, HibernateTemplate are one more useful concept given by Spring which does reduction of boilerplate code. But as a matter of fact, you need to wait to understand and discover the actual potential.

## **Layered architecture**

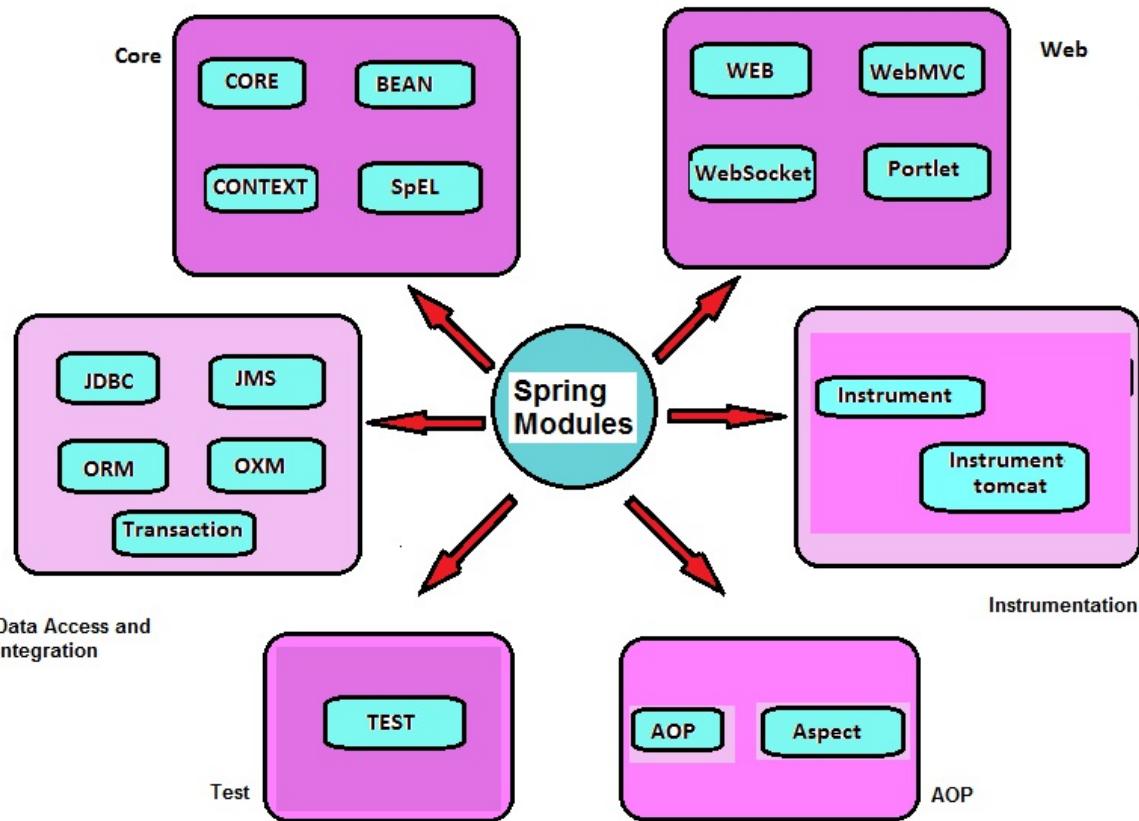
Unlike Struts and Hibernate which provides web persistency solutions respectively, Spring has a wide range of modules for numerous enterprise development problems. This layered architecture helps the developer to choose any one or more of the modules to write solution for his application in

a coherent way. E.g. one can choose Web MVC module to handle web request efficiently without even knowing that there are many other modules available in the framework.

# Spring Architecture

---

Spring provides more than 20 different modules which can be broadly summaries under 7 main modules which are as follows:



Spring modules

## Core modules

### Core

Spring Core module supports ways of creating Spring beans and injecting the dependencies in beans. It provide means to configure the beans and how to

obtain the configured beans from the Spring container using `BeanFactory` and `ApplicationContext` for developing standalone application

## **Beans**

The Beans module provides `BeanFactory` which provides alternative for programmatic singletons. The `BeanFactory` is an implementation of factory design pattern.

## **Context**

This module supports Java enterprise features such as EJB, JMX and basic remoting. It supports for integration of third party libraries for caching, Java Mailing and templating engines like Velocity.

## **SpEL**

Spring Expression Language (SpEL) is an extension of unified Expression Language which has been specified in JSP 2.1 specifications. SpEL module supports setting and getting of property values, configuring collections using logical as well as arithmetic operators, named variables from Spring IoC.

## **Data access and integration modules**

### **JDBC(DAO)**

This module provides abstraction layer on JDBC. It supports reduction of boiler plate code which occurs in getting connection object via loading of driver, getting statement object and many more. It also supports templates as `JdbcTemplate`, `HibernateTemplate` to simplify the development.

### **ORM**

The Object Relational Mapping (ORM) module supports integration of very popular frameworks like Hibernate, iBATIS, Java Persistence API(JPA), Java

Data Object(JDO).

## **OXM**

The Object XML Mapper (OXM) module supports object to XML mapping and integration for JAXB, castor, XStream etc.

## **JMS**

This module provides support and provides Spring abstract layer over Java Message Service(JMS)for asynchronous integration with other applications via messaging.

## **Transaction**

JDBC and ORM modules handle exchange of data to-and-fro between Java application and database. This module supports transaction management support while working with ORM and JDBC modules.

## **Web MVC and remoting modules**

### **Web**

This module supports integration of web application created in other frameworks. Using this module the developers can also develop web application using Servlet listener. It supports multipart file uploading and handling of request and response. It also provides web related remoting support.

### **Servlet**

This module contains Spring Model View Controller(MVC) implementation for web applications. Using Spring MVC developers can write handling of request and response to develop full-fledged web application. It helps in getting rid from the boiler plate code while handling request and response by

supporting handling form submission.

## Portlet

The Portlet module provides MVC implementation to be used in Portlet environment which support Java's portlet API.

### Note

The Portlet has been removed in Spring 5.0M1. If you want to use Portlet you need to use with 4.3 module.

## WebSocket

WebSocket is a protocol which provides two way communications between client and server which has been included in Spring 4. This module provides support for integration of Java WebSocket API in the application.

### Note

**struts module** This module contains supports for integrating Struts framework within Spring application. But this has been deprecated in Spring 3.0

## AOP modules

### AOP

The Aspect Oriented Programming module helps in handling and managing the cross cutting concern services in the application and helps in keeping the code cleaner.

### Aspects

This module provides integration support with AspectJ.

## Instrumentation modules

### Instrumentation

Java Instrumentation gives an innovative way to access a class from JVM with the help of class loader and modify its byte code by inserting the custom code. This module supports instrumentation and class loader implementations for some application servers.

### Instrument Tomcat

Instrument Tomcat module contains Spring instrumentation support for Tomcat.

### Messaging

The messaging module provides support for STOMP as websocket protocol. It also has annotations for routing and processing STOMP messages received from the clients.

- Spring messaging module has been included in Spring 4.

## Test module

The Test module support unit as well as integration testing with JUnit and TestNG. It also provides support for creating mock objects to simplify testing in isolated environment.

# **What more Spring supports underneath?**

---

## **Security module**

Now a days the applications alone with basic functionalities also need to provide sound ways to handle security at different levels. Spring5 support declarative security mechanism using Spring AOP.

## **Batch module**

The Java Enterprise Applications needs to perform bulk processing, handling of large amount of data in many business solutions without user interactions. To handle such things in batches is the best solution available. Spring provides integration of batch processing to develop robust application.

## **Spring integration**

In the development of enterprise application, the application may need interaction with them. Spring integration is extension of the core spring framework to provide integration of other enterprise applications with the help of declarative adapters. The messaging is one of such integration which is extensively supported by Spring.

## **Mobile module**

The extensive use of mobiles opens the new doors in development. This module is an extension of Spring MVC which helps in developing mobile web applications known as Spring Android Project. It also provide detection of the type of device which is making the request and accordingly renders the views.

## **LDAP module**

The basic aim of Spring was to simplify the development and to reduce the boilerplate code. The Spring LDAP module supports easy LDAP integration using template based development.

## **.NEW module**

The new module has been introduced to support .NET platform. The modules like ADO.NET, NHibernate, ASP.NET has been in the .NET module includes to simplify the .NET development taking the advantages of features as DI, AOP, loose coupling.

# **Spring road map**

---

## **1.0 March2004**

It supports for JDO1.0 and iBATIS 1.3 with integrated with Spring transaction management. This version was supporting the functionalities as, Spring Core, Spring Context, Spring AOP, Spring DAO, Spring ORM and Spring web.

## **2.0 October 2006**

Spring framework enhanced support for Java5. It added out of box namespaces like jee, tx, aop, lang, util to simplify the configuration. The IoC was supporting scopes as singleton and prototype. In addition to these scopes, scopes for HttpSession, Cluster cache and request has been also introduced. The annotation bases configuration as @Transactional, @Required, @PersistenceContext introduced.

## **2.5 November 2007**

In this version Spring supports full Java6 and JavaEE5 features as JDBC4, JavMail1.4, JTA1.1, JAX WS 2.0. It also extends the support for annotation based DI including support for qualifier as well. A new bean named pointcut element in AspectJ pointcut expressions has been introduced. The build in support for AspectJ for load time weaving which is based on LoadTimeWeaver abstraction has been provided. For the convenience an introduction of custom namespaces like context, jms has been included. The testing support extended for Junit4 and TestNG. The annotation based SpringMVC controllers has been added. It also supports for auto detection of components on the classpath such as @Repository,@Service, @Controller and @Conponent. Now SimpleJdbcTemplate supports named SQL parameters. The certified WebSphere support has been included. It also

include support for JSR-250 annotations like @Resource, PostConstruct, @PreDestroy

## **3.0 GA December 2009**

The entire code has been revised to support Java5 feature like generics, varargs. The Spring Expression Language(SpEL) has been introduced. It also supports for annotation for REST web application. It extends support for many Java EE6 features like JPA 2.0, JSF 2.0. The version 3.0.5 support hibernate 3.6 final as well.

## **3.1GA December 2011**

In this version Testing support has been upgraded for Junit 4.9. It also supports load time weaving on the WebSphere version 7 and 8.

## **4.0 December 2013**

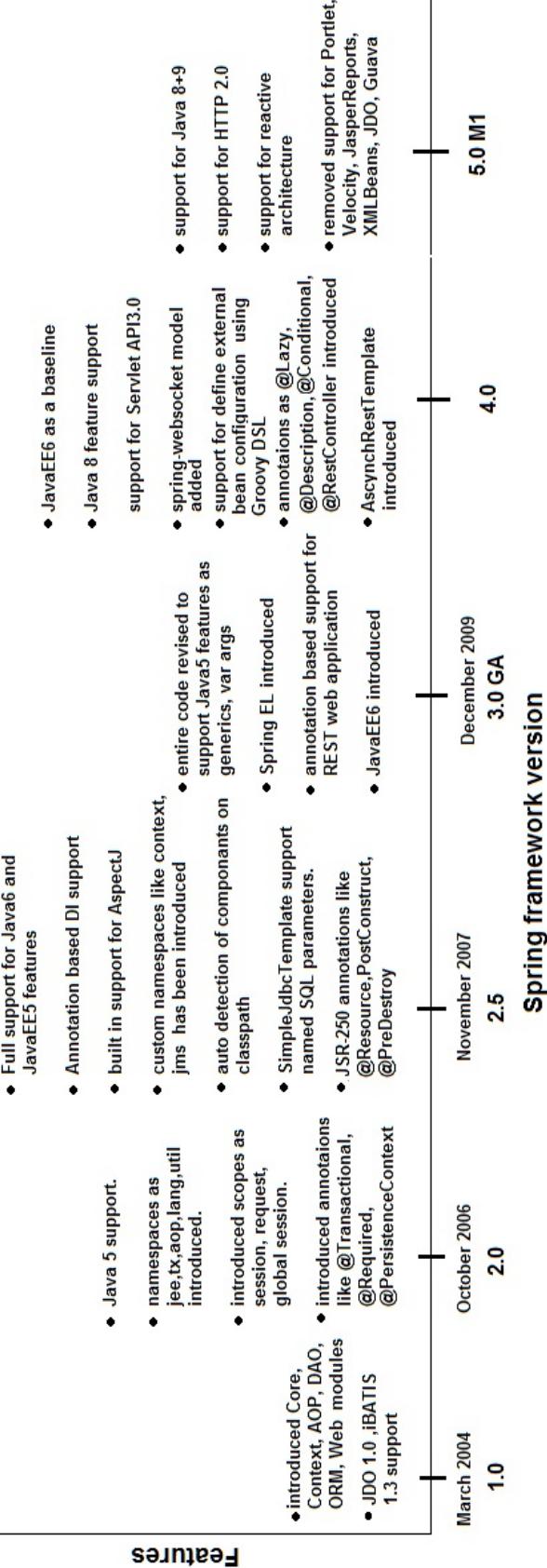
For very first time the full support for java 8 features has been included. This version uses JavaEE 6 as its baseline. Using Spring 4, now it is possible to define external bean configuration using Groovy DSL. Developers now can treat generic types as a form of qualifier. @Lazy annotation can be used on injection points as well as on @Bean definitions. The @Description has been introduced for developers using Java Based configuration. The @Conditional annotation has been introduced for conditional filtering. Now, there is no requirement to have default constructor to be used byCGLIB based proxy classes. The @RestController has been introduced to remove need of @ResponseBody to each of @RequestMapping, The AsyncRestTemplate has been included which allows non blocking asynchronous support for REST client development. The spring-websocket introduced as new model to provide support for WebSocket based two way communication between server and client. The spring- messaging module has been introduced for the support of WebSocket sub protocol STOMP. Most of the annotations from spring-test module can now be used as meta annotations to create custom composed annotations. The set of the mocks from

`org.springframework.mock.web` is based on Servlet API 3.0

## 5.0 M1 Q4 2016

Spring 5M1 will support Java8+ but basically, it aims to track and support greatly to the new bee Java9. It also will support reactive programming Spring 5 will focus on HTT2.0. It also aims to focus on reactive programming through reactive architecture. The `mock.staticmock` from `spring-aspects`, `web.view.tiles2` has been dropped. No more support for Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava.

It can be summarized as shown in the following figure:



Spring modules

# **Container-The heart of Spring**

---

POJO development is the backbone of Spring framework. The POJO configured in the and whose object instantiation, object assembly, object management is done by Spring IoC container is called as bean or Spring bean. We use Spring IoC as it on the pattern of Inversion of Control.

## **Inversion of Control (IoC)**

In every Java application, the first important thing which each developer does is, to get an object which he can use in the application. The state of an object can be obtained at runtime or it may be at compile time. But developers creates object where he use boiler plate code at a number of times. When the same developer uses Spring instead of creating object by himself he will be dependent on the framework to obtain object from. The term inversion of control comes as Spring container inverts the responsibility of object creation from developers.

Spring IoC container is just a terminology, the Spring framework provides two containers

- The BeanFactory
- The ApplicationContext

## **The BeanFactory-The history**

The BeanFactory container provides the basic functionalities and framework configuration. Now a days, developers won't prefer to use BeanFactory. Now the obvious question comes to your mind then why BeanFactory is still in framerwork? Why has it not been removed? If not BeanFactory, then what's the alternative? Let's answer them one by one. The very simple answer of BeanFactory in framework is to support for backward compatibility of JDK1.4. The beanFactory provides BeanFactoryAware, InitializingBean,

DisposableBean interfaces to support backward compatibility for third party framework which has integration with Spring.

## XMLBeanFactory

Today's enterprise application development demands much more than ordinary development. The Developer will be happy to get a helping hand for managing the object life cycle, injecting the dependencies or reduction in boilerplate code from the IoC container. XMLBeanFactory is a common implementation of BeanFactory.

Let's find out practically how the BeanFactory container get initialized:

1. Create a Java application with the name

Ch01\_Container\_Initialization.

2. Add the jars as shown in the following snapshot:



Jars to be added

## Note

Make sure that you are using JRE to 1.8 as it's a baseline for Spring5.0.0.M1. You can download the jars from.....

3. Create a class `TestBeanFactory` under the package `com.ch01.test` package.
4. Create a XML file `beans_classpath.xml` in the classpath where we can write bean definitions later. Each beans definition file contains the referencing schema to beans.xsd of the particular Spring version. The root tag of this XML file will be `<beans>`.

Basic structure of the XML file is as follows:

```
<?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/
        http://www.springframework.org/schema/beans/spring-bean.xsd">
</beans>
```

Our XML file contains the same code as shown above without any beans configured.

5. In the main function lets' write down the code to initialize the bean factory as shown:

```
BeanFactory beanFactory=new XmlBeanFactory(
    new ClassPathResource("beans_classpath.xml"));
```

Here, the `bean_classpath.xml` will contain the beans definitions (For simplicity, we haven't added any bean definition, we will see it in detail in next chapter). The `ClassPathResource` loads the resource from the classpath.

6. Sometimes the resource will not be in the classpath and it will be in the filesystem. The following code can be used to load the resource from filesystem:

```
BeanFactory beanFactory=new XmlBeanFactory(
    new FileSystemResource("d:\\beans_fileSystem.xml"));
```

7. We need to create `bean_fileSystem.xml` on D drive which will contain the same content as that of `bean_classpath.xml`. The complete code will be as follows:

```
public class TestBeanFactory {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        BeanFactory beanFactory=new XmlBeanFactory(new
            ClassPathResource("beans_classpath.xml"));
        BeanFactory beanFactory1=new XmlBeanFactory(new
            FileSystemResource("d:\\beans_fileSystem.xml"));
        System.out.println("beanfactory created successfully");
    }
}
```

There will not be any output on console apart from logging information of spring container as we haven't written any output code here. But the following snapshot shows the XML file loads and the container got initialized:

```
Aug 8, 2016 10:48:32 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beans_classpath.xml]
Aug 8, 2016 10:48:32 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [d:\beans_fileSystem.xml]
beanfactory created successfully
```

The console logger output

## Note

BeanFactory doesn't support multiple configuration files.

## The ApplicationContext: The present

The registration of BeanProcessor and BeanFactoryPostProcessor which plays important role in AOP and property place holders, needs the explicit code writing which makes it inconvenient to work with. Developers don't want to write the code which supports internationalization. The event publication to handle AOP integration is unavoidable. The web application needs to have application layer specific context. To all of these the simple solution is to expand the services provided by BeanFactory with ApplicationContext. The ApplicationContext is not replacement of BeanFactory but it's an extension for enterprise specific solutions and more advance mechanism for bean configuration.

Let's look at the implementations.

### ClassPathXmlApplicationContext

The subclass of AbstractXmlApplicationContext is used for Satndalone applications. It uses bean configured XML file from the class path. If the conditions of having more than one XML configuration files later bean definition from the XML file will override the earlier bean definition. It

provides the advantage of writing new bean definition to replace the previous one.

Let's find out practically how the `ClassPathXmlApplicationContext` container gets initialized. We will use the same `Ch01_Container_Initialization` project by following the steps as:

1. Create a class `TestClasspathApplicationContext` under the package `com.ch01.test` package.
2. Create a new XML file `beans_classpath.xml` in classpath as we had created in previous application.
3. In the main function let's write down the code to initialize the bean factory as shown in the following code:

```
try {
    ApplicationContext context=new
        ClassPathXmlApplicationContext("beans_classpath.xml");
    System.out.println("container created successfully");
}
catch (BeansException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

No need to create XML file as we already had created it for the previous example. `ClassPathXmlApplicationContext` loads the `bean_classpath.xml` file from the classpath which contains the beans definitions (For simplicity we haven't added any bean definition, we will see it in detail in next chapter).

4. Run the application which will give the following output suggesting the container created successfully:

```
Aug 15, 2016 10:21:02 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beans_classpath.xml]
container created successfully
```

Console output

5. In Java enterprise application, the project can have multiple configuration files as it's easy to maintain and support modularity as

well. To load multiple bean configuration files we can use the following code:

```
try {
    ApplicationContext context1 = new
        ClassPathXmlApplicationContext
        (new String[]
        {"beans_classpath.xml","beans_classpath1.xml" });
}
catch (BeansException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

To use the preceding code of lines we need to create beans\_classpath1.xml in classpath.

## FileSystemXmlApplicationContext

Similar to ClassPathXmlApplicationContext this class also extends AbstractXmlApplicationContext and is used for standalone applications. But this class helps to load the bean XML definition from the file system. The file path it relative to the current working directory. In case of specifying the absolute file path one can use file: as prefix. It also provides the advantage of writing new bean definition to replace the previous one in case of having multiple XML configurations.

Let's find out practically how the ClassPathXmlApplicationContext container gets initialized. We will use the same Ch01\_Container\_Initialization project by following the steps as follows:

1. Create a class TestFileSystemApplicationContext under the package com.ch01.test package.
2. Create a new XML file beans\_fileSystem.xml in D drive we had created in previous application.
3. In the main function, let's write down the code to initialize the bean factory as shown in the following code:

```
try {
    ApplicationContext context=new
```

```

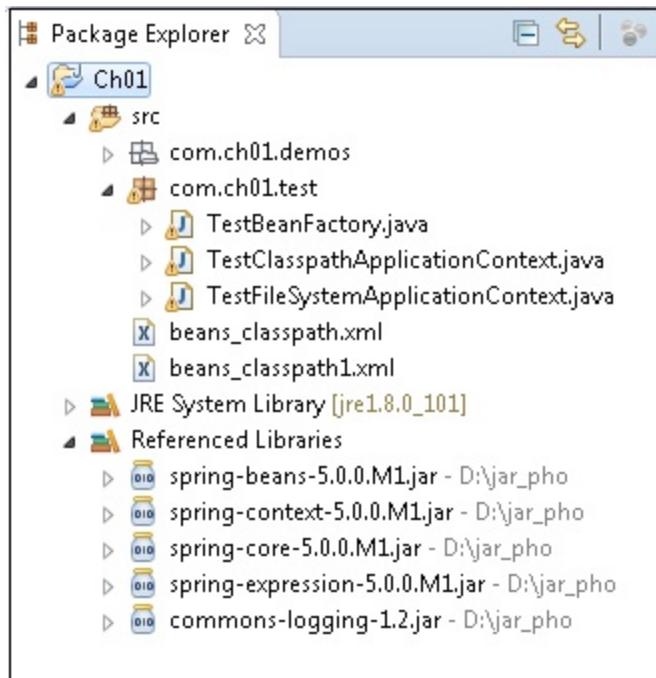
        FileSystemXmlApplicationContext
        ("d:\\beans_fileSystem.xml");
    System.out.println("container created successfully");
}
catch (BeansException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

```

FileSystemXmlApplicationContext loads the bean\_fileSystem.xml file from the path specified.

4. Run the application which will give the following output suggesting the container created successfully.

The structure of project discussed above will be as shown in the following snapshot:



Project directory structure

## WebXmlApplicationContext

The AbstractRefreshableWebApplicationContext has been extended by

WebXmlApplicationContext. We can write the context definition in related to root application context in `applicationContext.xml` and keep it under WEB-INF as its the default location from where the context definition will be loaded. The `XXX-servlet.xml` file is loaded to load the controller definition as in case of MVC we application. Also, we can override the default locations by configuring `contextConfigLocation` for the `context-param` and `init-param`.

# How beans are available from container?

---

Yes, beans or object of beans will not be available without doing anything from development side. Spring manages bean but what to manage has to be decided and pass on to the container. Spring supports declarative programming via XML file configuration. The beans definitions configured in XML file loaded by the container and using org.springframework.beans the object instantiation and the property value injection takes place. Bean lifecycle explain the stages, phases or activities through which each bean object goes through from making the object usable by application till its cleaned up and removed from the container when application doesn't required by the application. We will discuss in next chapter the detail initialization process.

# Summary

---

This chapter gives an overview of Spring framework. We discussed about the general problems faced in Java enterprise application development and how they have been address by Spring framework. We have seen the overall major changes happened in each version of Spring from its first introduction in market. The backbone of Spring framework is the bean. We use Spring to simplify the work of managing them by the container. We discuss in detail about two Spring containers BeanFactory and ApplicationContext and how they can be used by the developers. The containers are involved in process of bean lifecycle management. In next chapter we are aiming to discuss in depth about the bean state management with a very famous terminology Dependency Injection and the bean life cycle management in detail.

# Chapter 2. Dependency Injection

The previous chapter gave us over view of what is Spring framework and how it helps the developers to make the development faster and easier. But the question "how to use the framework?" is still unanswered. In this chapter we will discuss the answer in all perspectives and try to find out all the probable answers for it. The chapter is full of configuration and alternatives for the configuration. It all depends on how the developer looks forward with these solutions in available conditions and environmental setup of the application. We are aiming to cover following points in depth.

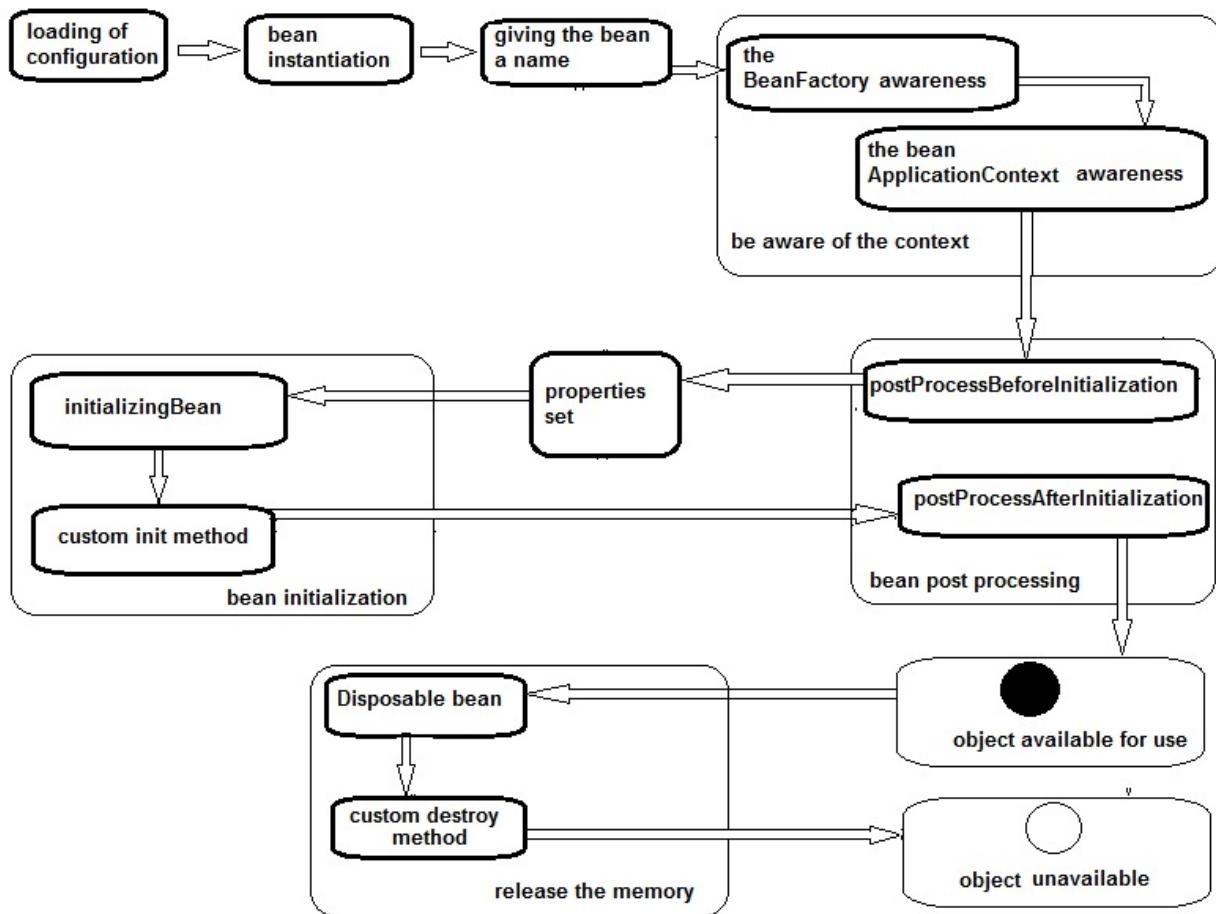
- We will start with Bean Life cycle management with custom initialization, InitializingBean, DisposableBean, and Aware Interfaces, Use of annotations like @PostConstruct and @PreDestroy in bean life cycle
- The dependency injection
- Setter and constructor Dependency injection
- DI for references, inner beans, Inheritance and Collection
- Bean scope and configuring scopes as Singleton or Prototype
- Auto wiring and ways of achieving auto wiring
- Problems occurring while auto wiring and ways to overcome

Lot many things to cover, so let's start with the very first 'the life of a bean'.

# The life of a bean

---

The Spring IoC container hides the complex communication between the container and the beans. The following figure gives an idea about the steps which container follows to maintain the life of every bean:



Bean life cycle

## Loading the configuration

This is the most important phase in bean life cycle which initiates the life cycle process. The container loads and reads the metadata information from

the bean configuration file and starts the next phase 'instantiation'.

## Object creation

Using Java Reflection API the Spring container creates an instance of a bean.

## Setting the bean name

Each bean contains a unique name configured in the configuration. This name can be made available to the bean class by `setBeanName()`. If the bean class implements `BeanNameAware` interface, its `setBeanName()` method gets invoked to set the bean name.

## Setting bean factory

Sometimes the bean class may need to get information about the factory which loaded it. If the bean class implements `BeanFactoryAware` its `setBeanFactory()` method will get invoked passing the instance of the `BeanFactory` to it which is may be an instance of `ApplicationContext`, `WebApplicationContext`,etc.

## Bean post processing with postProcessBeforeInitialization

In some of the scenarios before the values of the objects gets populated some pre initialization is required which cannot be done in the configuration files. In such cases, if an object of `BeanPostProcessor` does this task. The `BeanPostProcessors` are special kind of beans which get instantiates before any other beans are instantiate. These `BeanPostProcessor` beans interact with new instance created by the container. But, it will be done in two steps, once before the properties are set and second once the properties got set. In this phase, `BeanPostProcessor` which is associated with `BeanFactory`, it's `PostProcessorBeforeInitialization` will be called to do the pre initialization.

## **Property population**

The bean configuration may be specified with some bean properties. In this phase all the values will get associated to the instance initialized in the previous phase.

## **Initializing bean with**

### **The afterPropertiesSet() method**

It may happen that, the bean configured in the configuration hasn't set values of all the properties. And once the properties get populated, using some business logic or in some other way rest of the properties need to be set. `InitializingBean` interface helps in the task. If the class implements `InitializingBean` interface, its `afterPropertiesSet()` method will be called to set such properties.

### **The Custom init() method**

Though the `afterProperties()` helps to do initialization of properties based on some logic, the code gets strongly coupled with the Spring API. To overcome this drawback there is a way to initialize the bean using custom initialization method. If the developer has written custom `init` method and configured it for the bean in the bean configuration as an '`init-method`' attribute, it will get called by the container.

## **Bean post processing with postProcessAfterInitialization**

`BeanPostProcessor`'s `postProcessAfterInitialization()` will be called to do the `postProcessing` once the properties got initialized.

## **Use the bean**

Thank god!!!! Yes now the object is perfectly ready for use with its state defined.

## Destruct bean with

The developers used the objects and the objects have completed their tasks. Now we don't need them anymore. To release the memory occupied by the bean can be destroyed by,

### Dispose bean with destroy()

If the bean class implements DisposableBean interface, its destroy() method will be getting called to release memory. It has the same drawback as that of InitializingBean. To overcome we do have custom destroy method.

### Destruction with custom destroy()

It is also possible to write a custom method to release memory. It will be called when the attribute 'destroy-method' has been configured in the bean configuration definition.

- After knowing the lifecycle, let's now do some implementation to know the implementation perspective.

## Case1: Using Custom initialization and destruction methods

As we already discussed in bean life cycle, these two methods will leverage the developer to write their own methods for initialization and destruction. As developers are not coupled with Spring API, they can take advantage of choosing their own method signature.

Let's have a look on how to hook these methods in order to be used by Spring container step by step:

1. Create a Java Application as Ch02\_Bean\_Life\_Cycle and add to it jar which we did in previous project.
2. Create a class Demo\_Custom\_Init under the package com.ch02.beans as shown below:

```

public class Demo_Custom_Init {
    private String message;
    private String name;

    public Demo_Custom_Init() {
        // TODO Auto-generated constructor stub
        System.out.println("constructor gets called for
            initializing data members in Custom init");
        message = "welcome!!!";
        name = "no name";
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return message + "\t" + name;
    }
}

```

3. Add the class a method myInit() with the following code to do initialization. Here we are shifting "name" to uppercase:

```

public void myInit()
{
    name = name.toUpperCase();
    System.out.println("myInit() get called");
}

```

4. Create bean\_lifecycle.xml in class path to configure the bean similar to the previous project(refer to beans\_classpath.xml from Ch01\_Container\_Initialization)
5. Add to it bean definition as follows:

```

<?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.xsd">

<bean id="obj" class="com.ch02.beans.demo_Custom_Init"></
</beans>
```

- \* Each bean has to be configured within <bean> tag.
- \* A <bean> tag contains many attributes we need to configure minimum two of them which are as follows:
  - a. id : Specifies the reference name on which the container recognises whose object he is managing. 'id' must be unique within the container. Naming the 'id' is similar to the reference in Java application.
  - b. class: Specifies whose object container is creating and managing. The value of the class attribute must be fully qualified class name as we did in above configuration.
- The syntax to configure a bean definition in XML is as shown as follows:

```
<bean id="id_to_use" class="fully_qualified_class_name"></b
```

- The XML configuration is equivalent to the Java Code as,

```
Demo_Custom_Init obj= new com.ch02.beans.Demo_Custom_Init(
```

## Note

There are few more attributes which developer can use in configuration. We will see them one by one according to the scenarios in upcoming chapters.

6. The configuration shown in Step 5 is the very basic configuration without providing any information to the container about how to initialize the property 'name'. Let's modify the configuration by adding the attribute 'init-method' to specify the method name which is to be invoked to initialize the property after instantiation. The modified code as shown below:

```
<bean id="obj" class="com.ch02.beans.demo_Custom_Init"
    init-method="myInit">
</bean>
```

7. The way we perform initialize, in the same way we can release the resource as well. To do the release with custom destruct method we need to first add it to the code as:

```
public void destroy()
{
    name=null;
    System.out.println("destroy called");
}
```

8. Configure the destruct method in bean configuration by specifying destroy-method as shown in the following code:

```
<bean id="obj" class="com.ch02.beans.demo_Custom_Init"
    init-method="myInit" destroy-method="destroy">
</bean>
```

9. Create `Test_Demo_Custom_Init` with main function. Initialize the container as we did earlier in chapter 1. And get the instance of `Demo_Custom_Init` using `getBean()` as shown below:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
    ClassPathXmlApplicationContext("beans_lifecycle.xml");
    Demo_Custom_Init obj=(Demo_Custom_Init)context.getBean("o
    System.out.println(obj);
}
```

10. The execution of the code gives the following output:

```
INFO: Loading XML bean definitions from class path resource
[beans_lifecycle.xml]
constructor gets called for initializing data members
myInit() get called
welcome!!! NO NAME
```

The output clearly shows life cycle phases as construction, initialization, use and then destruction of the bean.

Don't be surprised by the absence of 'destroy called' statement. We can use the following code to elegantly shut down the container:

```
((AbstractApplicationContext) context).registerShutdownHook();
```

On addition of the above line to the main function even the 'destroy called' will be seen as a console output.

## Case2: Using InitializingBean to provide initialization

We will use the same project Ch02\_Bean\_Life\_Cycle which we develop in the Case1.

Follow the steps:

1. Add a class Demo\_InitializingBean in com.ch02.beans package which is implementing InitializingBean interface as shown below:

```
public class Demo_InitializingBean implements InitializingB
    private String message;
    private String name;

    public Demo_InitializingBean() {
        // TODO Auto-generated constructor stub
        System.out.println("constructor gets called for
            initializing data members in demo Initializing bean"
        message = "welcome!!!";
        name = "no name";
    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return message + "\t" + name;
    }
}
```

2. Override the method afterPropertiesSet() method for processing properties as follows:

```
@Override
public void afterPropertiesSet() throws Exception {
    // TODO Auto-generated method stub
```

```

        name=""Mr.""+name.toUpperCase();
        System.out.println("after propertiesSet got called");
    }
}

```

3. Add one more bean to the `bean_lifecycle.xml` as follows:

```

<bean id="obj_Initializing"
      class="com.ch02.beans.Demo_InitializingBean"/>

```

You can observe that we don't have to override any init-method attribute here as we did in Case1 as `afterPropertiesSet()` get a call by callback mechanism once properties got set.

4. Create a class `Test_InitializingBean` with main method as shown in the following code:

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
    ClassPathXmlApplicationContext("""beans_lifecycle.xml""");

    Demo_InitializingBean obj=
        (Demo_InitializingBean)context.getBean("""obj_InitializingBean""");
    System.out.println(obj);
}

```

5. The execution of output is as shown below:

```

INFO: Loading XML bean definitions from class path resource
[beans_lifecycle.xml]
constructor gets called for initializing data members in Custom
init
myInit() get called
constructor gets called for initializing data members in demo
after propertiesSet got called
welcome!!! Mr.NO NAME

```

From the above output the underlined statements are not related to the newly configured bean. But as container does initialization of all the beans configured, it will initialize the `Demo_Custom_Init` bean as well.

## **Case3: Using DisposableBean to provide release of**

## **memory**

We will use the same project Ch02\_Bean\_Life\_Cycle which we develop in the Case1.

Follow the steps:

1. Add a class Demo\_DisposableBean.in com.ch02.beans package which is implementing DisposableBean interface as shown below:

```
public class Demo_DisposableBean implements DisposableBean
    private String message;
    private String name;

    public Demo_DisposableBean() {
        // TODO Auto-generated constructor stub
        System.out.println("constructor gets called for
            initializing data members in Disposable Bean");
        message="welcome!!!";
        name="no name";
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return message+"\t"+name;
    }
}
```

2. Override the method destroy()method for memory release as follows:

```
@Override
public void destroy() throws Exception {
    // TODO Auto-generated method stub
    System.out.println("destroy from disposable bean get call
        name=null;
}
```

3. Add one more bean to the bean\_lifecycle.xml as follows:

```
<bean id="obj_Disposable"
    class="com.ch02.beans.Demo_DisposableBean"/>
```

You can observe that we don't have to override any destroy-method attribute here as we did in Case1. The `destroy()` get a callback once the container containing the bean getting shut down.

4. Create a class `Test_DisposableBean` with following code:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
        ClassPathXmlApplicationContext(""+beans_lifecycle.xml")  
  
    Demo_DisposableBean obj=  
        (Demo_DisposableBean)context.getBean("obj_Disposable");  
    System.out.println(obj);  
    ((AbstractApplicationContext)context).registerShutdownHook  
}
```

5. We will get the following code on execution of the main:

```
INFO: Loading XML bean definitions from class path resource  
[beans_lifecycle.xml]  
constructor gets called for initializing data members in Cu  
    init  
myInit() get called  
constructor gets called for initializing data members in de  
after propertiesSet got called  
constructor gets called for initializing data members in  
    Disposable Bean  
welcome!!! no name  
Sep 09, 2016 10:54:55 AM  
org.springframework.context.support.  
ClassPathXmlApplicationContext doClose  
INFO: Closing  
org.springframework.context.support.  
ClassPathXmlApplicationContext@1405ef7: startup date  
[Fri Sep 09 10:54:54 IST 2016]; root of context hierarchy  
destroy from disposable bean get called destroy called
```

The underlined line is from the `destroy()` from Disposable demo but as there is custom `destroy()` method for `Demo_Custom_Init` class as well.

## Case4: Making the bean aware of Container

We will use the same project Ch02\_Bean\_Life\_Cycle which we develop in the Case1.

Follow the steps:

1. Add a class MyBean in com.ch02.contextaware package which is implementing ApplicationContextAware interface.
2. Add a data member to the bean class of type ApplicationContext.
3. Override the method setApplicationContext() method.
4. Add display () to get one of the bean and display its properties. The class will be as shown below:

```
public class MyBean implements ApplicationContextAware {  
    private ApplicationContext context;  
  
    @Override  
    public void setApplicationContext(ApplicationContext ctx)  
        throws BeansException {  
        // TODO Auto-generated method stub  
        System.out.println("context set");  
        this.context=ctx;  
    }  
    public void display()  
    {  
        System.out.println((Demo InitializingBean)  
            context.getBean("obj Initializing"));  
    }  
}
```

Here we are accessing one of the other bean who is not a data member of the class and not injected. But the code shows we still can access their properties.

5. Add one more bean to the `bean_lifecycle.xml` as follows:

```
<bean id=""obj_myBean"" class=""com.ch02.contextAware.MyBea
```

6. Create a class `Test_MyBean` with main method as follows:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
        ClassPathXmlApplicationContext("beans_lifecycle.xml")
```

```

        MyBean obj=(MyBean) context.getBean(""+obj_myBean");
        obj.display();
    }
}

```

7. On execution we will get the following output as follows:

```

constructor gets called for initializing data members in Cu
init
myInit() get called
constructor gets called for initializing data members in de
after propertiesSet got called
constructor gets called for initializing data members in
Disposable Bean
context set
welcome!!! Mr.NO NAME

```

## Case4: Using BeanPostProcessor.

We will use the same project Ch02\_Bean\_Life\_Cycle which we develop in the Case1.

Follow the steps:

1. Add a bean class Demo\_BeanpostProcessor in com.ch02.beans package which implement BeanPostProcessor.
2. Override the method postProcessBeforeInitialization ()method.
3. Override the method postProcessAfterInitialization()method.
4. The complete class definition is as shown below,

```

public class Demo_BeanPostProcessor implements BeanPostProc
{
    @Override
    public Object postProcessBeforeInitialization(Object bean
        String beanName) throws BeansException {
        // TODO Auto-generated method stub
        System.out.println("initializing bean before init:-
            "+beanName);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean,
        String beanName) throws BeansException {

```

```

        // TODO Auto-generated method stub
        System.out.println("initializing bean after init:-"
                "+beanName);
        return bean;
    }
}

```

5. Add one more bean to the `bean_lifecycle.xml` as follows:

```
<bean id=""beanPostProcessor""
      class=""com.ch02.processor.Demo_BeanPostProcessor""/>
```

6. Create a class `TestBeanPostProcessor` with main method. We don't have to ask bean for "beanPostProcessor" as its methods are called before and after init method for each bean in the container.
7. Write the test code to find the order of methods called in initialization process as shown below:

```

public class Test_BeanPostProcessor {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new
            ClassPathXmlApplicationContext("beans_lifecycle.xml")

        Demo_Custom_Init
        obj=(Demo_Custom_Init)context.getBean("obj");
        System.out.println(obj);
    }
}

```

8. The output is as shown below:

```

INFO: Loading XML bean definitions from class path resource
[beans_lifecycle.xml]
initializing bean before init:-
org.springframework.context.event.
internalEventListenerProcessor
initializing bean after init:-
org.springframework.context.event.internalEventListenerProc
initializing bean before init:-
org.springframework.context.event.internalEventListenerFact
initializing bean after init:-
org.springframework.context.event.internalEventListenerFact
constructor gets called for initializing data members in Cu
init

```

```
initializing bean before init:- obj
myInit() get called
initializing bean after init:-obj
constructor gets called for initializing data members in de
initializing bean before init:- obj_Initializing
after propertiesSet got called
initializing bean after init:-obj_Initializing
constructor gets called for initializing data members in
initializing bean before init:- obj_Disposable
initializing bean after init:-obj_Disposable
context set
initializing bean before init:- obj_myBean
initializing bean after init:-obj_myBean
welcome!!! NO NAME
```

The underlined statements are for the bean which we asked from the container. But find the order which has been followed as constructor, `postProcessBeforeInitialization` method, custom-init method, `postProcessAfterInitialization`.

## Note

In an application more than one `BeanPostProcessors` can be configured. The order of their execution can be managed by setting "order" property if the bean implements `Ordered` interface. The scope of each `PostBeanProcessor` is per container.

# Using JSR-250 annotations for bean lifecycle

---

JSR-250 annotations plays a vital role in bean life cycle but we won't directly jump and discover them. Doing so may lead us skipping some of very important concepts. So relax we will discuss them at the time of JSR based annotations. But if you already known Spring, use of annotations in Spring and so eager to go for `@PreDestroy` or `@PostConstruct` you may directly go to the topic JSR annotations.

In huge development of Java Enterprise application is simplified by writing smaller units of codes generally, classes. Then developers reuse them by calling the methods of each other. This is very complex and difficult to maintain architecture. In order to invoke the method of another class, its knowledge is important. The class which holds the object of another class is called as container. And the object which container holds is called as contained object. Now the container is well aware of the contained object. The developers will be more than happy as now they can easily reuse the contained object which simplifies their development. But now there is a very major flaw in the designing. This can be well explained with two very famous terminologies as follows:

- **Loose coupling:** The container class will not be affected even though there is a change in the contained object. The container in such scenario is called as loosely coupled object. The developers always try to write down the code which follows loose coupling. In Java loose coupling can be well achieved with the help of interface programming. The interface states what the contract is? But it doesn't specify who and how the contract will be implemented. The container class will have less of the knowledge of the dependency making it more flexible.
- **Tight coupling:** the container class need to change whenever there is a code change in contained objects. The container is tightly coupled with the container object which makes development difficult.

## Note

Try to avoid writing tightly coupled classes.

Whether it's loose coupling or tight coupling we get reusable objects. Now the question is how these objects will be created. In Java the object creation can happen with two major ways as follows:

- Constructor invocation.
- Factory to give objects

As an abstract level both of these ways looks alike as the end user is going to get an object for use. But these are not the same. In factory the dependant classes have the responsibility of creating the object and in constructor the constructor gets invoked directly. The Java application is centric and revolving around objects. The very first thing every developer tries to get object properly initialized so that handling of data and performing operations can be done in sophisticated way. The instance creation and state initialization are the two steps in creating every properly initialised object. As container will be involving in both of these processes we should have a good knowledge of both of them. So let's start with instance creation.

# Instance creation

---

In java, following are the two ways to create an instance:

- Using constructor
- Using factory method

I will not go in detail the scenarios when to use which way as we all are from Java background and had done or read the reasons number of times. We will directly start with how to use them in Spring framework one by one.

## Using Constructor

Let's take an example of a Car to make it crystal clear how the container will create the object of Car with the help of following steps:

1. Create Java application Ch02\_Instance\_Creation and add jars which we added in previous project.
2. Create a class Car in com.ch02.beans package with chesis number, it's color, fuel type, price, average as data members. The code is as shown below:

```
class Car{
    private String chesis_number, color, fuel_type;
    private long price;
    private double average;

    public Car() {
        // TODO Auto-generated constructor stub
        chesis_number = "eng00";
        color = "white";
        fuel_type = "diesel";
        price = 5700001;
        average = 12d;
    }
}
```

3. Add show() in Car as shown below:

```
public void show()
{
    System.out.println("showing car "+chesis_number+" having
        color:"+color+" and price:"+price);
}
```

4. When the developer tries to create the object the code will be as follows:

```
Car car_obj=new Car();
```

Now we need to configure BeanDefination in XML file which represents a bean instance so that the bean will be managed by the Spring container.

1. Create instance.xml in Classpath to configure our Car BeanDefination we need to configure it as follows:

```
<bean id="car_obj" class="com.ch02.beans.Car"/>
</beans>
```

2. Create TestCar with main function in default package to get the bean to use business logic.

\* Get Spring container instance. We will Use ClassPathXmlApplicationContext as discussed in container initialization.  
\* Get the bean instance from the container.

The code will be as follows:

```
public class TestCar {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new
            ClassPathXmlApplicationContext("instance.xml");
        Car car=(Car)context.getBean("car_obj");
        car.show();
    }
}
```

The output is as shown in snapshot below:

```
INFO: Loading XML bean definitions from class path resource [instance.xml]
showing car null having color:-nulland price:-0
```

Figure 02

It's pretty clear from the output that the container have used default constructor to define the values. Let's prove it by adding default constructor in Car following code as follows:

```
public Car() {
    // TODO Auto-generated constructor stub
    chesis_number = "eng00";
    color = "white";
    fuel_type = "diesel";
    price = 570000;
    average = 12d;
}
```

The updated output will be as shown in the below snapshot:

```
INFO: Loading XML bean definitions from class path resource [instance.xml]
showing car eng00 having color:-white and price:-570000
```

## Using factory method

The bean is configured in the Spring container whose object is created through either instance or static factory method.

### Using instance factory method

The instance creation will be done through a non static method of a bean. To use the method for instance creation an attribute "factory-method" has to be configured. Sometimes some other class may also be used to create the instance. The "factory-bean" attribute will be configured along with "factory-method" to be used by the container for instance creation.

Let's follow the steps to use factory-method for instance creation. We will use the same Ch02\_Instance\_Creation.

1. Create class CarFactory in com.ch02.factory as shown in the code below:

```
public class CarFactory {  
    private static Car car=new Car();  
  
    public Car buildCar()  
    {  
        System.out.println("building the car ");  
        return car;  
    }  
}
```

The `buildCar()` method will build an instance of Car and return it. Now the task of making the Container aware of using the above code will be done by the bean definition.

2. In instance.xml file add two beans, one bean for CarFactory and second for Car as shown below:

```
<bean id="car_factory" class="com.ch02.factory.CarFactory"  
<bean id="car_obj_new" factory-bean="car_factory"  
      factory-method="buildCar" />
```

The attribute `factory-method` specifies `buildCar` as the method to be used from `car_factory` specified by `factory-bean` attribute to used for instance creation. No need to specify `class` attribute here.

3. Create TestCarFactory with main function with the following code:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context = new  
        ClassPathXmlApplicationContext("instance.xml");  
    Car car = (Car) context.getBean("car_obj_new");  
    car.show();  
}
```

1. On the execution the following snapshot will be shown,

```
INFO: Loading XML bean definitions from class path resource [instance.xml]
building the car
showing car eng00 having color:-white and price:-570000
```

## Using static factory method

We can define static method in the class which returns the object. The attribute 'factory-method' is used to specify the method name who does instance creation. Let's use Ch02\_Instance\_Creation project to use factory-method attribute with the help of following steps.

1. Create class CarService in com.ch02.service package as shown below:

```
public class CarService {
    private static CarService carService=new CarService();
    private CarService(){}
    public static CarService createService()
    {
        return carService;
    }
    public void serve()
    {
        System.out.println("""car service""");
    }
}
```

2. Add the configuration in XML as follows:

```
<bean id="carService" factory-method="createService"
      class="com.ch02.service.CarService"/>
```

The '**factory-method**' specifies the method who returns the instance.

3. Write the test code in TestCarService as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context = new
        ClassPathXmlApplicationContext("""instance.xml""");
    CarService carService = (CarService)
        context.getBean("""carService""");
    carService.serve();
```

```
}
```

The execution of the code give the following output as snapshot shown below:

```
INFO: Loading XML bean definitions from class path resource [instance.xml]
building the car
,car service
```

Once the instance is created, now it's time to initialize the state. The Java developers initialization state as follows:

```
car.setChassis_Number("as123er");
car.setColor("baker's chocolate");
car.setFuel_Type("Petrol");
car.setPrice(689067L);
car.setAverage(21);
```

Here, if we change the values of the data members the state also gets changed. So it's pretty clear that the car is dependent on data member values. But as here we have set them the values are part of the code, any change in them need to change the code or redeployment of the code. In Dependency injection, the design is done in such a way that the object is achieved its state externally instead of hard coding them from a piece of the code.

# Dependency Injection

---

Dependency Inversion Principle states two modules should not be tightly coupled with each other. The modules should depend on abstraction where the details of dependency are not specified. Dependency Inversion Principle(DIP) helps in ensuring loosely coupled modular programming. Dependency Injection is the implementation of the DIP. To know what is dependency injection we first have to clearly understand what is dependency?

The state of an object is given by the values of its data members. These data members as we all are aware can be of primitive or secondary type. If the data members are primitive they get their values directly and in secondary data type, the value is dependent on state of that object. That means whenever an object initialization happens the data member initialization plays a very important role. In other words we can say the data members are the dependency in object initialization. To insert or set the values of the dependency onto the object is Dependency Injection.

The dependency injection helps in achieving loosely coupled architecture. The loose coupling helps in easy testing of the modules. The code and the values which the code uses are separated and can be controlled by central configuration, which make easy code maintenance. The code remains unaffected as the values are in external configuration making it easy to migrate with minimum changes.

In Spring framework the dependency injection can be achieved by,

- Setter injection
- Constructor injection

The above two are the ways which we can use for DI but these can be achieved with number of ways which are

- XML based configuration
- XML based configuration with namespace "p"

- Annotation based configuration

So let's start exploring them one by one

## **XML based configuration**

### **Setter Injection**

The dependencies of an object are fulfilled by the setter methods in setter injection. So the very important thing is when we do setter injection is to have a bean whose data members will be set through setter methods. The steps to use setter injection are:

1. Declare a class and its properties using standard java naming convention.
2. Configure the bean in bean definition XML as follows:

```
<bean id="bean_id" class="fully qualified _class_name"></be
```

3. The bean in above configuration create the instance. It has to be updated to configure the properties.

\* Each <property> tag will configure a data member

\* Each <property> tag accepts two values

1. name : The "name" attribute specifies the name of the data member whose value the developer wants to configure.
2. value: The "value" specifies the value to be given to the data member.

The updated configuration will be as follows:

```
<bean id="bean_id" class="fully qualified _class_name">
  <property name="name_of_property" value="value_of_property">
</bean>
```

If we have more than one data member whose values to set we need to use

more than one <property> tags.

4. Get the bean from the Spring container and you are ready to use it.

Let's first of all find out how to configure a bean with setter injection with the help of following steps:

1. Create Ch02\_Dependency\_Injection as Java project.
2. Add to it all core Spring jars which we already had used in previous chapter.
3. Create a class Car in com.ch2.beans. You can refer the code from previous project(Ch02\_Instance\_Creation).

\* As we are going to inject the dependencies using setter injection, create setter methods as well.

\* Add show() method as well.

The code will be as follows:

```
public class Car {  
    private String chesis_number, color, fuel_type;  
    private long price;  
    private double average;  
  
    public void setChesis_number(String chesis_number) {  
        this.chesis_number = chesis_number;  
    }  
  
    public void setColor(String color) {  
        this.color = color;  
    }  
  
    public void setFuel_type(String fuel_type) {  
        this.fuel_type = fuel_type;  
    }  
  
    public void setPrice(long price) {  
        this.price = price;  
    }  
  
    public void setAverage(double average) {  
        this.average = average;  
    }  
}
```

```

    }

    public void show() {
        System.out.println("showing car "+chesis_number+
                           having color:-"+color+"and price:-"+price);
    }
}

```

## Note

Make sure you should follow Bean naming convention whenever you create a class to configure in Spring Container.

The state of object can be obtained using getter methods. So normally the developers add both getters and setter. Adding getters always depends on business logic of the application:

1. Now we need to configure bean definition in beans.xml in classpath which represents a bean definition so that the bean will be managed by the Spring container. The configuration will be as follows:

```
<bean id=""car_obj"" class=""com.ch02.beans.Car"" />
```

2. In the previous step, only instance has been created we now want to set the properties of it also, which is injecting dependencies using setter. The code will be as follows:

```
<bean id=""car_obj"" class=""com.ch02.beans.Car"">
    <property name=""chesis_number"" value=""eng2012"" />
    <property name=""color"" value=""baker's chocolate"" />
    <property name=""fule_type"" value=""petrol"" />
    <property name=""average"" value=""12.50"" />
    <property name=""price"" value=""643800"" />
</bean>
```

If we don't want to set the values of any dependency, the simple thing is not to add it in the configuration.

3. Now we are ready to use the bean. We have to ask the container to give the object of the bean. Create class TestCar in default package with main function. We don't have to change anything in the main code which we

already done in TestCar in Ch02\_Instance\_Creation due to externalization of dependencies the code remains intact. The code will look as follows:

```
public class TestCar {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        ApplicationContext context=new  
            ClassPathXmlApplicationContext("beans.xml");  
        Car car=(Car)context.getBean("car_obj");  
        car.show();  
    }  
}
```

4. On execution of the code we will get the following output:

```
INFO: Loading XML bean definitions from class path resource [beans.xml]  
showing car eng2012 having color:-baker's chocolate and price:-643800
```

The values shown in the figure are the same which we set from the configuration.

## Constructor Injection

In constructor injection the dependencies will be fulfilled by parameters of the constructor or simple parameterized . Let's develop an application to use constructor based dependency injection.

Way 1: Without ambiguities

We will use the same project Ch02\_Dependency\_Injection with the help of following steps:

1. Add a parameterized constructor to Car which has the following code:

```
public Car(String chesis_number, String color, double averag  
         long price, String fuel_type) {  
    // TODO Auto-generated constructor stub  
    this.chesis_number = chesis_number;  
    this.average = average;
```

```

        this.price = price;
        this.color=color;
        this.fuel_type=fuel_type;
    }

```

## Note

If you are adding parameterized constructor and you want to use both setter, as well as constructor injection, you must add default constructor

2. Add one more bean in beans.xml file but this time instead of using <property> tag we will use <constructor-arg> to use constructor DI. The code will look like:

```

<bean id="car_const" class="com.ch02.beans.Car">
    <constructor-arg value=""eng023""></constructor-arg>
    <constructor-arg value=""green""></constructor-arg>
    <constructor-arg value=""12""></constructor-arg>
    <constructor-arg value=""678900""></constructor-arg>
    <constructor-arg value=""petrol""></constructor-arg>
</bean>

```

3. Create a class TestCarConstructorDI in default package which will take Car object from container. The code will be as follows:

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("instance.xml");
    Car car=(Car)context.getBean("car_const");
    car.show();
}

```

Here the container is happy to have each parameters of constructor with distinguished data type. But same is not the case every time. We may come across the codes where the constructor will have more than one parameter which are ambiguous in their data types. Sometimes we do have more than one constructor in class and due to auto up casting the unexpected constructor may get invoked by container. It may also happen that developer just missed the order of arguments.

## Way2: with ambiguities

1. Let's add one more bean definition in the beans.xml as shown below:

```
<bean id=""car_const1"" class=""com.ch02.beans.Car"">
    <constructor-arg value=""eng023""></constructor-arg>
    <constructor-arg value=""green""></constructor-arg>
    <constructor-arg value=""petrol""></constructor-arg>
    <constructor-arg value=""12""></constructor-arg>
    <constructor-arg value=""678900""></constructor-arg>
</bean>
```

The number of arguments are matching to the constructor definition but, the third argument instead of passing average we passed fuel\_type value. Don't worry just continue your journey and have faith!

2. Create TestConstructor\_Ambiguity to find what happens on mismatching arguments. The code is as shown below:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("beans.xml");
    Car car=(Car)context.getBean("car_const1");
    car.show();
}
```

3. The execution of main function gives exception as shown in the snapshot below:

```
Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException:
Error creating bean with name 'car_const1' defined in class path resource [beans.xml]:
Unsatisfied dependency expressed through constructor parameter 2:
Could not convert argument value of type [java.lang.String] to required type [double]:
Failed to convert value of type [java.lang.String] to required type [double];
nested exception is java.lang.NumberFormatException: For input string: "petrol"
```

The underline line expressed the ambiguity of the values of arguments. We can have two Solutions as:

- \* You can change the order of configuration so as to match to the constructor argument.

\* Spring provides handy way by providing "index" attribute to resolve the order of constructor arguments.

One more way Spring gives to configure "type" attribute.

4. Let's try out configuring "index" by updating bean definition from Step 1 as shown below:

```
<bean id=""car_const1"" class=""com.ch02.beans.Car"">
    <constructor-arg value=""eng024"" index=""0"">
        </constructor-arg>
    <constructor-arg value=""yellow"" index=""1"">
        </constructor-arg>
    <constructor-arg value=""petrol"" index=""4"">
        </constructor-arg>
    <constructor-arg value=""15"" index=""2"">
        </constructor-arg>
    <constructor-arg value=""688900"" index=""3"">
        </constructor-arg>
</bean>
```

You can find one more extra attribute we configure this time as "index" which will tell the container which value is for which argument. The "index" always starts with "0". We haven't changed the order of the properties in configuration.

5. Run the same TestConstructor\_Ambiguity. And you will get your instance without any problem.

## Note

Specifying the index is the safest way to overcome ambiguity but even we can specify "type" instead of index to overcome the situation. But if the constructor has more than one argument with the same data type "type" will not help us out. To use type in our previous code we need to change the bean definition in XML file as,

Here we have explored the way by which bean properties to set. But if you keenly observe all the properties which we set here are primitive but even we

can have secondary data as data member. Let's find out how to set the properties of secondary data type with the help of following demo.

Let's develop an example of Customer who has Address as one of the data member. For better understanding follow the steps:

1. Create Java Application,"Ch02\_Reference\_DI" and add to it the jars as we did in previous example.
2. Create Address class in com.ch02.beans package having city\_name, build\_no, pin\_code as data members. Add to it setter methods as well. The code will be as follows:

```
public class Address {  
    private String city_name;  
    private int build_no;  
    private long pin_code;  
  
    public void setCity_name(String city_name) {  
        this.city_name = city_name;  
    }  
    public void setBuild_no(int build_no) {  
        this.build_no = build_no;  
    }  
    public void setPin_code(long pin_code) {  
        this.pin_code = pin_code;  
    }  
    @Override  
    public String toString() {  
        // TODO Auto-generated method stub  
        return this.city_name+"\t"+this.pin_code;  
    }  
}
```

3. Create Customer in com.ch02.beans package with cust\_name, cust\_id, cust\_address. Add getter and setter methods. The code will as follows:

```
public class Customer {  
    private String cust_name;  
    private int cust_id;  
    private Address cust_address;  
    //getter and setter methods  
}
```

You can easily find that cus\_address is of secondary data.

4. For configuration create customer.xml in Classpath.
5. We have two beans to be configured in the XML. First bean for Address and second for Customer. Let's first off all configure bean for Address as follows:

```
<bean id=""cust_address"" class=""com.ch02.beans.Address"""
    <property name=""build_no"" value=""2"" />
    <property name=""city_name"" value=""Pune"" />
    <property name=""pin_code"" value=""123"" />
</bean>
```

Note the id for Address which we use here is "cust\_address". But if you want you can use your own.

6. Now, add the configuration for Customer as shown in the code below:

```
<bean id=""customer"" class=""com.ch02.beans.Customer"""
    <property name=""cust_id"" value=""20"" />
    <property name=""cust_name"" value=""bob"" />
    <property name=""cust_address"" ref=""cust_address"" />
</bean>
```

The cust\_id and cust\_name will have the value attribute directly. But, the cust\_address is not primitive so instead of using "value" as an attribute we need to use "ref" here.

**\* ref:** The "ref" attribute is used to refer to the object we need to inject. The value of "ref" is value of "id" from the container. Here we use ref value as "cust\_address" as we already had declared one bean with the similar id for the Address data type.

7. Now it's time to test how the code is working. Add TestCustomer in default package with main method to get object of Customer from the container with the help of following code:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("customer.xml");
    Customer customer=(Customer)context.getBean("customer");
    System.out.println(customer.getCust_name()+"\t"+
```

```
        customer.getCust_id());
    }
```

## Note

Instead of directly using data members developers normally invoke business logic method of the bean.

8. On execution we will get customer id and customer name on the console.

## Note

We even can use here constructor injection by adding parameterized constructor and using <constructor-arg> instead of <property>tag. As seen above the "value" attribute has to be replaced by "ref". You can find the code from Customer\_Constructor\_DI.java

## Namespace "p" for property

In one of the previous example we had used <property>tag for injecting values of the properties in the instance. The framework have provided more sophisticated way and alternative for <property> with the help of "p" namespace. To use "p" namespace the developers have to add schema URI <http://www.springframework.org/schema/p> in the configuration file as shown below:

```
xmlns:p=""http://www.springframework.org/schma/p""
```

The syntax to use "p" for setting the primitive values of the properties is:

```
p: name_of_property =value_of_the_property_to_set.
```

In case of setting more than one properties separate them with spaces.

The syntax changes for reference data types as follows:

```
p: name_of_property-ref =id_of_beans_which_to_inject
```

Let's use the new configuration in the XML. We will use the same project structure as that of Ch02\_Dependency\_Injection and just modify the beans.xml. Let's create a new application with the help of following steps:

1. Create a new Java Project named Ch02\_DI\_using\_namespce and add the jars to it.
2. Create or copy Car class in com.ch02.beans package. You can refer the code from Ch02\_Dependency\_Injection.
3. Create beans.xml file and update it for the declaration of namespace "p" as shown above. The configuration file will be as shown below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
                           http://www.springframework.org/schema/beans/spring-beans.
                           </beans>
```

4. Add the bean definition using namespace 'p' as follows:

```
<bean id="car_obj" class="com.ch02.beans.Car"
      p:chassis_number="eng2013" p:color="baker's chocolate"
      p:fuel_type="petrol" p:average="12.50" p:price="750070">
    </bean>
```

5. Create class TestCar to get instance of Car from container and execute the code. You can refer to the same TestCar.Java from Ch02\_Dependency\_Injection project.

Once we know how to set primitives let's do the coding for reference configuration as well. We will use the same DI\_using-namespce project to develop the further code:

1. Copy or Create class Address . (refer to the code from Ch02\_Reference\_DI) in com.ch02.beans package.
2. Copy or Create class Customer . (refer to the code from Ch02\_Reference\_DI) in com.ch02.beans package.
3. Add a bean for Address using setter DI in bean.xml.
4. Add a bean for Customer using namespace "p" and the configuration

will look like:

```
<bean id=""customer"" class=""com.ch02.beans.Customer"""
      p:cust_id=""2013"" p:cust_name=""It's Bob"""
      p:cust_address-ref=""cust_address"">
</bean>
```

You can observe here the customer address is not primitive so instead of using value we use reference as p:cust\_address ="cust\_address" where "cust\_address" is the id representing Address bean.

5. Create TestCustomer with main function containing the following code:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext ("beans.xml");
    Customer customer=(Customer)context.getBean ("customer");
    System.out.println(customer.getCust_name () +"\t"+
        customer.getCust_id ());
}
```

6. The execution of code will show the following output:

```
INFO: Loading XML bean definitions from class path resource [beans.xml]
It's Bob      2013
```

## Configuring the inner beans

The bean definition file has more than one beans which are managed by the container. As we are well aware these beans are reusable by each other. At one point this reusability is amazing thing but along with it a problem comes. Let's take an example of Customer's Address. The configuration can be done for Address as well as Customer. Can we use address bean for the another Customer? Yes, we can. But.....what if the Customer don't want to share his address and to keep it personal? With the current configuration it's not possible. But we can have an alternative configuration which gives the facility to keep the address personal. The configuration of customer bean from Ch02\_Reference\_DI can be modified for using inner as shown below:

```

<bean id=""customer_obj"" class=""com.ch02.beans.Customer"">
    <property name=""cust_id"" value=""20"" />
    <property name=""cust_name"" value=""bob"" />
    <property name=""cust_address"">
        <bean class=""com.ch02.beans.Address"">
            <property name=""build_no"" value=""2"" />
            <property name=""city_name"" value=""Pune"" />
            <property name=""pin_code"" value=""123"" />
        </bean>
    </property>
</bean>

```

The address bean is inner been whose instance will be created and wired with the address property of the Cutomer instance. This is similar to Java inner classes.

As setter injection supports inner beans, they are supported by the constructor injection as well. The configuration will be:

```

<bean id=""customer_obj_const"" class=""com.ch02.beans.Customer"">
    <constructor-arg value=""20"" />
    <constructor-arg value=""bob"" />
    <constructor-arg>
        <bean id=""cust_address"" class=""com.ch02.beans.Address"">
            <property name=""build_no"" value=""2"" />
            <property name=""city_name"" value=""Pune"" />
            <property name=""pin_code"" value=""123"" />
        </bean>
    </constructor-arg>
</bean>

```

You can find the complete code in Ch02\_InnerBeans project.

## Inheritance mapping

Inheritance is the major pillar of Java. Spring supports for bean definitions to configure in XML. The inheritance support is provided by "parent" attribute which says the missing properties can be used from the parent bean. Using "parent" gives similarity with parent class in inheritance when we develop Java code. Even it's possible to override the properties from the parent bean. Let's find out how to use practically with the help of following steps:

1. Create Ch02\_Inheritance as Java project.
2. Add the jars.
3. Create class Student in com.ch02.beans package with following code:

```
public class Student {
    private int rollNo;
    private String name;
    // getters and setters
}
```

4. Create class EnggStudent inherited from Student as:

```
public class EnggStudent extends Student {
    private String branch_code;
    private String college_code;
    // getters and setters
}
```

5. Create the student.xml in classpath to configure bean for student as:

```
<bean id=""student"" class=""com.ch02.beans.Student"">
    <property name=""rollNo"" value=""34"" />
    <property name=""name"" value=""Sasha"" />
</bean>
```

6. Now it's time to configure bean for EnggStudent. First off all an ordinary configuration which we don't want to use:

```
<bean id=""engg_old"" class=""com.ch02.beans.EnggStudent"">
    <property name=""rollNo"" value=""34"" />
    <property name=""name"" value=""Sasha"" />
    <property name=""branch_code"" value=""Comp230"" />
    <property name=""college_code"" value=""Clg_Eng_045"" />
</bean>
```

It's very clear that we repeated the configuration for rollNo and name. We don't have to repeat the configuration by configuring "parent" attribute as shown below:

```
<bean id="engg_new" class="com.ch02.beans.EnggStudent"
    parent="student">
    <property name="branch_code" value="Comp230"/>
    <property name=""college_code"" value=""Clg_Eng_045"" />
</bean>
```

Though here we skip configuring name and rollNo and reusing it from "student" bean, it's possible to override any one of them as shown below:

```
<bean id="engg_new1" class="com.ch02.beans.EnggStudent"
  parent="student">
  <property name=""rollNo"" value=""40"" />
  <property name=""branch_code"" value=""Comp230"" />
  <property name=""college_code"" value=""Clg_Eng_045"" />
</bean>
```

The choice is yours, which one to use!!

1. Write TestStudent with main function as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("student.xml");
    //old configuration
    EnggStudent student_old=
        (EnggStudent)context.getBean("engg_old");
    System.out.println("with old configuration");
    System.out.println(student_old.getName()+
        "\t"+student_old.getRollNo()+"\t"+
        student_old.getCollege_code());
    //new configuration
    EnggStudent student_new=
        (EnggStudent)context.getBean("engg_new");
    System.out.println("with new configuration");
    System.out.println(student_new.getName()+
        "\t"+student_new.getRollNo()+"\t"+
        student_new.getCollege_code());
    System.out.println("with new configuration with
        overriding roll number");
    //new configuration with overriding the roll number
    EnggStudent student_new1=
        (EnggStudent)context.getBean("engg_new1");
    System.out.println(student_new1.getName()+
        "\t"+student_new1.getRollNo()+"\t"+
        student_new1.getCollege_code());
}
```

2. The below snapshot show the console output as follows:

```

INFO: Loading XML bean definitions from class path resource [student.xml]
with old configuration
Sasha 34 Clg_Eng_045
with new configuration
Sasha 34 Clg_Eng_045
with new configuration with overriding roll number
Sasha 40 Clg_Eng_045

```



↑ the overridden rollNo

- The developers can able to obtain Student instance as well as EnggStudent. In some cases we don't want anybody to use Student instance or nobody should able to get Student instance. In such situations configure attribute "abstract" on the bean whose instance developers don't want to make publically available. By default the value of abstract=false which states anybody can obtain instance of bean. We will configure abstract=""true"" making it unavailable. The updated configuration of the Student will be as follows:

```

<bean id=""student"" class=""com.ch02.beans.Student"""
      abstract=""true"">
    <property name=""rollNo"" value=""34"" />
    <property name=""name"" value=""Sasha"" />
</bean>

```

- Now, whenever someone ask for student bean BeanIsAbstractException will be thrown. You can try out the code by adding following lines in TestStudent:

```

System.out.println("obtaining Student instance");
Student student=(Student)context.getBean("student");

```

- On execution we will get the following stack trace which specifies bean creation exception occur while obtaining Student bean:

```
Exception in thread "main" org.springframework.beans.factory.BeanIsAbstractException: 
at org.springframework.beans.factory.support.AbstractBeanFactory.
       checkMergedBeanDefinition(AbstractBeanFactory.java:1291)
at org.springframework.beans.factory.support.AbstractBeanFactory.
       doGetBean(AbstractBeanFactory.java:283)
at org.springframework.beans.factory.support.AbstractBeanFactory.
       getBean(AbstractBeanFactory.java:195)
at org.springframework.context.support.AbstractApplicationContext.
       getBean(AbstractApplicationContext.java:1076)
at TestStudent.main(TestStudent.java:29)
```

## Configuring the null values

In Java unless if any the values of the data member is not set each will get their default ones. The reference properties will be defined as null and primitive integers to "0" respectively. These nulls later on will be overridden by the set values by either constructor injection or setter injection. It may also be possible that developers want to keep it null until some business logic won't give the calculated value or get it from some external resource. Whatever the reason, we want to configure the value as null simply use "<null/>". The Customer's address will be set to null as shown below:

```
<bean id=""customer_obj"" class=""com.ch02.beans.Customer"">
  <property name=""cust_id"" value=""20"" />
  <property name=""cust_name"" value=""bob"" />
  <property name=""cust_address""><null/></property>
</bean>
```

You can find the configuration in customer.xml of Ch02\_InnerBeans.

Up till now, we had seen the mapping of primitives, references, null or inner beans. We are more than happy, but wait a very important fundamental concepts of Java is Collection framework. Yes, we have to discuss mapping of collection as well.

## Configuring Collection

The Collection framework in Java facilitates handling objects to perform various operations as addition, removal, searching, sorting of objects in simplified way. The interfaces Set, List, Map, Queue are the interfaces has

many implementation like HashSet, ArrayList, TreeMap, PriorityQueue and many more which gives means for handling data. We will not go in detail which one to choose for the operations but, we will be discussing different configurations supported by Spring in injecting the Collection.

## Mapping List

List is ordered collection which offers handling of data in order of data insertion . It maintains the insertion, removal, fetching of data by indices where duplicate elements are allowed. ArrayList, LinkedList are some of its implementations. The framework supports List configuration with help of <list> tag. Let's develop an application to configure List by following steps:

1. Create Ch02\_DI\_Collection as Java project and add Spring jars to it.
  2. Create POJO class Book in com.ch02.beans package.
- \* Add isbn, book\_name, price and publication as data members.
  - \* Add default and parameterised constructors.
  - \* Write .getter and setter methods.
  - \* As we are handling collection add equals() and hashCode()
  - \* To display object add toString()

The Book will be as shown in the code below,

```
public class Book {  
    private String isbn;  
    private String book_name;  
    private int price;  
    private String publication;  
  
    public Book()  
    {  
        isbn = "310IND";  
        book_name = "unknown";  
        price = 300;  
        publication = "publication1";  
    }  
}
```

```

public Book(String isbn, String book_name, int price, String publication)
{
    this.isbn=isbn;
    this.book_name=book_name;
    this.price=price;
    this.publication=publication;
}

@Override
public String toString() {
    // TODO Auto-generated method stub
    return book_name+"\t"+isbn+"\t"+price+"\t"+public
}

@Override
public boolean equals(Object object) {
    // TODO Auto-generated method stub
    Book book=(Book)object;
    return this.isbn.equals(book.getIsbn());
}

public int hashCode() {
    return book_name.length()/2;
}

// getters and setters
}

```

3. Create Library\_List in com.ch02.beans who has List of Books. Write displayBooks() to display list of the books. The code will be as:

```

public class Library_List {
    private List<Book> books;
    public void displayBooks()
    {
        for(Book b:books)
        {
            System.out.println(b);
        }
    }
    // getter and setter for books
}

```

4. Create books.xml in ClassPath. Add to it four Book beans. I am trying

following three configurations:

- \* book bean with setter DI.
- \* book bean with constructor DI.
- \* book bean using "p" namespace.

No need to try out all combination, you can follow any one of them. We already have used all of them in the previous Demos. The configuration will be as shown below:

```
<bean id=""book1"" class=""com.ch02.beans.Book"">
    <property name=""isbn"" value=""20Novel"" />
    <property name=""book_name"" value=""princess Sindrella"" />
    <property name=""price"" value=""300"" />
    <property name=""publication"" value=""Packt-Pub""></prope
</bean>

<bean id="book2" class="com.ch02.beans.Book">
    <constructor-arg value="Java490" />
    <constructor-arg value="Core Java" />
    <constructor-arg value="300" />
    <constructor-arg value="Packt-Pub" />
</bean>

<bean id="book3" class="com.ch02.beans.Book"
    p:isbn="200Autobiography"
    p:book_name="Playing it in my way" p:price="300"
    p:publication="Packt-Pub">
</bean>
```

Intentionally the fourth book is second copy of one of the first three books, we in couple of steps discover the reason. Just wait and watch!!

## 5. Add a Library bean with <list> configuration as:

```
<bean id=""library_list"" class=""com.ch02.beans.Library_Li
<property name=""books"">
    <list>
        <ref bean=""book1""></ref>
        <ref bean=""book2""></ref>
        <ref bean=""book3""></ref>
        <ref bean=""book3""></ref>
```

```

        </list>
    </property>
</bean>
```

The <list> contains list of <ref> of the beans to be injected for list of books where 'book1','book2','book3','book4' are the id's of beans which we created in Step 4.

6. Create TestLibrary\_List with main function to get instance of Library and list of Books it has. The code is as follows:

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("books.xml");
    Library_List list=
        (Library_List)context.getBean("library_list");
    list.displayBooks();
}
```

7. Execute it to get the output displaying list of books. Find last two entries which indicate List allows duplicate elements:

```

INFO: Loading XML bean definitions from class path resource [books.xml]
princess Sindrella      20Novel 300      Packt-Pub
Core Java          Java490 300      Packt-Pub
Playing it in my way   200Autobiography      300      Hodder and Stoughton
Playing it in my way   200Autobiography      300      Hodder and Stoughton
```

## Mapping Set

The Set interface is an unordered collection, which doesn't allow duplicate entries in the collection. HashSet, TreeSet are the implementations of Set. Spring provides <set> tag to configure Set. Let's use Ch02\_DI\_Collection project to add Set of Books by following steps:

1. Add Library\_Set class in com.ch02.beans package and declare Set of Books as Data member. Add getters and setters for it. The code is as shown below:

```
public class Library_Set {
```

```

        HashSet<Book> books;

        public void displayBooks()
        {
            for(Book b:books)
            {
                System.out.println(b);
            }
        }
        //getter and setter for books
    }

```

2. Add a bean for Library\_Set in beans.xml with <set> configuration as shown below:

```

<bean id=""library_set"" class=""com.ch02.beans.Library_Set">
    <property name=""books"">
        <set>
            <ref bean=""book1""></ref>
            <ref bean=""book2""></ref>
            <ref bean=""book3""></ref>
            <ref bean=""book3""></ref>
        </set>
    </property>
</bean>

```

3. Create TestLibrary\_Set with main function as shown below:

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("books.xml");
    Library_Set set=
        (Library_Set)context.getBean("library_set");
    set.displayBooks();
}

```

4. The output of execution is as shown below:

```

INFO: Loading XML bean definitions from class path resource [books.xml]
princess Sindrella      20Novel 300      Packt-Pub
Core Java      Java490 300      Packt-Pub
Playing it in my way   200Autobiography      300      Hodder and Stoughton

```

We injected four objects of books but got only three in output so it's crystal clear that Set doesn't allow duplicates.

## Mapping Map

Map handles collection of objects having key and value pair. Map can have duplicate values but duplicate keys are not allowed. It has implementations as TreeMap, HashMap and LinkedHashMap.

Let's explore configuring Map with the help of following steps:

1. We will use the same Ch02\_DI\_Collection project. Create class Library\_Map in com.ch02.beans package.
2. Add Map<String,Book> books as data member in it along with getters and setters. Don't forget to add displayBooks() in it. The code will be as shown below:

```
public class Library_Map {  
    private Map<String, Book> books;  
    //getters and setters  
  
    public void displayBooks()  
    {  
        Set<Entry<String, Book>> entries=books.entrySet();  
        for(Entry<String, Book> entry:entries)  
        {  
            System.out.println(entry.getValue());  
        }  
    }  
}
```

3. Configure the Map in beans.xml as shown below:

```
<bean id=""library_map"" class=""com.ch02.beans.Library_Map"">  
    <property name=""books"">  
        <map>  
            <entry key=""20Novel"" value-ref=""book1"" />  
            <entry key=""200Autobiography"" value-ref=""book3"" />  
            <entry key=""Java490"" value-ref=""book2"" />  
        </map>  
    </property>  
</bean>
```

Unlike List and Set the Map will take extra attribute of 'key' to specify the key. We used here name of the book as key but if you want you can declare something else as well. Just don't forget that key is always unique in Map:

1. Write TestLibrary\_Map with main function as follows:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
        ClassPathXmlApplicationContext("books.xml");  
    Library_Map map=  
        (Library_Map)context.getBean("library_map");  
    map.displayBooks();  
}
```

2. Execute the code to get book details displayed on console.

Here we configure single object as per entry. But in practice an entry may contain List or Set of books. In such cases instead of having <entry> with "value-ref" the configuration will contain <entry> with <list> as shown below:

```
<bean id=""library_map1"" class=""com.ch02.beans.Library_Map1"">  
    <property name=""books"">  
        <map>  
            <entry key=""20Novel"">  
                <list>  
                    <ref bean=""book1""></ref>  
                    <ref bean=""book1""></ref>  
                </list>  
            </entry>  
            <entry key=""200Autobiography"">  
                <list>  
                    <ref bean=""book3""></ref>  
                    <ref bean=""book3""></ref>  
                </list>  
            </entry>  
        </map>  
    </property>  
</bean>
```

In the above configuration each "entry" has name of books containing "<list>" of books which is but obvious if we are talking about Library of books. The complete code of Library\_Map1.java and TestLibrary\_Map1.java can be

refer from Ch02\_DI\_Collection.

## Mapping Properties

Properties also hold the collection of elements in key-value pair, but unlike Map here the key and value both are of String types only. The Properties can be saved or read from stream.

Let's consider a Country who has multiple States as Properties. Follow the steps to find out how to configure Properties in beans.xml:

1. Create a class Country in com.ch02.beans package.
2. Declare name, continent and state\_capital as data members. Add getters and setters as well. To display state capitals add printCapital(). The code is as shown below:

```
public class Country {  
    private String name;  
    private String continent;  
    private Properties state_capitals;  
  
    // getters and setter  
  
    public void printCapitals()  
    {  
        for(String state:state_capitals.stringPropertyNames())  
        {  
            System.out.println(state+":\t"+  
                state_capitals.getProperty(state));  
        }  
    }  
}
```

3. In beans.xml configure the definition of Country as shown below:

```
<bean id=""country"" class=""com.ch02.beans.Country"">  
    <property name=""name"" value=""India""></property>  
    <property name=""continent"" value=""Asia""></property>  
    <property name=""state_capitals"">  
        <props>  
            <prop key=""Maharashtra"">Mumbai</prop>  
            <prop key=""Goa"">Panji</prop>  
            <prop key=""Punjab"">Chandigarh</prop>  
        </props>  
    </property>  
</bean>
```

```

        </props>
    </property>
</bean>
```

The 'state\_capitals' contains '<props>' configuration to hold name of state as 'key' and its capital as 'value'.

1. Write TestProperties with main function having the below code,

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
            ClassPathXmlApplicationContext("books.xml");
    Country country=(Country)context.getBean("country");
    country.printCapitals();;
}
```

2. The output will be as shown in the snapshot:

```

INFO: Loading XML bean definitions from class path resource [books.xml]
Maharashtra: Mumbai
Goa: Panji
Punjab: Chandigarh
```

The 'util' namespace provides a means to the developers for configuring collections in XML file elegantly. Using 'util' namespace one can configure List, Map, Set, Properties. To use "util" namespace the schema has to be updated for [www.springframework.org/schema/util](http://www.springframework.org/schema/util) URI as shown in snapshot below:

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
```

The underlined lines have to be added for using 'util' namespace in the configuration.

The configuration of List using "util" namespace will be as shown below:

```
<bean id=""library_list1"" class=""com.ch02.beans.Library_L
<property name=""books"">
    <util:list>
        <ref bean=""book1""></ref>
        <ref bean=""book2""></ref>
        <ref bean=""book3""></ref>
    </util:list>
</property>
</bean>
```

You can find the updated configuration in books.xml.

We know how to get the bean and how to fulfill different types of dependencies it has. The bean configuration defines the way by which instances created and its state will be defined by injecting the dependencies. At any point of time for business logic requirement the state of bean can be changed. But we yet don't know how many instances Spring container creates or what if developers want single instance to serve every request? Or what is every operation needs different instances. Actually we are talking about "scope"

## Bean Scope

The scope of bean defines how many instances will be created by the Spring container and make it available for application to use. Use "scope" attribute of `<bean>` to provide information about number of instances. We cannot move before discovering the default process of creating and providing instances. It will make the term "scope" clear as well it will define why understanding 'scope' is so important.

Let's use Ch02\_Dependency\_Injection project to find how many instances the container creates by default. You can use the same project or can create a new copy of it as we are doing in following steps.

1. Create Ch02\_Bean\_Scope as Java project.
2. Add Spring jars to it.
3. Create or copy Car in com.ch02.beans package.

4. Create beans.xml in class path and configure "car" bean as shown below,

```
<bean id=""car"" class=""com.ch02.beans.Car"">
    <property name=""chesis_number"" value=""eng2012"" />
    <property name=""color"" value=""baker's chocolate"" />
    <property name=""fuel_type"" value=""petrol"" />
    <property name=""average"" value=""12.50"" />
    <property name=""price"" value=""643800"" />
</bean>
```

The scope is no where related to how the bean has been configured.

5. Create TestCar to request for 'car' bean twice as shown below. Don't be surprise. We want to find out how many instances created. So let's start:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("beans.xml");
    // first request to get the car instance
    Car car_one=(Car)context.getBean("car");
    car_one.show();

    //second request to get the car instance
    Car car_two=(Car)context.getBean("car");
    car_two.show();
}
```

The execution of the code will give the following output,

```
showing car eng2012 having color:-baker's chocolate and price:-643800
showing car eng2012 having color:-baker's chocolate and price:-643800
```

Yes both the object has the same value. And why not? We configure them in XML. This won't lead us to any conclusion. Let's move ahead for second step.

\* Use the same TestCar code but, this time change the state of any one car object. We will change for "car\_one" and observe what happens to car\_two? Will car\_two contains changed values or configured values ? The changed code will be as below:

```

public class TestCarNew {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new
            ClassPathXmlApplicationContext("beans.xml");
        // first request to get the car instance
        Car car_one=(Car)context.getBean("car");
        car_one.show();

        //second request to get the car instance
        Car car_two=(Car)context.getBean("car");
        car_two.show();

        System.out.println("after changing car-one is");
        // change average and color of car_one
        car_one.setAverage(20.20d);
        car_one.setColor("white");
        car_one.show();
        System.out.println("after changing car-two is");
        car_two.show();
    }
}

```

On execution you will get the following output.

```

showing car eng2012 having color:-baker's chocolate and price:-643800
showing car eng2012 having color:-baker's chocolate and price:-643800
after changing car-one is
showing car eng2012 having color:-white and price:-800000
after changing car-two is
showing car eng2012 having color:-white and price:-800000

```

We just changed state of car\_one but the output shows even state of car\_two got changed which proves no matter how many time you ask the container for the instance every time the same instance will be given back to us.

## Note

"singleton" is the default scope of bean means a single instance per Spring container.

\* Keep TestCarNew as it is and configure "scope" attribute in car bean as

shown below,

```
<bean id=""car"" class=""com.ch02.beans.Car"" scope=""prototype""
```

Execute TestCarNew and you will get the following output:

```
showing car eng2012 having color:-baker's chocolate and price:-643800
showing car eng2012 having color:-baker's chocolate and price:-643800
after changing car-one is
showing car eng2012 having color:-white and price:-800000
after changing car-two is
showing car eng2012 having color:-baker's chocolate and price:-643800
```

The output shows change state of car\_one doesn't change state of car\_two. Which means, on every request for instance of car from the container, the container creates and give a new instance.

## Note

"prototype" specifies a new instance per request for instance from the container.

The following are few more scopes given by Spring framework,

- **request** : By default in web application all HTTP request get served by the same instance which may lead to the problems in handling data or consistency of data. To make sure each request get its own new fresh instance "request" scope will be helpful.
- **session**: We are well aware of handling session in web application. "request" creates instance per request but when multiple request are bound to the same session then instance per request become clumsy and so do managing the data. The "session" scope is the rescue in such situation where the developers needs instance per session.
- **application**: Each web application has its own ServletContext. The "application" scope creates instance per ServletContext.
- **globalSession**: In protletContext the Spring configuration provides instance per global HttpSession.
- **websocket**: The "websocket" scope specifies instance creation per

WebSocket".

## Annotation Based Configuration

From Spring 2.5 onwards Spring started supporting Annotation based configuration. We already have discussed the concepts in Java and how to configure them in XML? While doing the annotation based configuration we come across two types of configuration

- Spring based annotations
- JSR based annotations.

### Spring based annotations

Number of annotations has been provided by Spring which may be categories as managing life cycle, creating bean instance, wiring annotations any many more. Let's find them one by one. But before that we need to know one very important thing. The bean definition has to be registered by configuring it in XML. Now to have annotation based configuration the bean registration has to be done using context namespace as follows:

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

The `<context:annotation-config>` asks the container to consider the annotations on the bean.

Instead of the above configuration one can even go with the following configuration,

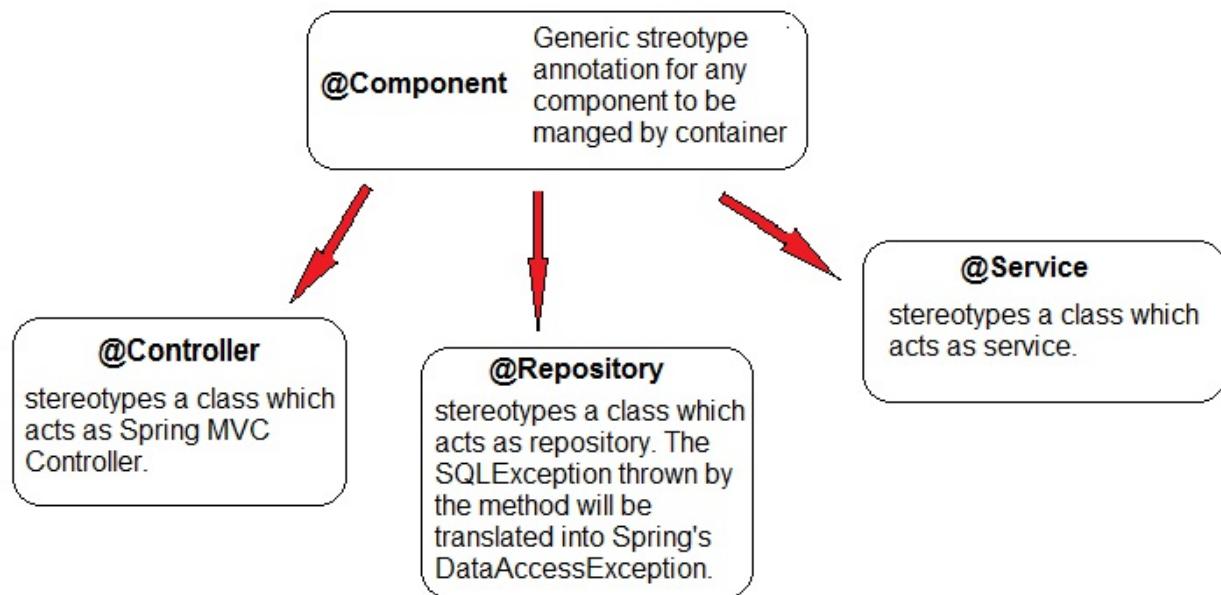
```
<bean calss=""or.springframework.bean.factory.annotation.
```

```
AutowiredAnnotationBeanPostProcessor""/>
```

Let us start with using annotations one by one for different scenarios as follows,

### Stereotype annotations

These annotations are class level annotations which get registered with Spring container. The following figure shows stereotype annotations:



Let's use `@Component` in the application. We will use Ch02\_Reference\_DI as base project and use the following steps:

1. Create Ch02\_Demo\_Annotation and all the jars which we have added earlier , along with them add spring-aop.jar to it as well.
2. Create or copy Address.java in com.ch02.beans package.
3. Create or copy Customer.java in com.ch02.stereotype.annotation package. Rename it to Customer\_Component.java
4. Add to it default constructor as the code we are refereeing is not having any default constructor. The code can be as below:

```
public Customer_Component() {  
    // TODO Auto-generated constructor stub  
    cust_id=100;
```

```

        cust_name = "cust_name";
        cust_address = new Address();
        cust_address.setBuild_no(2);
        cust_address.setCity_name("Delhi");
    }
}

```

5. Annotate the class with `@Component` as shown below:

```

@Component
public class Customer_Component { }

```

Here we had configured a component which is auto scanable. By default the id of this component will be decapitalized class name. In our case it's "customer\_Component". If we want to use customized id we can modify the configuration to use "myObject" as bean id will be,

```
@Component(value = "myObject")
```

And there is no need to configure it in XML as we did earlier. But we still need XML for some other configurations.

6. Create or copy customer.xml in classpath.
7. Add the following code to consider the annotations:

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

We already had seen how to configure "context" namespace.

8. Create TestCustomer\_Component to get the bean of Customer\_Component as show below to find out what is the out put of our configuration:

```

public class TestCustomer_component {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context = new
            ClassPathXmlApplicationContext("customer.xml");
        Customer_Component customer = (Customer_Component)
            context.getBean("customer_Component");
        System.out.println(customer.getCust_name() + "\t" +
            customer.getCust_id() +
            customer.getCust_address());
    }
}

```

```
}
```

On execution we will get the following stack trace of exception,

```
Exception in thread "main"
org.springframework.beans.factory.NoSuchBeanDefinitionException
No bean named ''customer_component'' is defined
```

9. If we did everything then why we still are getting exception. The reason made component auto scan able but forgot to tell the container where to scan for annotations? Let's configure where to scan for components as:

```
<context:component-scan
    base-package=""com.ch02.stereotype.*"">
</context:component-scan>
```

Here all the subpackages of com.ch02.stereotype will be scanned to find out bean having id as "customer\_Component".

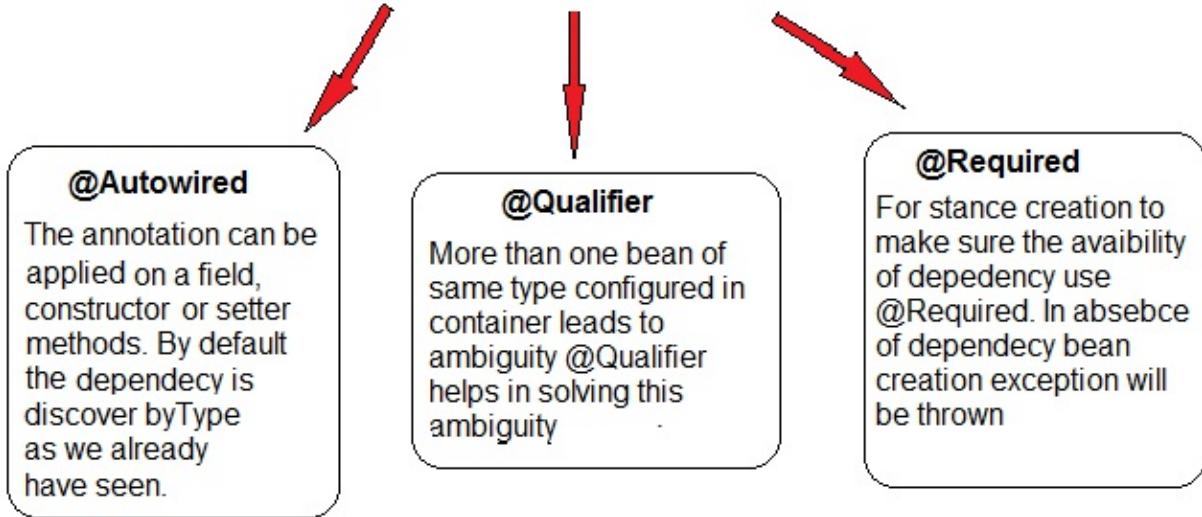
10. After adding the configuration we will get the following output displaying the data member values as:

```
INFO: Loading XML bean definitions from class path resource
[customer.xml]
cust_name 100 Delhi 2
```

In the same way we can use the other annotations but we will discuss them one by one in the upcoming chapters.

### **Wiring annotations**

We already had discussed in depth about "auto wiring" which gives us the facility to reduce the configuration and automatically discover which object to inject as the dependency in the bean. Following are the annotations which help in auto wiring and to sort out the problems which occur in auto wiring.



Let's use Cho2\_Demo\_Annotation project to demonstrate the above annotations.

### Case1. Using @Autowired

1. Create class Customer\_Autowired in com.ch02.autowiring.annotation. This is similar to Customer\_Component which we has created in stereotype annotations. Don't forget to add default customer if you don't have one.
2. Add @Autowired annotations on the field cust\_address as shown below:

```
@Autowired
private Address cust_address;
```

3. In customer\_new.xml configure a bean for Address as shown below,

```
<bean id=""cust_address"" class=""com.ch02.beans.Address"">
    <property name=""build_no"" value=""2"" />
    <property name=""city_name"" value=""Pune"" />
    <property name=""pin_code"" value=""123"" />
</bean>
```

4. Create TestCustomer\_Autowired to get the bean with "customer\_Autowired" as id:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
```

```

ApplicationContext context = new
    ClassPathXmlApplicationContext("customer_new.xml");
Customer_Autowired customer = (Customer_Autowired)
    context.getBean("customer_Autowired");
System.out.println(customer.getCustomer_name() + "\t" +
    customer.getCustomer_id() +
    customer.getCustomer_address());
}

```

5. We will get the following console output on execution:

```

INFO: Loading XML bean definitions from class path resource
[customer_new.xml]
my name 10 Mumbai 12

```

### **Case2. Using autowired=false**

Whatever the care developers take we always come across scenarios where the dependency is not available or doesn't get injected due to some reason. It's quite obvious to get exception here. In order to get rid of such forceful injection add '`autowired=false`' to `@Autowired`.

1. You can try this out by deleting `cust_address` from `customer.xml` in previous project. On execution of main we will get exception as:

```

Caused by:
org.springframework.beans.factory.NoSuchBeanDefinitionException
No qualifying bean of type [com.ch02.beans.Address] found for dependency [com.ch02.beans.Address]: expected at least 1 bean which qualifies as autowire candidate for this dependency.
Dependency annotations:
{@org.springframework.beans.factory.annotation.Autowired(required=true)}

```

2. Change `@Autowired` annotation to:

```

@.Autowired(required=false)
private Address cust_address;

```

1. Again rerun main where we get the following output:

```
my name 10null
```

The value null denotes no address has been injected but we haven't got any bean creation exception here.

## Note

Once you finished the demonstration of required=false undo all the changes we just made in this demo.

### Case3. Using @Qualifier

In some situations we may have more than one bean of the same type configured in the container. To have more than one bean is not a problem unless developers control the dependency injection. It may also be possible to have different id of the dependency and having more than one bean of the same type but none of the id is matching. Let's use `@Qualifier` to overcome these problems. We will use the same class `Customer_Autowired` from previous step:

1. Rename the id of bean `cust_address` to `cust_address1` which we created earlier.
2. Add one more bean of type `Address` in it as shown below:

```
<bean id=""address"" class=""com.ch02.beans.Address"">
    <property name=""build_no"" value=""2"" />
    <property name=""city_name"" value=""Pune"" />
    <property name=""pin_code"" value=""123"" />
</bean>
```

3. Create `TestCustomer_Autowiring1` to get the customer instance as shown below:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context = new
        ClassPathXmlApplicationContext("customer_new.xml");
    Customer_Autowired customer = (Customer_Autowired)
        context.getBean("customer_Autowired");
    System.out.println(customer.getCustomerName() + "\t" +
        customer.getCustomerID() +
        customer.getCustomerAddress());
```

```
}
```

4. On execution we will get the following exception specifying there are multiple instances of type Address available for injection leading to ambiguity:

Caused by:  
org.springframework.beans.factory.NoUniqueBeanDefinitionException  
:No qualifying bean of type [com.ch02.beans.Address] is defined  
expected single matching bean but found 2: cust\_address1,ad

5. Update the annotation of data member cust\_address with @Qualifier as shown below:

```
@Qualifier(value = "address")  
private Address cust_address;
```

We are specifying the id of bean which we want to inject to fulfill the dependency. In our case the bean whose id is "address" will be injected.

6. Execute main function which we had created in step 2 to get the following output which shows beans with id as 'address' gets injected in Customer\_Autowried:

```
INFO: Loading XML bean definitions from class path resource  
[customer_new.xml]  
my name 10 Pune 2
```

### Case 3. Using @Required

While developing the code the business logic will fail if the correct values have not been provided or if those are absent. So it's must to have dependency injected and whatever happened it should not be failed. To make sure the dependency has been injected and ready for use apply @Required after @Autowired annotation. It works in the same way 'autowired=true'. But in contrast to it, the annotation can be applied only to the setter methods. If the dependency is not available BeanInitializationException will be thrown.

Following is the code which gives the clear understanding of the use of

@Required:

```
@Component(value = "cust_required")
public class Customer_Annot_Required {

    private String cust_name;
    private int cust_id;

    private Address cust_address;

    public void setCust_name(String cust_name) {
        this.cust_name = cust_name;
    }

    public void setCust_id(int cust_id) {
        this.cust_id = cust_id;
    }

    @Autowired
    @Required
    public void setCust_address(Address cust_address) {
        this.cust_address = cust_address;
    }

    public Customer_Annot_Required() {
        // TODO Auto-generated constructor stub
        cust_id=10;
        cust_name = "my name";
    }
    // getter methods
}
```

#### Case 4 Scope management annotations

Few pages back we had discussed in depth about the scopes and their use. We also had seen how to manage scope according to the requirements in the XML. Usually the `@Scope` is class level annotation. But it also can be used with Bean annotation to a method which indicates the Scope of the instance returned from it.

Following is the code indicates use of `@Scope`:

```
package com.ch02.scope.annotation;
```

```

@Component
@Scope(scopeName = "prototype")
public class Customer_Scope {

    private String cust_name;
    private int cust_id;

    @Autowired
    @Qualifier(value = "address")
    private Address cust_address;

    public Customer_Scope() {
        // TODO Auto-generated constructor stub
        cust_id=10;
        cust_name = "my name";
    }
    //getters and setters
}

```

You can use the following code to check whether each time a new instance is been provided or not:

```

public class TestCustomer_Scope {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context = new
            ClassPathXmlApplicationContext("customer_new.xml");
        Customer_Scope customer = (Customer_Scope)
            context.getBean("customer_Scope");
        System.out.println(customer.getCust_name() + "\t" +
            customer.getCust_id() +
            customer.getCust_address());
        customer.setCust_name("new name set");
        Customer_Scope customer1 = (Customer_Scope)
            context.getBean("customer_Scope");
        System.out.println(customer1.getCust_name() + "\t" +
            customer1.getCust_id() +
            customer1.getCust_address());
        System.out.println("after changing name and using prototype
            scope");
        System.out.println(customer.getCust_name() + "\t" +
            customer.getCust_id() +
            customer.getCust_address());
        System.out.println(customer1.getCust_name() + "\t" +
            + customer1.getCust_id()+
            customer1.getCust_address());
    }
}

```

```
}
```

You can refer the complete code form Ch02\_Demo\_Annotation.

## JSR standard Annotation

The Spring 2.5 supports JSR-250 and 3.0 onwards the framework started supporting JSR-330 standard annotations whose discovery is been done in the same way as that of Spring based annotations. JSR provides annotations for lifecycle management, bean configuration, auto wiring and for many more such requirements. Let's discuss them one by one.

### Annotations for Life cycle

We had discussed enough about the life cycle of a bean and what are different ways to achieve it by XML configurations? But we haven't discussed about annotation based. Spring 2.5 onward support the life cycle management by `@PostConstruct` and `@PreDestroy` which gives an alternative to `InitializingBean` and `Disposable` interfaces respectively.

#### **@PostConstruct:**

The method which has been annotated with `@PostConstruct` get called after the bean which has been instantiated using the default constructor by the container. This method is called just before the instance is returned.

#### **@PreDestroyed:**

The method annotated with `@PreDestroy` will be invoked just before the bean is getting destroyed which allows taking the values back up of the values from the resource which may get lost if the object is destroyed.

Let's follow the steps to use it to understand bean lifecycle:

1. In Ch02\_Bean\_Life\_Cycle java project which we already had used while discussing bean life add `Car_JSR` class in `com.ch02.beans` package.
2. Add a method `init_car` for initiliazation of Car and annotate it with

**@PostConstruct** as shown below:

```
@PostConstruct  
public void init_car()  
{  
    System.out.println("initializing the car");  
    price=(long)(price+(price*0.10));  
}
```

3. Add to the class a method which will have the code of destruction of car which is nothing but resource release as shown below:

```
@PreDestroy  
public void destroy_car()  
{  
    System.out.println("demolishing the car");  
}
```

4. Add beans\_new.xml to configure the bean and don't forget the rule of using annotations in the application. The XML will look like:

```
<context:annotation-config>  
    <bean id=""car"" class=""com.ch02.beans.Car""  
        scope=""prototype"">  
        <property name=""chesis_number"" value=""eng2012"" />  
        <property name=""color"" value=""baker's chocolate"" />  
        <property name=""fuel_type"" value=""petrol"" />  
        <property name=""average"" value=""12.50"" />  
        <property name=""price"" value=""643800"" />  
    </bean>
```

5. Add a main method in TestCar as shown below:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
        ClassPathXmlApplicationContext("beans.xml");  
    // first request to get the car instance  
    Car_JSR car_one=(Car_JSR)context.getBean("car");  
    car_one.show();  
}
```

6. On execution of the code we will get the following output which shows the init\_car() got invoked:

```
INFO: Loading XML bean definitions from class path resource  
[beans.xml]  
initializing the car  
showing car eng2012 having color:-baker's chocolate and  
price:-708180
```

7. As the destroy method will get after context got switch the output will not be shown on console. We can use the following code to elegantly shut down of the container:

```
((AbstractApplicationContext)context).registerShutdownHook(
```

## **@Named**

`@Named` annotation is used instead of `@Component` which is applied to the class level. `@Named` annotation doesn't provide composable model but the scanning of it is in the same way as that of `@Component`:

```
@Named  
public class Customer {  
  
    private String cust_name;  
    private int cust_id;  
  
    private Address cust_address;  
    //default and parameterize constructor  
    //getters and setters  
}
```

## **@Inject**

`@Inject` is used for auto wiring the dependencies in the same way as we do for `@Autowired`. But it won't has attribute "required" to specify the dependency is optional. The following code shows the implementation:

```
@Named  
public class Customer {  
  
    private String cust_name;  
    private int cust_id;  
  
    @Inject  
    private Address cust_address;  
    // default and parameterized constructor
```

```
    // getters and setters  
}
```

## **@Configuration**

The class level `@Configuration` annotation is used to indicate the class as source of the bean definition the same way as we do in XML file within `<beans></beans>` tags.

```
@Configuration  
public class MyConfiguration {  
  
    // bean configuration will come here  
  
}
```

The Spring annotations as `@Component`, `@Repository` needs registration with the framework. In XML scanning of these annotation is enabled by providing `<context:component-scan>` with an attribute `base-package`. `@Component-Scan` is an alternative to enable scanning of Spring stereotype annotations. The syntax is as shown below:

```
@Component-Scan (basePackage=name_of_base_package)
```

## **@Bean**

The `@Bean` is used with `@Configuration` to declare bean definition as we usually do in `<bean></bean>` tag in XML. It's applicable to the method who has the bean definition. The code can be as follow:

```
@Configuration  
public class MyConfiguration {  
    @Bean  
    public Customer myCustomer()  
    {  
        Customer customer=new Customer();  
        customer.setCust_name("name by config");  
        return customer;  
    }  
}
```

To define a bean returned from `myCustomer()` can also have customised bean

`id` which can be given by:

```
@Bean(name = "myCustomer")
```

The XML has been replaced by the annotation based configuration. The `AnnotationConfigApplicationContext` class helps in loading the configuration in the same way we were doing by `ClasspathXmlApplicationContext`. The test class can be written as:

```
public static void main(String[] args) {  
    ApplicationContext context=new  
        AnnotationConfigApplicationContext(MyConfiguration.class);  
    Customer customer=(Customer)context.getBean("myCustomer");  
    System.out.println(customer.getCust_id()+"\t"+  
        customer.getCust_name());  
}
```

You can refer the complete code from [Ch02\\_Demo\\_JSR\\_Annot](#).

## Note

XML provides centralized way to do the bean configurations. As the dependencies are kept out of source code the change in them will not affect the source code. Also in XML configuration the source code doesn't need to be recompiled. But annotation based configuration is directly a part of source code and scattered throughout the application. It becomes decentralised which ultimately become difficult to control. Annotation based injected values will be overridden by the XML injection as annotation based injections take place before the XML injection.

## Spring Expression Language (SpEL)

Up till now we configured the values which are secondary types. For wiring the beans we used the XML configuration which we wrote and available at compile time. But using it we cannot able to configure the runtime values. Spring Expression provides the required solution whose values will be evaluated at runtime.

Using SpEL the developers can reference to other beans using id's and set literal values as well. It provides a means to invoke methods and properties of the objects. SpEL unable evaluation values using mathematical, relational and logical operators. It also supports and injection of Collection. It is somewhat similar to using EL in JSP. The sysntax to use SpEL is "#{value}". Let's find out how to use SpEL in the application one by one.

## Using Literals

Using literals in SpEL unable the developers to set primitive values for the properties of the bean. Though using literals in configuration doesn't interesting but to know how to use in expression certainly help us to go for the complex expressions. Follow the steps to understand use of literals,

1. Create Ch02\_SpringEL as Java application and add to it Spring jars.
2. Define Customer.java in com.ch02.beans as shown below:

```
public class Customer {  
    private String cust_name;  
    private String cust_id;  
    private boolean second_handed;  
    private double prod_price;  
    private String prod_name;  
    // getters and setters  
    // toString()  
}
```

3. Create beans.xml in class path to configure Customer bean as shown below:

```
<bean id=""cust_new"" class="com.ch02.beans.Customer">  
    <property name=""cust_name"" value=""Bina"" />  
    <property name=""cust_id"" value="#{2}" />  
    <property name=""prod_name"" value="#{'Samsung Fridge'}" />  
    <property name=""prod_price"" value="#{27670.50}" />  
    <property name=""second_handed"" value="#{false}" />  
</bean>
```

You can observe the values for cust\_id, prod\_price configured using SpEL syntax as "#{ value}" and we use single quote to specify the value for prod\_name. You even can use double quotes for specifying String values.

The `cust_name` has been configured in old style. Yes, it's still possible to use old style and SpEL together to set the values.

4. Add `TestCustomer.java` as shown in the code below:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
    ClassPathXmlApplicationContext("beans.xml");  
    Customer customer=(Customer)context.getBean("cust_new");  
    System.out.println(customer);  
}
```

5. We will get the output as:

```
INFO: Loading XML bean definitions from class path resource  
[beans.xml]  
Bina has bought Samsung Fridge for Rs 27670.5
```

## Using bean reference

Each bean configured in the XML has its unique bean id. This can be used to refer for value or for auto wiring using SpEL. Use the following steps to understand how to use bean reference.

1. In the above `Ch02_SpringEL` project add new POJO class as `Address` and `Customer_Reference` in `com.ch02.beans` as shown below:

```
public class Address {  
    private String city_name;  
    private int build_no;  
    private long pin_code;  
  
    // getter and setter  
    @Override  
    public String toString() {  
        // TODO Auto-generated method stub  
        return city_name+ "", ""+pin_code;  
    }  
}  
public class Customer_Reference {  
    private String cust_name;  
    private int cust_id;
```

```

private Address cust_address;

// getter and setter

@Override
public String toString() {
    return cust_name + " is living at " + cust_address;
}
}

```

2. Add Address and Customer\_Reference beans in beans.xml as shown below:

```

<bean id=""cust_address"" class=""com.ch02.beans.Address"">
    <property name=""build_no"" value=""2"" />
    <property name=""city_name"" value=""Pune"" />
    <property name=""pin_code"" value=""123"" />
</bean>
<bean id="cust_ref" class=""com.ch02.beans.Customer_Reference">
    <property name=""cust_name"" value=""Bina"" />
    <property name=""cust_id"" value="#{2}" />
    <property name=""cust_address"" value="#{cust_address}"" />
</bean>

```

Observe the way `cust_address` has been initialized. The customer's address is placed value as `cust_address` which is the id of the bean defined for Address.

3. Define TestCustomer\_Reference as we did in Step 4 of previous case to get the bean 'cust\_ref'.
4. On execution we will get the out put as shown below:

```

INFO: Loading XML bean definitions from class path resource
[beans.xml]
Bina is living at Pune,123

```

5. Sometimes instead of using the bean we may want to inject only one of the property of the bean as shown below:

```

<bean id="cust_ref_new" class="com.ch02.beans.Customer_Reference">
    <property name="cust_name"
              value="#{cust_ref.cust_name.toUpperCase()}" />
    <property name="cust_id" value="#{2}" />
    <property name="cust_address" value="#{cust_address}" />
</bean>

```

We injected `cust_name` by borrowing it from bean '`cust_ref`' and converting it to upper case.

## Using mathematical, logical, relational operators

In some of the scenarios the values of the dependencies needs mathematical, logical or relational operators to define and place values of the dependencies. You can find how to use them in the following demo.

1. We will use the same project Ch02\_SpringEL defined in previous case.
2. Define a bean for Customer in beans.xml as shown below:

```
<bean id=""cust_calculation"" class="com.ch02.beans.Customer">
    <property name="cust_name" value="Bina" />
    <property name="cust_id" value="#{2}" />
    <property name="prod_name" value="#{'Samsung Fridge'}" />
    <property name="prod_price" value="#{27670.50*12.5}" />
    <property name="second_handed"
        value="#{cust_calculation.prod_price > 25000}" />
</bean>
```

The value of `prod_price` is calculated at runtime with the help of mathematical operator and value for "`second_handed`" evaluated by relational operator.

3. Write `TestCustomer_Cal` to get `cust_calculation` and display its properties as follows:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new
        ClassPathXmlApplicationContext("beans.xml");
    Customer customer=
        (Customer)context.getBean("cust_calculation");
    System.out.println(customer);
    System.out.println(customer.isSecond_handed());
}
```

Throughout this chapter we had seen how to do the configuration. But we missed a very important point "loose coupling". Everywhere we use classes

which leads for tight coupling and we know programming by contract gives developers the facility to write loosely coupled modules. We avoid use of interfaces for above all demos because it may make them unnecessarily complex. But in the real time applications you will find use of interfaces on each stage in Spring. From next chapter onwards we will use standard way of writing Spring based applications.

# Summary

---

This chapter is full of configurations, different ways and alternative to do the same thing. This chapter helps throughout your journey with Spring no matters whether its 5.0 or below versions. In this chapter we explore range of ways to do instance creation and initializing it. We are now aware of almost all kind of Java Concepts with how to do their configuration both using XML and annotation based. We also seen ways like auto wiring , SpEL to minimise the configuration. We demonstrate Spring provided as well as JSR annotations.

Now we are set to implement all these in our application. Let's start with database handling. In next chapter we will find how to handle database using Spring and how Spring helps us doing faster development.

# Chapter 3. Accelerate with Spring DAO

In the second chapter, we discussed in depth about dependency injection. Obviously, we discussed various ways to use DI in the configuration files as well as using annotations, but still somewhere due to unavailability of real time application it was incomplete. We were not having any choice as these were the most important basics, which each of Spring framework developer should be aware of. Now, we will start with handling database which is the backbone of application using the co

nfigurations which we discussed. In this chapter we will discuss the following points:

- We will discuss about DataSource and its configuration using JNDI, pooled DataSource, and JDBC Driver based DataSource
- We will learn how to integrate database in the application using DataSource and JDBCTemplate
- Understanding ORM and its configuration in application using SessionFactory will be the next point to discuss
- We will configure HibernateTemplate to talk with database using ORM.
- We will cover configuring cache manager to enable the support of caching the data

As we are well aware that, the database gives easy structuring of the data facilitating easy access using various ways. Not only many ways are available but also there are number of databases available in market. On one hand it's good to have many options of databases but on other its complex as each of them needs to handle separately. The Java application on enormous stages needs the persistency for accessing, updating, deleting, adding the records in database. JDBC APIs helps in accessing such records through drivers. JDBC provides the low level database operations as defining, opening and closing the connections, creation of statements, iterating through the ResultSet to fetch the data, processing the exceptions and many more. But at one point, this became repetitive operation and tightly coupled. The Spring Framework gives loosely coupled, high level, clean solution with a range of

customized exceptions using DAO design pattern.

# How Spring handles database?

---

In Java application the developers generally uses a concept of utility class to create, open and close database connection. A pretty sound, smart and reusable way for connection management but still the application is tightly coupled with the utility class. Any change in the either the database or its connectivity as URL, username, password or schema, it need to be changed in the class. This needs to be followed by the recompilation and redeployment of the code. In such scenario externalizing the connection parameters will be a good solution. We cannot externalise the Connection object, that still has to be managed by the developer and so do the exceptions while handling it. Spring has a elegant, loosely coupled ways to manage the connection using the DataSource at centre.

## The DataSource

The DataSource is the factory for data source connections similar to the DriverManager in JDBC who helps for Connection management. Following are some of the implementations which can be used in the application to obtain the Connection object:

- **DriverManagerDataSource** : The `DriverManager` class provides a simple implementation of the DataSource to be used in test or standalone application which enable a new Connection object on every request via `getConnection()`.
- **SingleConnectionDataSource**: The class is an implementation of the SmartDataSource which gives a single Connection object which doesn't get closed after its use. It's useful only in single threaded applications and in making testing easy outside of the application server.
- **DataSourceUtils**: This is a helper class who has static methods for obtaining the database Connection from DataSource.
- **DataSourceTransactionManager**: This class is an implementation of the PlatformTransactionManager for a Connection per data source.

## Configuring DataSource

The DataSource can be configured by following ways in the application:

- **Obtained from JNDI look up :** Spring helps in maintaining large scale JavaEE application which run in application servers like Glassfish, JBoss, Wlblogic, Tomcat. All these servers support facility to configure the pool of data sources which can be acquired by the Java Naming Directory Interface (JNDI) look up helps in better performance. The jee namespace can be used to obtain data source configured in the application as shown below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jee="http://www.springframework.org/schema/jee"
       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.xsd http://www.springframework.org/schema/jee
http://www.springframework.org/schema/jee/spring-jee.xsd"

<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"
              value="java:comp/env/jdbc/myDataSource"/>
  </bean>

  <jee:jndi-lookup jndi-name="jdbc/myDataSource" id="dataSo
      resource-ref="true"/>
</beans>
```

- Where:
- **jndi-name :** specifies name of the resource configured on server in JNDI
- **id :** id of the bean which gives object of DataSource
- **resource-ref :** specifies whether to prefix with java:comp/env or not.
- **Fetched from the pooled connection :** Spring doesn't have pooled data source but, we can configure pooled data source provided by Jakarta Commons Database Connection Pooling. The DBCP provided

BasicDataSource can be configured as,

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
              value="org.hsqldb.jdbcDriver"/>
    <property name="url"
              value="jdbc:hsqldb:hsq1://localhost/name_of_schama"/>
    <property name="username"
              value="credential_for_username"/>
    <property name="password"
              value="credential_for_password"/>
    <property name="initialSize"
              value="" />
    <property name="maxActive"
              value="" />
</bean>
```

- Where:
- **initialSize**: specifies how many connections to be created when the pool is started
- **maxActive**: specifies how many connections can be allocated from the pool.
- Along with these attributes we can even specify the time which needs to wait till the connection is returned from the pool(maxWait), the maximum/ minimum number of connections that can be idle in the pool(maxIdle/ minIdle), the maximum number of prepared statements that can be allocated from the statement pool (maxOperationPreparedStatements).
- With the help of JDBC driver : One can use of the following class to configure the simplest way to get and object of the DataSource:

\* **SingleConnectionDataSource**: As we already discussed it returns single connection.

\* **DriverManagerDataSource**: It returns a new connection object on a request.

- The DriverMangerDataSource configuration can be done as follows:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
    <property name="driverClassName"
      value="org.hsqldb.jdbcDriver"/>
    <property name="url"
      value="jdbc:hsqldb:hsq://localhost/name_of_schama"/>
    <property name="username"
      value="credential_for_username"/>
    <property name="password"
      value="credential_for_password"/>
</bean>
```

## Note

SingleConnectionDataSource is suitable for small single threaded applications. DriverManagerDataSource supports multithreaded application but hampers the performance due to managing number of connections. It's recommended to use pooled data source for better performance.

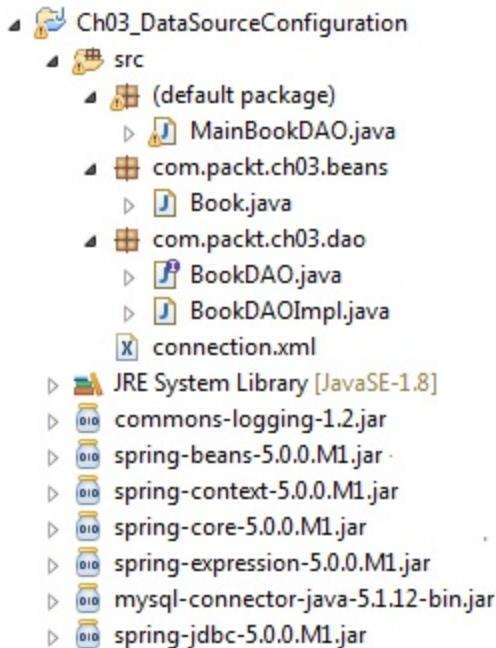
Let's develop a sample demo using loosely coupled modules so that we will understand the practical aspects of Spring framework application development.

The DataSource facilitates handling of connection with database so it needs to be injected in the module. The choice of using setter DI or constructor DI can be totally decided by you as you are well aware of both of these dependency injections. We will use setter DI. Instead of starting from the class we will start by considering interface as it's the best way to do programming by contract. The interfaces can have multiple implementations. So the use of interface and Spring configuration helps to achieve loosely coupled modules. We will declare the methods for database operations, you are free to choose the signature. But make sure that they will be testable. As loose coupling is major feature of the framework the application will also demonstrate why we keep on saying loosely coupled modules can be easily written using Spring? You can use any database but we will use MySQL

throughout the book. Whatever the database you choose make sure to install it before moving ahead So let's start by following the steps!

### Case 1: using XML configuration of DriverManagerDataSource

1. Create a Core Java application as Ch03\_DatasourceConfiguration and add to it jar for Spring and JDBC as shown in the outline of the application below:



2. Create a Book POJO in com.ch03.packt.beans package as shown below:

```
public class Book {  
    private String bookName;  
    private long ISBN;  
    private String publication;  
    private int price;  
    private String description;  
    private String [] authors;  
  
    public Book() {  
        // TODO Auto-generated constructor stub  
        this.bookName="Book Name";  
        this.ISBN =985645671;  
        this.publication="Packt Publication";  
    }  
}
```

```

        this.price=200;
        this.description="this is book in general";
        this.author="ABCD";
    }

    public Book(String bookName, long ISBN, String
        publication,int price,String description,String
        author)
    {
        this.bookName=bookName;
        this.ISBN =ISBN;
        this.publication=publication;
        this.price=price;
        this.description=description;
        this.author=author;
    }
    // getters and setters
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return bookName+"\t"+description+"\t"+price;
    }
}

```

3. Declare an interface BookDAO in com.ch03.packt.dao package. (DAO means Data access Object).
4. Add to it a method for adding a book to the database. The code will be as shown below:

```

interface BookDAO
{
    public int addBook(Book book);
}

```

5. Create an implementation class for BookDAO as BookDAOImpl and add a data member of type DataSource in the class as follows:

```
private DataSource dataSource;
```

- Don't forget to use standard bean naming conventions.
6. As we are following setter injection write or generate setters for DataSource.
  7. The overridden method will deal with getting the connection from

DataSource and using PreparedStatement insert a book object in the table as we do in JDBC as shown in the code below:

```
public class BookDAOImpl implements BookDAO {  
    private DataSource dataSource;  
  
    public void setDataSource(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    @Override  
    public int addBook(Book book) {  
        // TODO Auto-generated method stub  
        int rows=0;  
        String INSERT_BOOK="insert into book values(?, ?, ?, ?, ?, ?, ?)  
        try {  
            Connection connection=dataSource.getConnection();  
            PreparedStatement ps=  
                connection.prepareStatement(INSERT_BOOK);  
            ps.setString(1,book.getBookName());  
            ps.setLong(2,book.getISBN());  
            ps.setString(3,book.getPublication());  
            ps.setInt(4,book.getPrice());  
            ps.setString(5,book.getDescription());  
            ps.setString(6,book.getAuthor());  
            rows=ps.executeUpdate();  
        } catch (SQLException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
        return rows;  
    }  
}
```

8. Create connection.xml in classpath to configure the beans.
9. Now the question will be how many beans to be declared and what has to be their id's?

## Note

A very simple thumb rule: First find out class to configure and then what are its dependencies?

Here:

- \* A bean for BookDAOImpl
- \* BookDAOImpl has DataSource as a dependency so a bean for DataSource.

You will be wondering that DataSource is an interface! So, how could we create and inject its object? Yes, this is what our point is! This is what we are talking about loosely coupled modules. The DataSource implementation which we are using here is, DriverManagerDataSource. But, if we inject directly DriverManagerDataSource then the class will be tightly coupled on it. Also, if tomorrow instead of using DriverManagerDataSource the team decides to use some other implementation then the code has to be changed which leads to recompilation and redeployment. That mean the better solution will be using interface and injecting its implementation from the configuration.

The id's can be of developers choice, but don't neglect to take the advantage of auto wiring and then accordingly set the id. Here we will use auto wiring 'byName' so choose the id's accordingly. (If you are confused or want to dig about auto wiring you can refer the previous chapter.) So the final configuration in XML will be as follows:

```
<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/bookDB" />
  <property name="username" value="root" />
  <property name="password" value="mysql" />
</bean>

<bean id="bookDao" class="com.packt.ch03.dao.BookDAOImpl"
  autowire="byname">
</bean>
```

You use may need to customize the URL, username, password to match your connection parameters.

1. Normally, the DAO layer will be invoked by the Service layer but here we are not dealing with it as the application proceed we will add it. As we yet not discussed about testing we will write a code with main function to find out the output of it. The main function will get BookDAO bean and invoke on it a method to insert the Book. If the value of the row returned by the implementation code is greater than zero the book got successfully added otherwise not. Create a class MainBookDAO and add the following code to it:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
        ClassPathXmlApplicationContext("connection.xml");  
    BookDAO bookDAO=(BookDAO) context.getBean("bookDao");  
    int rows=bookDAO.addBook(new Book("Learning Modular  
        Java Programming", 9781234,"PacktPub  
        publication",800,"explore the power of  
        Modular programming","T.M.Jog"));  
    if(rows>0)  
    {  
        System.out.println("book inserted successfully");  
    }  
    else  
        System.out.println("SORRY!cannot add book");  
}
```

If you keenly observed though we configure object of BookDAOImpl we are accepting it in BookDAO interface which helps in writing flexible code where the main code is unaware of in actual whose object is giving implementation.

2. Open your MYSQL console, use credentials to login. Fire query to create BookDB schema and Book table as shown below:

```
mysql> create database bookDB;  
Query OK, 1 row affected (0.00 sec)  
  
mysql> use bookDB;  
Database changed  
mysql> create table book(bookName varchar(50),ISBN bigint,publication varchar(50)  
,price int,description varchar(100),author varchar(50),PRIMARY KEY(ISBN));  
Query OK, 0 rows affected (0.39 sec)  
  
mysql> _
```

3. All set to execute the code to get "book inserted successfully" on

console. You can fire "select \* from book" in MySQL to get the book details as well.

### Case2: Using Annotations for DriverManagerDataSource

We will use the same Java application Ch03\_DataSourceConfiguration developed in Case1:

1. Declare a class BookDAO\_Annotation implementing BookDAO in com.packt.ch03.dao package and annotate it with @Repository as it deals with database and specify id as 'bookDAO\_new'.
2. Declare a data member of type DataSource and annotate it with @Autowired to support auto wiring.
  - Don't forget to use standard bean naming conventions.
  - The overridden method will deal database to insert book in table. The code will be as shown below:

```
@Repository(value="bookDAO_new")
public class BookDAO_Annotation implements BookDAO {
    @Autowired
    private DataSource dataSource;

    @Override
    public int addBook(Book book) {
        // TODO Auto-generated method stub
        int rows=0;
        // code similar to insertion of book as shown in
        Case1.
        return rows;
    }
}
```

3. We can edit the same connection.xml from Case1 but that may complex the configuration. So, let's create connection\_new.xml in classpath to configure the instructions for container to consider annotation and search for stereotype annotations as follows:

```
<context:annotation-config/>
<context:component-scan base-
    package="com.packt.ch03.*"></context:component-scan>
```

```

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
    <!--add properties similar to Case1 -- >
</bean>

```

- To find out addition of context namespace and using annotation you can refer to second chapter.

#### 4. It's now time to find the output with the help of following code:

```

public class MainBookDAO_Annotation {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new
            ClassPathXmlApplicationContext("connection_new.xml");

        BookDAO bookDAO=(BookDAO) context.getBean("bookDAO_new"
        int rows=bookDAO.addBook(new Book("Learning Modular Java
            Programming", 9781235L,"PacktPub
            publication",800,"explore the power of
            Modular programming","T.M.Jog"));
        if(rows>0)
        {
            System.out.println("book inserted successfully");
        }
        else
            System.out.println("SORRY!cannot add book");
    }
}

```

Execute the code to add the book in database.

You may have observed that, we never come to know who is providing the implementation of the JDBC code and the injection is loosely coupled due to configuration. Though, we will be able to insert the data in database we will be still involved in the JDBC code like getting Connection, creating Statements from it and then setting the values for the columns of table where we want to insert the records. This is very preliminary demo which is rarely used in JavaEE applications. The better solution is to use template classes provided by Spring.

# **Using Template classes to perform JDBC operations**

Template classes provides an abstract way to define operations by giving rid from common issues as opening and maintaining the connection, the boiler plate code as getting the Statement objects. Spring provides many such template classes where dealing with the JDBC, JMS and Transaction management becomes easy than ever before. JdbcTemplate is one of such core component by Spring who helps in handling JDBC. To handle JDBC we can use any one of the following three templates.

## **JDBCTemplate**

The JdbcTemplate helps the developers to concentrate on the core business logic of the application without getting involved in how to open or mange the Connection. They don't have to be worried about what if they forget to release connection? All these things will be elegantly done by the JdbcTemplate for you. It provides specifying indexed parameters to use in SQL queries for JDBC operations as we normally do in PreparedStatements.

## **SimpleJdbcTemplate**

This is very similar to JDBCTemplate along with an advantage of Java5 features as generics, var-args, autoboxing.

## **NamedParameterJdbcTemplate**

The JdbcTemplate uses index to specify the values of the parameters in SQL which may become complicate to remember the parameters with their indexes. If you are uncomfortable with numbers or more number of parameters to set we can use the NamedParamterJdbcTemplate who facilitates use of named-parameters to specify parameters in the SQL. Each parameter will have a named prefixed with a colon(:). We will see the syntax shortly while developing the code.

Let's demonstrate these templates one by one.

## Using JdbcTemplate

We will use the similar outline of the project used in Ch03\_DataSourceConfiguration and redevelop it with the help of following steps,

1. Create a new Java application named Ch03\_JdbcTemplate and add jar which we used in Ch03\_DataSourceIntegration. Along with it add spring-tx-5.0.0.M1.jar as well.
2. Create or copy Book in com.packt.ch03.beans package.
3. Create or copy BookDAO in com.packt.ch03.dao package.
4. Create BookDAOImpl\_JdbcTemplate in com.packt.ch03.dao package and add to it JdbcTemplate as data member.
5. Annotate the class with @Repository and data member JdbcTemplate with @Autowired annotation respectively.
6. The overridden methods will deal with insertion of Book in table. But we don't have to get the connection. Also we will not do creation and setting the parameters of the PreparedStatement. The JdbcTemplate will do it for us. Things will be pretty clear from the code shown below:

```
@Repository (value = "bookDAO_jdbcTemplate")
public class BookDAO_JdbcTemplate implements BookDAO {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @Override
    public int addBook(Book book) {
        // TODO Auto-generated method stub
        int rows = 0;
        String INSERT_BOOK = "insert into book
            values(?,?,?,?,?,?)";

        rows=jdbcTemplate.update(INSERT_BOOK, book.getBookName(
            book.getPrice(),book.getDescription(),
            book.getAuthor()));
        return rows;
    }
}
```

- The JdbcTemplate has update() method where the developers need to pass the SQL query followed by the values of the parameters in the

query. So, we can use it for insertion, updation as well as deletion of data. Rest all will be done by the template. If you keenly observe, we are not handling any exceptions. We forgot to? No, we don't care to handle them as Spring provides the `DataAccessException` which is unchecked exception. So leave the worries. In the upcoming pages we will discuss about the exceptions Spring provides.

- Let's add method in the code for updating book's price as well as deleting the book. Don't forget to change the interface implementation first. The code is as shown below:

```
@Override  
public int updateBook(long ISBN, int price) {  
    // TODO Auto-generated method stub  
    int rows = 0;  
    String UPDATE_BOOK = "update book set price=? where ISBN=";  
  
    rows=jdbcTemplate.update(UPDATE_BOOK, price,ISBN);  
    return rows;  
}  
  
@Override  
public boolean deleteBook(long ISBN) {  
    // TODO Auto-generated method stub  
    int rows = 0;  
    boolean flag=false;  
    String DELETE_BOOK = "delete from book where ISBN=?";  
  
    rows=jdbcTemplate.update(DELETE_BOOK, ISBN);  
    if(rows>0)  
        flag=true;  
  
    return flag;  
}
```

7. Let's add beans configuration file as `connection_new.xml`. You can simply copy it from `Ch03_DataSourceIntegration` project. We are using `JdbcTemplate` who has `DataSource` as its dependency. So, we need to configure two beans one for `DataSource` and another for `JdbcTemplate` as shown below:

```
<context:annotation-config/>  
<context:component-scan base-
```

```

package="com.packt.ch03.*"></context:component-scan>

<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/bookDB" />
    <property name="username" value="root" />
    <property name="password" value="mysql" />
</bean>

<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"></property>
</bean>

```

8. Write the code to get 'bookDAO\_jdbcTemplate' bean and execute the operations in MainBookDAO\_operations as shown below:

```

public class MainBookDAO_operations {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new
            ClassPathXmlApplicationContext("connection_new.xml");
        BookDAO bookDAO=(BookDAO)
            context.getBean("bookDAO_jdbcTemplate");
        //add book
        int rows=bookDAO.addBook(new Book("Java EE 7 Developer
            Handbook", 97815674L,"PacktPub
            publication",332,"explore the Java EE7
            programming","Peter pilgrim"));
        if(rows>0)
        {
            System.out.println("book inserted successfully");
        }
        else
            System.out.println("SORRY!cannot add book");
        //update the book
        rows=bookDAO.updateBook(97815674L,432);
        if(rows>0)
        {
            System.out.println("book updated successfully");
        }
        else
            System.out.println("SORRY!cannot update book");
    }
}

```

```

        //delete the book
        boolean deleted=bookDAO.deleteBook(97815674L);
        if(deleted)
        {
            System.out.println("book deleted successfully");
        }
        else
            System.out.println("SORRY!cannot delete book");
    }
}

```

## Using NamedParameterJdbc Template

We will use Ch03\_JdbcTemplate to add new class for this demonstration with the help of following steps.

1. Add BookDAO\_NamedParameter class in com.packt.ch03.dao package which is implementing BookDAO and annotate it with @Repository as we did earlier.
2. Add to it NamedParameterJdbcTemplate as a data member and annotate it with @Autowired.
3. Implement the overridden methods to perform JDBC operations with the help of update(). The NamedParameterJdbcTemplate supports giving names to the parameters in SQL query. Find the below query to add Book:

```

String INSERT_BOOK = "insert into book
values (:bookName, :ISBN, :publication, :price, :description,
: author)";

```

## Note

Each parameter has to be prefixed with colon as :name\_of\_parameter.

- If these are the names of the parameters then these have to be registered so that the framework will place them in the query. To do this we have to create a Map where these parameter names acts as keys whose values will be specified by the developer. The following code will give a clear

idea:

```
@Repository(value="BookDAO_named")
public class BookDAO_NamedParameter implements BookDAO {

    @Autowired
    private NamedParameterJdbcTemplate namedTemplate;

    @Override
    public int addBook(Book book) {
        // TODO Auto-generated method stub
        int rows = 0;
        String INSERT_BOOK = "insert into book
            values (:bookName,:ISBN,:publication,:price,
            :description,:author)";
        Map<String, Object>params=new HashMap<String, Object>();
        params.put("bookName", book.getBookName());
        params.put("ISBN", book.getISBN());
        params.put("publication", book.getPublication());
        params.put("price", book.getPrice());
        params.put("description", book.getDescription());
        params.put("author", book.getAuthor());
        rows=namedTemplate.update(INSERT_BOOK, params);

        return rows;
    }

    @Override
    public int updateBook(long ISBN, int price) {
        // TODO Auto-generated method stub
        int rows = 0;
        String UPDATE_BOOK =
            "update book set price=:price where ISBN=:ISBN";

        Map<String, Object>params=new HashMap<String, Object>();
        params.put("ISBN", ISBN);
        params.put("price", price);
        rows=namedTemplate.update(UPDATE_BOOK, params);
        return rows;
    }

    @Override
    public boolean deleteBook(long ISBN) {
        // TODO Auto-generated method stub
        int rows = 0;
        boolean flag=false;
        String DELETE_BOOK = "delete from book where ISBN=:ISBN
```

```

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("ISBN", ISBN);
        rows = namedTemplate.update(DELETE_BOOK, params);
        if (rows > 0)
            flag = true;
        return flag;
    }
}

```

4. Add a bean in connection\_new.xml for NamedParameterJdbcTemplate as follows:

```

<bean id="namedTemplate"
      class="org.springframework.jdbc.core.namedparam.
      NamedParameterJdbcTemplate">
    <constructor-arg ref="dataSource"/>
</bean>

```

- In all other demos we have used setter injection but here we cannot use setter injection, as the class doesn't have default constructor. So, use constructor dependency injection only.
5. Use the MainBookDAO\_operations.java developed to test how JdbcTemplate work. You just have to update the statement which will get **BookDAO\_named** bean to execute operations. The changed code will be:

```
BookDAO bookDAO = (BookDAO) context.getBean("BookDAO_named");
```

- You can find the complete code in MainBookDAO\_NamedTemplate.java
6. Execute the code to get success message.

In small Java application the code will have less number of DAO classes. So, to the developers handling DAOs with Template classes in each of them to handle JDBC will not be complex. This also leads to duplication of the code. But the complexity becomes difficult to handle when we deal with enterprise applications with more number of classes. The alternative will be instead of injecting the template class in each of the DAO, choose a parent who has the

ability as that of Template classes. Spring has JdbcDaoSupport, NamedParameterJdbcSupport as such supportive DAOs. These abstract support class provide a common base leaving out repetition of the code, wiring of properties in each DAO.

Let's take the same project ahead to use supportive DAOs. We will use JdbcDaoSupport class to understand the practical aspects:

1. Add BookDAO\_JdbcTemplateSupport.java in com.packt.ch03.dao which extends JdbcDaoSupport and implementing BookDAO.
2. Override the methods from interface which will deal with database. The BookDAO\_JdbcTemplateSupport class inherits JdbcTemplate from JdbcDaoSupport. So the code remains same as we did in using JdbcTemplate with a little change. The JdbcTemplate has to be accessed through getter method as shown by underlining in the code below:

```
@Repository(value="daoSupport")
public class BookDAO_JdbcTemplateSupport extends JdbcDaoSupport
    implements BookDAO
{
    @Autowired
    public BookDAO_JdbcTemplateSupport(JdbcTemplate jdbcTemplate)
    {
        setJdbcTemplate(jdbcTemplate);
    }

    @Override
    public int addBook(Book book) {
        // TODO Auto-generated method stub
        int rows = 0;
        String INSERT_BOOK = "insert into book values(?, ?, ?, ?, ?, ?"

        rows=getJdbcTemplate().update(INSERT_BOOK,
            book.getBookName(), book.getISBN(),
            book.getPublication(), book.getPrice(),
            book.getDescription(), book.getAuthor());

        return rows;
    }

    @Override
    public int updateBook(long ISBN, int price) {
        // TODO Auto-generated method stub
        int rows = 0;
```

```

String UPDATE_BOOK = "update book set price=? where ISBN=?";

rows=getJdbcTemplate().update(UPDATE_BOOK, price,ISBN);
return rows;
}

@Override
public boolean deleteBook(long ISBN) {
    // TODO Auto-generated method stub
    int rows = 0;
    boolean flag=false;
    String DELETE_BOOK = "delete from book where ISBN=?";

    rows=getJdbcTemplate().update(DELETE_BOOK, ISBN);
    if(rows>0)
        flag=true;

    return flag;
}
}

```

## Note

To use DAO classes the dependencies will be injected through constructor.

We already had discussed couple of pages back about handling the exceptions in short. Let's find out more in detail about it. The JDBC code forces handling of exception through checked exceptions. But, they are generalized and handled only through DataTruncationException, SQLException, BatchUpdateException, SQLWarning. Opposite to JDBC Spring support various unchecked exceptions for different scenarios providing specialized information. The following table shows few of them which we may need frequently:

### Spring Exceptions

DataAccessException

### When they get thrown?

This is the root of the Spring Exception hierarchy we can use it

for all situations.

**PermissionDeniedDataAccessEception**  
When trying to access the data without correct authorization to access the data

**EmptyResultDataAccessException**  
On no row returned from the database but at least one is expected.

**IncorrectResultSizeDataAccessException**  
When the result size is mismatching with the expected result size.

**TypeMismatchDataAccessException**  
On mismatch of the data type between Java and database.

**CannotAccqireLockException**  
On failure to acquire the lock while an update of the data

**DataRetrivalFailureException**  
When searching and retrieving of the data using Id using ORM tool

While handling the database operations using Spring DataSource, Template classes, DAOSupport classes we still are involved in JDBC operations using SQL queries without Object centric operations. The easy way to handle database operations is by keeping Object at the center using Object Relational Mapping.

# Object Relation Mapping

---

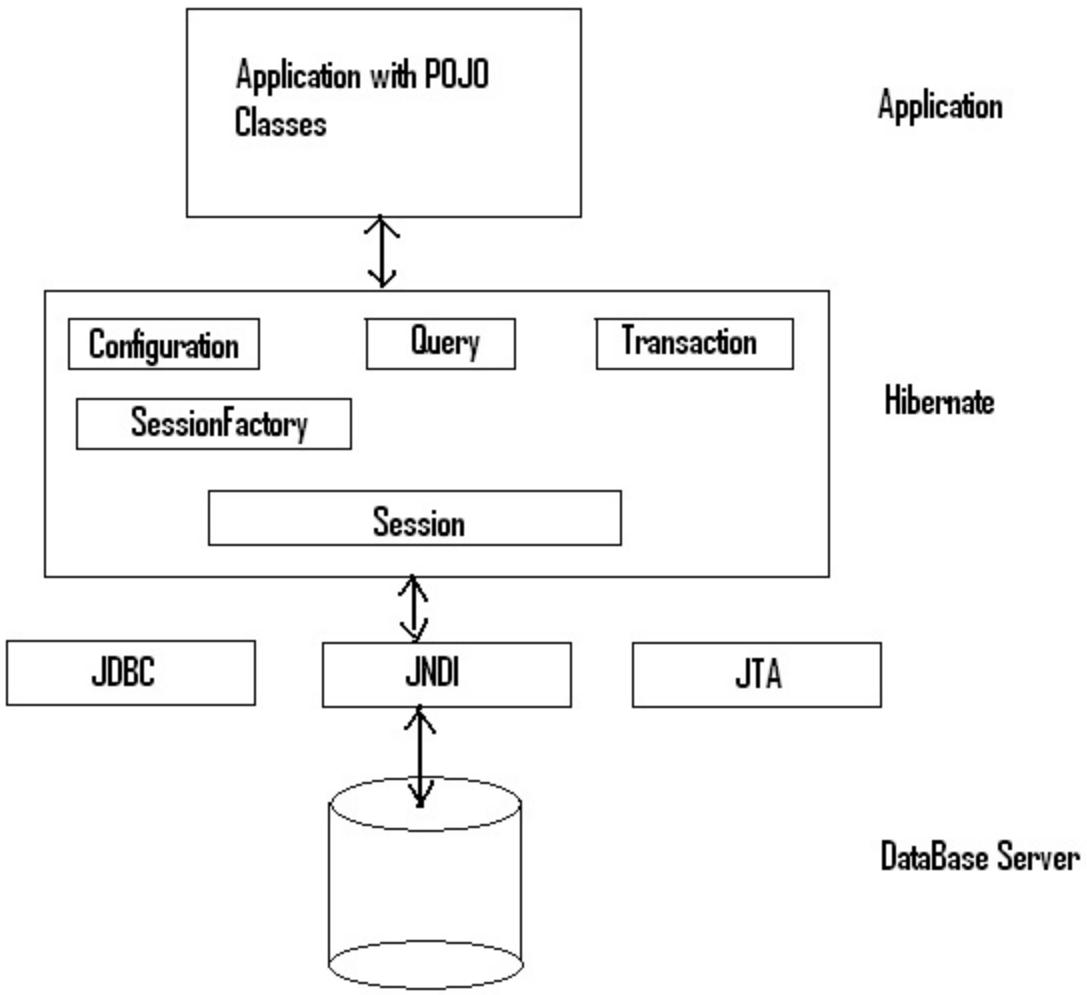
The JDBC APIs gives a means for performing relational database operations to achieve persistency. The Java developers are rigorously involved in writing SQL queries to do such database operations. But Java is Object Oriented Programming Language (OOPs) and database uses **Sequential Query Language (SQL)**. OOPs has object at its centre which SQL has database. OOPs doesn't have Primary Key concept, as it has identity. OOPS uses inheritance but SQL does not have. These and many more mismatches makes JDBC operations difficult to perform without sound hands on knowledge of database and its structure. A good solution has been provided by the ORM tools. ORM deals with database operations keeping object at centre developers without asking developers to deal with SQL. The iBATIS, JPA, Hibernate are some of the ORM frameworks in the market.

## Hibernate

Hibernate is one of the famous middleware tool among the developers for ORM solutions. It provides a solution to problems of granularity, inheritance, identity, relational association and navigation in easy way. The developers doesn't have to hard code the SQL queries as Hibernate provides rich API to deal with CRUD database operations making it more maintainable and easy. The SQL queries are database dependant but in Hibernate there is no need to write SQL as it provides vendor independence. It also supports Hibernate Query Language (HQL) and native SQL support to perform customised database operations by writing queries. Using Hibernate developers can reduce in development time leading to increase in productivity.

### Hibernate Architecture

The following figure shows the architecture of the hibernate and the interfaces in it:



The Hibernate has Session, SessionFactory, Configuration, Transaction, Query, Criteria interfaces at its core helping in providing the ORM support to the developers.

### **Configuration interface**

The instance of Configuration is used to specify the database properties like URL, username and password, the location of mapping files or class containing information about mapping of data members to the tables and their columns. This Configuration instance is then used to obtain instance of the SessionFactory.

### **SessionFactory interface**

`SessionFactory` is heavy weight and every application typically has single instance per application. But sometimes an application uses more than one database which leads to one instance per database. `SessionFactory` is used to obtain the instance of `Session`. It is very important as it caches the generated SQL statements and the data generated which Hibernate uses at runtime in one unit of work as First level cache.

### **Session interface**

The `Session` interface is the basic interface which each Hibernate the application uses to perform database operations obtained from `SessionFactory`. `Session` is light weight, inexpensive component. As `SessionFactory` is per application, the developer creates `Session` instance per request.

### **Transaction interface**

`Transaction` helps the developers to bind number of operations as a unit of work. JTA, JDBC provides the implementation of the `Transaction` implementation. Unless the developers don't commit the transaction, the data won't reflect in database.

### **Query interface**

`Query` interface provides a facility to write queries using Hibernate Query Language(HQL) or native SQL to perform database operations. It also allows developers to bind values to the SQL parameters, specify how many number of results are returned from the query.

### **Criteria interface**

The `Criteria` interface is similar to `Query` interface allows the developers to write criteria query object to get result based on some restrictions or criteria.

In Spring framework developers has the choice of using `SessionFactory` instance or `HibernateTemplate` to do Hibernate integrations. The

SessionFactory is obtained from the configuration of database connectivity parameters and location of the mapping which then using DI can be used in Spring application. The SessionFactory can be configured as shown below:

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>book.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key=
                "hibernate.dialect">org.hibernate.dialect.MySQLDialect
            </prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
```

- **dataSource** - provides information about the database properties.
- **mappingResource**- specifies the name of files which provides information about mapping of the data members to the table and it's columns.
- **hibernateProperties**- provides information about hibernate properties

\* **dialect** - it is used by the framework to generate SQL queries as per the underlying database.

\* **show\_sql** - it displays the SQL query fired by framework on console.

\* **hbm2ddl.auto**- it provides the info whether to create, update the table with which operations to perform.

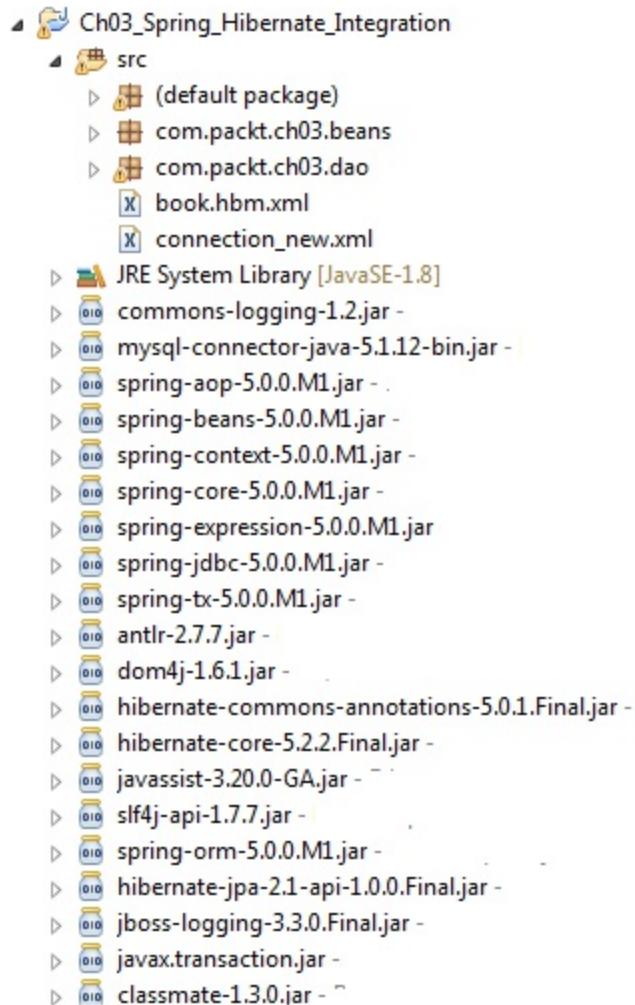
While using SessionFactory the developers are not writing code which uses Spring APIs. But we already had discussed about Template classes few pages back. HibernateTemplate is one of such template which helps developers to write decoupled applications. The HibernateTemplate configuration is as:

```
<bean id="hibernateTemplate" \
      class="org.springframework.orm.hibernate5.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory"></proper
</bean>
```

Let's integrate SessionFactory in our Book project one by one with the help of following steps.

### Case1: Using SessionFactory

1. Create Java application named Ch03\_Spring\_Hibernate\_Integration and add to it jars for Spring, JDBC and hibernate as shown in the outline below:



- You can download the zip containing jar for hibernate framework from official website of Hibernate.
2. Copy or create Book.java in com.packt.ch03.beans package.
  3. Create book.hbm.xml in classpath to map the Book class to the book\_hib table as shown in the configuration below:

```

<hibernate-mapping>
    <class name="com.packt.ch03.beans.Book" table="book_hib">
        <id name="ISBN" type="long">
            <column name="ISBN" />
            <generator class="assigned" />
        </id>
        <property name="bookName" type="java.lang.String">
            <column name="book_name" />
        </property>
        <property name="description" type="java.lang.String">
            <column name="description" />
        </property>
        <property name="author" type="java.lang.String">
            <column name="book_author" />
        </property>
        <property name="price" type="int">
            <column name="book_price" />
        </property>
    </class>

</hibernate-mapping>

```

- Where the configuration of tags are as:

- \* **id**- defines mapping for primary key from table to book class
  - \* **property**- provides mapping of data members to the columns in the table
4. Add BookDAO interface as we did in Ch03\_JdbcTemplate application.
  5. Implement BookDAO by BookDAO\_SessionFactory and override the methods. Annotate the class with `@Repository`. Add a data member of type SessionFactory annotated with `@Autowired`. The code is as shown below:

```

@Repository(value = "bookDAO_sessionFactory")
public class BookDAO_SessionFactory implements BookDAO {

```

```
@Autowired
SessionFactory sessionFactory;

@Override
public int addBook(Book book) {
    // TODO Auto-generated method stub
    Session session = sessionFactory.openSession();
    Transaction transaction = session.beginTransaction();
    try {
        session.saveOrUpdate(book);
        transaction.commit();
        session.close();
        return 1;
    } catch (DataAccessException exception) {
        exception.printStackTrace();
    }
    return 0;
}

@Override
public int updateBook(long ISBN, int price) {
    // TODO Auto-generated method stub
    Session session = sessionFactory.openSession();
    Transaction transaction = session.beginTransaction();
    try {
        Book book = session.get(Book.class, ISBN);
        book.setPrice(price);
        session.saveOrUpdate(book);
        transaction.commit();
        session.close();
        return 1;
    } catch (DataAccessException exception) {
        exception.printStackTrace();
    }
    return 0;
}

@Override
public boolean deleteBook(long ISBN) {
    // TODO Auto-generated method stub
    Session session = sessionFactory.openSession();
    Transaction transaction = session.beginTransaction();
    try {
        Book book = session.get(Book.class, ISBN);
        session.delete(book);
        transaction.commit();
        session.close();
        return true;
    } catch (DataAccessException exception) {
        exception.printStackTrace();
    }
    return false;
}
```

```

        return true;
    } catch (DataAccessException exception) {
        exception.printStackTrace();
    }
    return false;
}
}

```

- Add connection\_new.xml to configure SessionFactory and other details as shown below:

```

<context:annotation-config />
<context:component-scan base-package="com.packt.ch03.*">
</context:component-scan>

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
    <!--properties for dataSource-->
</bean>

<bean id="sessionFactory" class=
      "org.springframework.orm.hibernate5.LocalSessionFactoryBe
      <property name="dataSource" ref="dataSource" />
      <property name="mappingResources">
        <list>
          <value>book.hbm.xml</value>
        </list>
      </property>
      <property name="hibernateProperties">
        <props>
          <prop key=
                "hibernate.dialect">org.hibernate.dialect.MySQLDial
            </prop>
          <prop key="hibernate.show_sql">true</prop>
          <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
      </property>
    </bean>

```

- Create or copy MainBookDAO\_operations.java to get the bean 'bookDAO\_sessionFactory' to test the application. The code will be:

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context=new

```

```

ClassPathXmlApplicationContext("connection_new.xml");
BookDAO bookDAO=(BookDAO)
    context.getBean("bookDAO_sessionFactory");
//add book
int rows=bookDAO.addBook(new Book("Java EE 7 Developer
Handbook", 97815674L,"PacktPub
publication",332,"explore the Java EE7
programming","Peter pilgrim"));
if(rows>0)
{
    System.out.println("book inserted successfully");
}
else
    System.out.println("SORRY!cannot add book");

//update the book
rows=bookDAO.updateBook(97815674L,432);
if(rows>0)
{
    System.out.println("book updated successfully");
}
else
    System.out.println("SORRY!cannot update book");
//delete the book
boolean deleted=bookDAO.deleteBook(97815674L);
if(deleted)
{
    System.out.println("book deleted successfully");
}
else
    System.out.println("SORRY!cannot delete book");
}

```

We already had seen how to configure the HibernateTemplate in the XML. It extensively works with Transaction but we yet have not discussed anything about what is transaction, it's configuration and how to manage it? We will discuss it after few chapters.

The real time applications handles huge amount of data in each step. Let's say we want to find a book. Using hibernate we will simply invoke a method which returns the book depending upon the ISBN of it. In day today use the book will be searched countless times and each time the database will be hit leading to performance issues. Instead of that it will be great to have a mechanism which will use the outcome of the previous query next time if

someone asks for it again. The Spring 3.1 introduced effective yet simplest way 'a cache' mechanism to achieve it and added JSR-107 annotation support in 4.1. The cached result will be stored in the cache repository and will be used next time to unnecessary hits to the database. You might be thinking of buffer, but it's different from cache. The buffer is an intermediate temporary storage to write and read data once. But cache is hidden to improve the performance of an application and from where the data is read multiple times.

The cache repository is the location where the objects fetched from the database will be saved in key-value pair. Spring supports following the repositories,

### **JDK based ConcurrentMap cache:**

In JDK ConcurrentHashMap is used as backing Cache store. Spring framework has SimpleCacheManager to get CacheManager and giving it a name. This cache is best for relatively small data which doesn't change frequently. But it cannot be used outside of Java heap to store data also there is no built in way to share the data between multiple JVMs.

### **EhCache based cache:**

EhcacheChacheManager is used to get a cache manager where the Ehcache configuration specifications to be configured in the configuration file generally named ehcache.xml. The developers can use different cache manager for different databases with different configurations.

### **Caffeine cache:**

Caffeine is Java8 based caching library to provide high performance. It helps in overcoming the important drawback of ConcurrentHashMap that it persist the data until explicitly removed. Along with it also provides automatic loading of the data, expiration of data based on time, provides notification of the evicted data entries.

Spring provides both XML based as well as annotation based cache configuration. The easiest way is to use annotation based configuration. From Spring 3.1 onward versions have enables JSR-107 support. To take advantage

of the cache using JSR-107 the developers need to first do cache declaration which will help in identifying the methods to be cached and second configuring the cache to inform where the data is stored.

## The cache declaration

The cache declaration can be done using Annotation as well as XML based. Following are the annotations which developers can use for the declaration:

### **@Cacheable:**

The annotation is used to declare that result of these methods is going to be stored in the cache. It takes the name of the cache associated with it. Each time when the developers invoked the methods first off all cache is checked to find out whether the invocation is already done or not.

### **@Caching:**

The annotation is used when more than one annotations as `@CacheEvict`, `@CachePut` need to be nested on the same method.

### **@CacheConfig:**

The annotation `@CacheConfig` is used to annotate the class. In the class whose methods annotated using cache based annotations to specify the cache name each time. If the class has multiple methods annotating it with `@CacheConfig` allows us to specify cache name only once.

### **@CacheEvict:**

It is used to remove the unused data from the cache region.

### **@CachePut**

The annotation is used to update the cache result each time the method

annotated with it is invoked. The annotation behaves exactly opposite to that of `@Cacheable` as it forcefully invokes the method to update the cache and `@Cacheable` skip the execution.

### The cache configuration:

To enable the use of annotation based configuration first off all the Spring has to be registered using cache namespace. The following configuration can be used to declare the namespace for cache and to register the annotation:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:cache="http://www.springframework.org/schema/cache"
       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.xsd
                           http://www.springframework.org/schema/cache
                           http://www.springframework.org/schema/cache/spring-cache.xsd">
    <cache:annotation-driven />
</beans>
```

Once the registration is done now it's time to provide the configuration to specify what is the name of the repository and using which cache manager repository the results will be cached. We will define the configuration very soon in the sample demo for the `SimpleCacheManager`.

Let's integrate JDK based `ConcurrentMap` repository in our Book application. We will use `Ch03_Spring_Hibernate_Integration` as base project for the demonstration. Follow the steps for the integration,

1. Create a new Java application named `Ch03_CacheManager` and add to it jars for Spring, JDBC and hibernate. You can refer to the `Ch03_Spring_Hibernate_Integration` application as well.
2. Create or copy `Book.java` in `com.packt.ch03.beans` package.
3. Create or copy `BookDAO` interface in `com.packt.ch03.dao` package and add to it a method to search the book from database using ISBN. The signature of the method is as shown below:

```
public Book getBook(long ISBN);
```

4. Implement the methods in `BookDAO_SessionFactory_Cache` as we

already did in the BookDAO\_SessionFactory.java from Hibernate application. The method to get the book from database will be as:

```
public Book getBook(long ISBN) {  
    // TODO Auto-generated method stub  
    Session session = sessionFactory.openSession();  
    Transaction transaction = session.beginTransaction();  
    Book book = null;  
    try {  
        book = session.get(Book.class, ISBN);  
        transaction.commit();  
        session.close();  
    } catch (DataAccessException exception) {  
        exception.printStackTrace();  
        book;  
    }  
}
```

The method is going to use 'repo' repository for caching the result.

5. Copy book.hbm.xml in classpath.
6. Add the MainBookDAO\_Cache.java with main function to get the data from the database but purposely we will fetch the data twice as shown below:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    ApplicationContext context=new  
        ClassPathXmlApplicationContext("connection_new.xml");  
    BookDAO bookDAO=(BookDAO)  
        context.getBean("bookDAO_sessionFactory");  
    Book book=bookDAO.getBook(97815674L);  
  
    System.out.println(book.getBookName() +  
        "\t"+book.getAuthor());  
    Book book1=bookDAO.getBook(97815674L);  
    System.out.println(book1.getBookName() +  
        "\t"+book1.getAuthor());  
}
```

7. Before execution make sure the ISBN we are searching is already present in the database. We will get the following output:

```
Hibernate: select book0_.ISBN as ISBN1_0_0_, book0_.book_na  
book0_.book_author as book_aut4_0_0_, book0_.book_price as
```

```

book_pri5_0_0_ from book_hib book0_ where book0_.ISBN=?
book:-Java EE 7 Developer Handbook Peter pilgrim

Hibernate: select book0_.ISBN as ISBN1_0_0_, book0_.book_na-
book0_.book_author as book_aut4_0_0_, book0_.book_price as
book_pri5_0_0_ from book_hib book0_ where book0_.ISBN=?
book1:-Java EE 7 Developer Handbook Peter pilgrim

```

The above output clearly shows the query to search for book is executed twice, denoting the database has been hit twice.

Let's now configure the Cache manager to cache the result of the search book as follow,

1. Annotate the method whose result to be cached by `@Cacheable` as shown below:

```

@Cacheable("repo")
public Book getBook(long ISBN) { // code will go here }

```

2. Configure the cache namespace in the `connection_new.xml` as we already discussed.
3. Register the annotation based cache in the XML as shown below:

```
<cache:annotation-driven />
```

4. Add the CacheManger for setting the repository as 'repo' as shown in the following configuration:

```

<bean id="cacheManager"
      class="org.springframework.cache.support.SimpleCacheManag
      <property name="caches">
          <set>
              <bean class="org.springframework.cache.concurrent.
                  ConcurrentMapCache FactoryBean">
                  <property name="name" value="repo"></property>
              </bean>
          </set>
      </property>
  </bean>

```

5. Execute the `MainBookDAO_Cache.java` without change to get the following output:

```
Hibernate: select book0_.ISBN as ISBN1_0_0_, book0_.book_na  
book0_.book_author as book_aut4_0_0_, book0_.book_price as  
book_pri5_0_0_ from book_hib book0_ where book0_.ISBN=?  
book:-Java EE 7 Developer Handbook Peter pilgrim
```

```
book1:-Java EE 7 Developer Handbook Peter pilgrim
```

The console output shows the query has been fired only once even we searched the book twice. The result of the book fetched first time by `getBook()` is cached and used next time whenever someone ask for the book without heating the database.

# Summary

---

In this chapter we discussed in depth about the persistency layer. The discussion gave us orientation about how to integrate JDBC in application using Spring via DataSource. But using JDBC still exposes the developers to JDBC APIs and its operations like getting Statement, PreparedStatement and ResultSet. But the JdbcTemplate and JdbcDaoSupport provides a means to perform database operations without getting involved in JDBC APIs. We also have seen the exception hierarchy given by Spring which can be used according to the situation in the application. We also discuss about Hibernate as ORM tool and its integration in the framework. Cache helps in minimum hits to the database and enhancing the performance. We discuss about cache mangers and how to integrate the CacheManger in the application.

In the next chapter we will discuss about Aspect Oriented Programming which helps in handling cross cutting technologies.

# Chapter 4. Aspect Oriented Programming

The previous chapter on Spring DAO gives good hands on practice about how Spring handles JDBC API with loose coupling. But, we neither talked about JDBC Transactions nor about how Spring handles Transactions. If you already handled transactions you know the steps for it and more over, you are well aware of these steps which are repetitive and spread all over the code. On one hand, we are saying use Spring to stop duplication of code and on other hand we are writing such code. Java insist to write modules which are highly cohesive. But writing transaction management in our code won't allow us to write cohesive modules. Also the transaction is not the motive of writing the code. It just provides support so that the business logic of application will not carry out any undesired effect. We haven't discussed about how to handle such supportive functionalities along with the main motive of application development. Apart from transaction what else functionalities does the work of providing support to application? This chapter will help us in writing the highly cohesive modules without repetition of the code to handle such supportive functionalities. In this chapter we will discuss the following points:

- What are cross cutting technologies?
- What role do the cross cutting technologies play in application development?
- We will discuss about AOP and how AOP plays an important role in handling cross cutting technologies.
- We will explore in depth what are Aspects, Advices, PointCut in AOP.

The software application provides a reliable solution to a client's problem. Though we say reliable, always there is a chance of getting some runtime problems. So along with development, the maintenance of the software is also equally important. Whenever a problem occurs in application the client gets back to the developers for the solution. Unless and until the client is not able to state the accurate reason of the problem the developers are helpless. The developers have to recreate the same situation in order to prevent it's next occurrence. In enterprise application, due to huge number of modules the

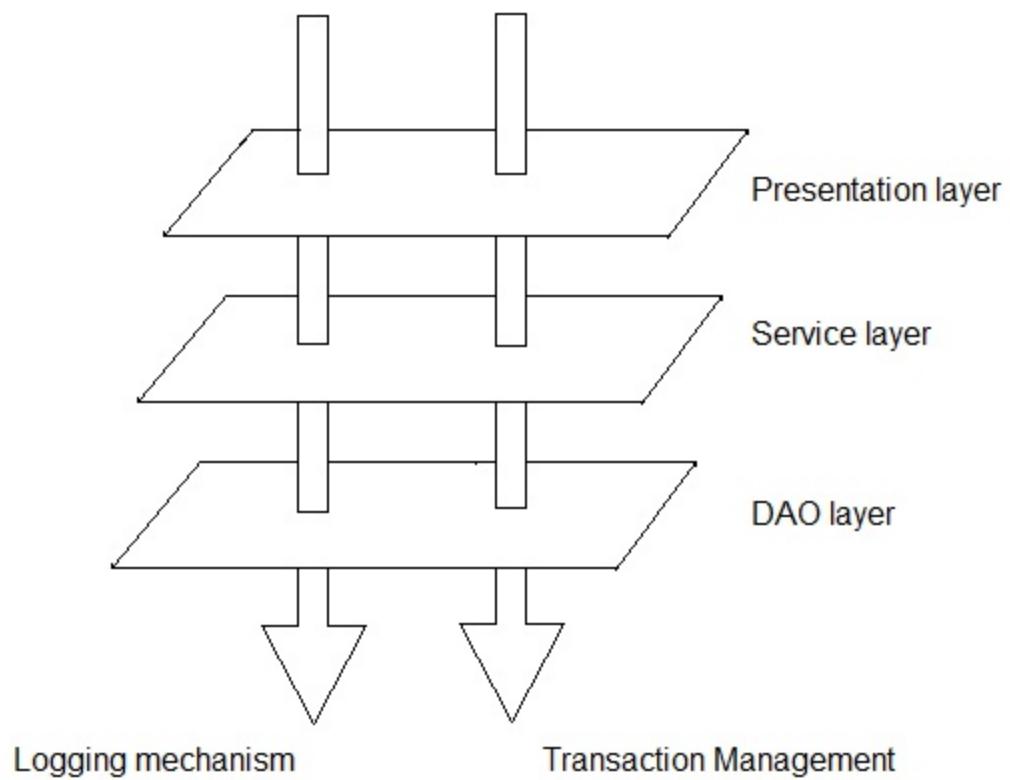
recreation of the same problem becomes complex. It will be great to have someone who keeps on tracking what the user is doing. The trace of this tracker helps the developers to know why and what went wrong or even using the trace so that they can recreate it easily. Yes, I am talking about logging mechanism.

Let's take another very common situation of Railway ticket reservation. At the time of ticket reservation we select the available seat from the chart and proceed for money transfer. Sometimes money gets transferred successfully as well as the ticket gets booked. But unfortunately, sometime due to the time for money transaction, delay in form filling or some server side issues may cause problems to transfer of money without booking the ticket. The money gets deducted without issue of the ticket. The customer will be unhappy and moreover tense for the refund. Such situations need to be handled carefully with the help of transaction management so that if the ticket is not issued the money should get deposited back into the customer's account. Doing it manually will be tedious task instead transaction management handles it elegantly.

We can write the workable code without logging or transaction management, as both of these are not part of your business logic. The Java application revolves around providing a customized, easy solution for an enterprise problem. The business logic is at the centre to provide primary functionality of the application may sometime referred as 'primary concern'. But it also has to be supported for some other functionalities or services and it can't be neglected. These services play vital role in the application. Either the migration of application becomes time consuming or backtracking the problem occurred at run time will be difficult. These concerns are scattered throughout the application mostly with repetitive code. Such secondary concerns are called as 'cross cutting concerns' or sometimes also called as 'horizontal concerns'. Logging, transaction management, security mechanism are some of the cross cutting concerns which developers use in the application.

The below diagram shows the way cross cutting concerns as logging and transaction management are scattered in the application code:

### Cross Cutting Concern



# Aspect Oriented Programming (AOP)

---

Similar to Object Oriented Programming, the Aspect Oriented Programming is also a style of programming which allows developers to write cohesive code by separating cross cutting concern from the business logic code. The AOP concepts have been developed by Gregor Kiczale and his colleagues. It provides different ways or tools to write cross cutting concerns.

AOP to handle Cross cutting concern is written at one place which helps in achieving the following benefits,

- Reduction in the duplication of the code to achieve writing clean code.
- Helps in writing the loosely coupled modules.
- Helps in achieving highly cohesive modules.
- The developer can concentrate on writing the business logic
- Easy to change or modify the code for new functionalities without touching the exiting code.

To understand AOP we must be aware of the following common terminologies without which we cannot imagine AOP.

## Join Point

The join points are the points in the application where an aspect can be plugged in for some miscellaneous functionality without being the part of the actual business logic. Each code has numerous opportunities, which can be treated as join point. The class which is the smallest unit in an application has data members, constructors, setters and getters, other functional classes. Each of them can be an opportunity where the aspect can be applied. Spring supports only methods as join points.

## Pointcut

The join points are the opportunities, but all of them are not considered where the aspects can be applied. A pointcut is where the developers decide to apply the aspect to perform a specific action for the cross cutting concern. The pointcut will be written using the method names, class names, regular expressions to define the matching packages, classes, methods where aspects can be applied.

## Advice

The action taken by the aspect at pointcuts is called as 'advice'. The advice has the code which gets executed for the respective cross cutting concern. If we consider the method as the join point, the aspect can be applied before or after the method gets executed. It is also possible that method has exception handling code where the aspect can be plugged. Following are the available advices in the Spring framework.

### Before

The Before advice contains the implementation which will be applied before the business logic method which matches the pointcut expression. It will continue with the execution of the method unless the exception is not thrown. The `@Before` annotation or `<aop:before>` configuration can be applied to a method to support Before advice.

### After

In After advice the implementation will be applied after the business logic method irrespective of whether the method executed successfully or exception is thrown. The `@After` annotation or `<aop:after>` configuration can be used to support Before advice by applying it to a method.

### After returning

The After Returning advice has the implementation which is applied only after the successful execution of the business logic method takes place. The `@AfterReturning` annotation or `<aop:after-returning>` configuration can be

applied to a method to support after returning advice. The after returning advice method can use the value returned by the business logic method.

## **After throwing**

The After Throwng advice has the implementation which is applied after the execution of the business logic method which has thrown an exception. The `@AfterThrowing` annotation or`<aop:throwing>` configuration can be used to support After throwing advice by applying it to a method.

## **Around**

The Around advice is the most important among all the advices and the only one which is applied on a method both before and after the execution of the business logic method. It can be used to choose whether to proceed for the next join point or not using the invocation of `proceed()` method of `ProceedingJoinPoint`. The `proceed()` helps in choosing whether to proceed to the join point or not by returning its own returned value. It can be used in scenarios where developers needs to perform pre processing, post processing or both. The calculation of how much time taken by the method for its execution is one of such scenario. The `@Around` annotation or`<aop:around>` configuration can be used to support around advice by applying it to a method.

## **Aspect**

An aspect defines the opportunities by the pointcuts expressions and the advices to specify when and where the action will be taken. `@Aspect` annotation or`<aop:aspect>` configuration is applied to a class to declare it as an aspect.

## **Introduction**

The Introduction helps in the declaration of additional methods, fields in the existing class without changing the existing code. Spring AOP allows

developers to introduce a new interface to any class which has been advised by the aspect.

## Target object

The target objects are the objects of the classes on whom the aspects are applied. Spring AOP creates proxy of target object at runtime. The method from the class is overridden and the advice will be included to it to get the desired result.

## AOP proxy

By default the Spring AOP uses JDK's dynamic proxy to get the proxy of the target classes. The use of CGLIB for the proxy creation is also very common. The target object is always proxied using Spring AOP proxy mechanism.

## Weaving

We as developers write business logic and the aspect code in two separate modules. Then these two has to be combined as proxied target class. The process of plugging the aspect in the business logic code is known as 'weaving'. The weaving can happen at compile time, load time or at runtime. Spring AOP does weaving at runtime.

Let's take a very simple example to understand the discussed terminologies. My son loves watching drama. So we had gone to watch one. As we all aware we cannot enter unless and until we have entry passes or tickets. Obviously we need to collect them first. Once we have the ticket my son dragged me to the seat and showed me the seat in excitement. The show started. It was a funny drama for kids. All kids were laughing on the jokes, clapping on dialogues, getting excited on the dramatic scenes. At interval most of the audience went to take pop corn, snacks and cold drinks. Everyone enjoyed the drama and left happily from the exit. Now, we might be thinking that we all know this. Why we are discussing this and what is its relation to the aspect. Are we not going off the way from the discussion? No, we are on the

right track. Just wait for a while and you all also will be agreed. Here watching the drama was our main task, let's say it's our business logic or core concern. Purchasing the tickets, paying the money, entering in the theatre, leaving it once the drama is over are the functionalities are the part of the core concern. But we cannot just sit quietly, we react on what is going on? We clap, laugh and sometimes even cry. But are these main concerns? No! But without them we cannot imagine audience watching drama. These will be supportive functionalities which each audience does spontaneously. Correct !!! These are the cross cutting concerns. The audience won't get instructions for cross cutting concerns individually. These reactions are the part of aspects having advices. Some will clap before the drama and few after the drama and the most excited whenever they feel. These are nothing but before, after or around advices of the aspect. If the audience doesn't enjoy the drama they may leave in between similar to after throwing exception. On very unfortunate day, the show may get cancelled or even stopped in between which needs to be introduced by the organizers as an emergency. Hope you now know the concepts as well as their practical approach. We will cover this and many more in the demo shortly.

Before moving on with the demonstration, let's first off all discuss about some of the AOP frameworks in the market as follows.

## **AspectJ**

AspectJ is the easy to use and to learn Java compatible framework for integrating cross cutting implementations. The AspectJ has been developed at PARC. Now a day, it is one of the famous AOP framework due to its simplicity yet has power to support component modularization. It can be used to apply AOP on fields which are static or non static, constructors, methods which are private, public or protected.

## **AspectWertz**

AspectWertz is another Java compatible light weight powerful framework. It can be used easily to integrate in new as well as existing application. The AspectWertz supports both by XML as well as annotation based aspect writing and configuration. It supports compile time, load time and runtime

weaving. Since AspectJ5, it has been merged in AspectJ.

## **JBoss AOP**

The JBoss AOP supports writing of aspects and dynamic proxy target objects. It can be used to apply AOP on fields which are static or non static, constructors, methods which are private, public or protected using interceptors.

## **Dynaop**

The Dynaop framework is proxy based AOP framework. The framework helps in reducing the dependencies and code reusability.

## **CAESAR**

CASER is Java compatible AOP framework. It supports implementation of abstract component as well as their integration.

## **Spring AOP**

It is Java compatible easy to use framework which is used to integrate AOP in Spring framework. It provides a close integration of AOP implementation in components taking advantages of Spring IoC. It is proxy based framework which can be used on method execution.

The Spring AOP fulfills maximum requirements for applying cross cutting concerns. But following are few limitations where Spring AOP cannot be applied,

- Spring AOP cannot be applied on fields
- We cannot apply any other Aspect on one aspect
- Private and protected methods can't be advised
- Constructors cannot be advised

Spring supports AspectJ and Spring AOP integration to use cross cutting

concerns with less coding. Both Spring AOP and AspectJ are used for implementation of cross cutting technology but following are few points which helps the developers to make the best choice to be used in implementation:

- Spring AOP is based on dynamic proxy which supports only method join points but AspectJ can be applied on fields, constructors even they are private, public or protected supporting a fine grained advise.
- Spring AOP cannot be used on method which calls methods of the same class or which is static or final but AspectJ can.
- AspectJ doesn't need Spring container to manage component while Spring AOP can be used only with the components which are managed by Spring container.
- Spring AOP supports runtime weaving based on proxy pattern and AspectJ supports compile time weaving which does not required proxy creation. The proxy of the objects will be created once the bean is asked by the application.
- Aspects written by Spring AOP are Java based components but those written in AspectJ are with language which is extension of Java, so the developers need to learn it before use.
- Spring AOP is easy to implement by annotating a class with `@Aspect` annotation or by simple configuration. But to use AspectJ one need to create `*.aj` files.
- Spring AOP doesn't required any special container but aspects needs as aspects created using AspectJ needs to compile using AspectJ compiler.
- AspectJ is a best choice for the applications which already exists.

## Note

A simple class without final, static methods simply use Spring AOP otherwise choose AspectJ to write the aspects.

Let's discuss in depth about Spring AOP and its ways of implementations. The Spring AOP can be implemented using either XML based aspect configuration or AspectJ style annotation based implementation. The XML based configuration can be split up at several point making it bit complex. In XML we cannot define named pointcuts. But the aspect written by

annotations is within the single module which supports writing named pointcuts. So, without wasting time let's start the XML based aspect development.

## XML based aspect configuration

Following are the steps need to be followed for developing XML based aspect,

1. Select the cross cutting concern to be implemented
2. Write the aspect to fulfill the requirement of cross cutting concern.
3. Register the aspect as a bean in Spring context.
4. Write the aspect configuration as:
  - \* Add AOP namespace in XML.
  - \* Add aspect configuration which will have pointcut expressions and advises.
  - \* Register the bean on whom the aspect can be applied.

From the available join points the developers need to decide which to track and then need to write pointcut using expression to target them. To write such pointcuts the Spring framework uses AspectJ's pointcut expression language. We can write point cuts with the help of following designators in the expression.

### Using method signatures

The method signature can be used to define the pointcuts from the available join points. The expression can be written using the following syntax:

```
expression(<scope_of_method>      <return_type><fully_qualified_nam
```

Java supports private, public, protected and default as method scope but Spring AOP supports only public methods while writing the pointcut expressions. The parameter list is use to specify what data types will be considered when the method signature is matched. Two dots(..) can be used

by the developers, if they don't want to specify either number of arguments or their data types.

Let's consider the following expressions to understand writing of expressions in depth to decide which join points will be advised:

- `expression(* com.packt.ch04.MyClass.*(..))` - specifies all the methods with any signature from MyClass within com.packt.ch03 package.
- `expression(public int com.packt.ch04.MyClass.*(..))` - specifies all the methods returning integer value from MyClass within com.packt.ch03 package.
- `expression(public int com.packt.ch04.MyClass.*(int,..))` - specifies all the methods returning integer and its first argument of integer type from MyClass within com.packt.ch03 package.
- `expression(* MyClass.*(..))` - specifies all the methods with any signature from MyClass will be advised. It's a very special kind of expression which can be used only if the advise and the class belongs to the same package.

## Using type

The type signature is used to match the join point having the specified types. We can use the following syntax to specify type:

`within(type_to_specify)`

Here the type will be either the package or class name. Following are some of the expressions which can be written to specify the join points:

- `within(com.packt.ch04.*)` - specifies all the methods from all the classes belonging to com.packt.ch04 package
- `within(com.packt.ch04..*)` - specifies all the methods from all the classes belonging to com.packt.ch04 package and its sub packages. We specified two dots instead of one to track the sub packages as well.
- `within(com.packt.ch04.MyClass)` - specifies all the methods from the MyClass belonging to com.packt.ch04 package
- `within(MyInterface+)` - specifies all the methods from all the classes

which are implementing MyInterface.

## Using Bean name

Spring 2.5 onwards all versions supports use of bean name to be used in expression to match the join point. We can use the following syntax:

```
bean(name_of.Bean)
```

Consider the following example:

bean(\*Component) - the expression specifies the join points to be match which belongs to the bean whose name ends with Component. The expression can't be used with AspectJ annotations.

## Using this

'this' is used to match the join points where the bean reference is instance of the specified type. It is used when the expression specifies name of class instead of interface. It used when Spring AOP uses CGLIB for proxy creation.

## 5.sing target

The target is used to match the join points where the target object is an interface of the specified type. It is used when Spring AOP uses JDK based proxy creation. The target is used only if the target object is implementing an interface. The developers even can configure property 'proxy target class' set to true.

Let's consider the following example to understand use of this and target in expression:

```
package com.packt.ch04;
Class MyClass implements MyInterface{
    // method declaration
}
```

We can write the expression to target the methods as:

```
target( com.packt.ch04.MyInterface) or  
this(com.packt.ch04.MyClass)
```

## For annotation tracking

The developers can write pointcut expressions which are not tracking the methods but to track the annotations applied. Let's take following examples to understand how to monitor annotations.

### Using with execution:

`execution(@com.packt.ch03.MyAnnotation)` - specifies to target the method or class which has been annotated with MyAnnotation.

`execution(@org.springframework.transaction.annotation.Transactional)` - specifies to target the method or class which has been annotated with Transactional.

### Using with @target:

It is used to consider the join points where the class has been annotated with specified annotation. The following example makes it clear,

`@target(com.packt.ch03.MyService)` - used to consider the join point which has been annotated by MyService annotation.

### Using @args:

The expression is used to specify the join points where the arguments have been annotated with the given type.

`@args(com.packt.ch04.annotations.MyAnnotation)`

The above expression is used to consider the join points whose accepts objects annotated by `@Myannotation`.

## **Using @within:**

The expression is used to specify the join points within types which have been specified by the given annotation.

`@within(org.springframework.stereotype.Repository)`

The above expression helps in providing advise to the join points which has been annotated by `@Repository`.

## **Using @annotation:**

`@annotation` is used to match the join points which have been annotated by the respective annotation.

`@annotation(com.packt.ch04.annotations.Annotation1)`

The expression matches all the join points annotated by `Annotation1`.

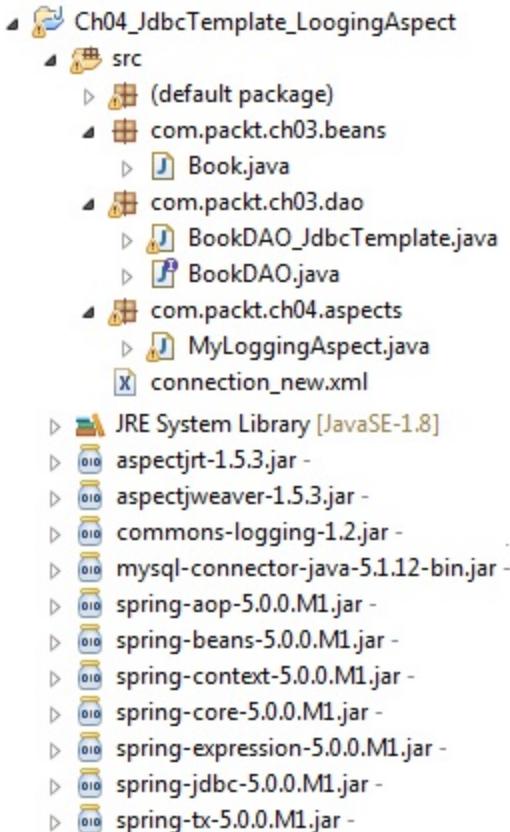
Let's use pointcut expressions, advises to implement logging aspect to understand the real time implementation. We will use application `Ch03_JdbcTemplates` developed in previous chapter to integrate Log4j in it as a base. In first part we will create a copy of the main application and part two integrate it with log4J and in third part we will apply the customized logging aspect.

# Part I : Creating application for the core concern(JDBC)

---

Follow the steps to create the base application:

1. Create a Java application Ch04\_JdbcTemplate\_LoggingAspect and add to it jars for Spring core, Spring JDBC, spring-aop, aspectjrt-1.5.3 and aspectjweaver-1.5.3.jar files.
2. Copy the required source code files and configuration files in respective packages. The final outline of the application is as shown below:



3. Copy connection\_new.xml from Ch03\_JdbcTemplate in classpath of the application and edit it to remove bean with id as 'namedTemplate'.

# PartII: Integration of Log4J

---

Log4j is the most simple thing to do. Let's use following steps for the integration:

1. To integrate Log4J we first of all have to add log4j-1.2.9.jar in the application.
2. Add log4j.xml in classpath with the following configuration to add Console and File appenders:

```
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration
    xmlns:log4j='http://jakarta.apache.org/log4j/'>
    <appender name="CA"
        class="org.apache.log4j.ConsoleAppender">
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-4r [%t]
                %-5p %c %x - %m%n" />
        </layout>
    </appender>
    <appender name="file"
        class="org.apache.log4j.RollingFileAppender">
        <param name="File" value="C:\\log\\log.txt" />
        <param name="Append" value="true" />
        <param name="MaxFileSize" value="3000KB" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{DATE}
                %-5p %-15c{1}: %m%n" />
        </layout>
    </appender>
    <root>
        <priority value="INFO" />
        <appender-ref ref="CA" />
        <appender-ref ref="file" />
    </root>
</log4j:configuration>
```

You can modify the configuration as per your requirement.

1. Now to log the messages we will add the code for getting logger and

then logging mechanism. We can add the code to BookDAO\_JdbcTemplate.java as shown below:

```
public class BookDAO_JdbcTemplate implements BookDAO {  
    Logger logger=Logger.getLogger(BookDAO_JdbcTemplate.class)  
    public int addBook(Book book) {  
        // TODO Auto-generated method stub  
        int rows = 0;  
        String INSERT_BOOK = "insert into book  
            values(?, ?, ?, ?, ?, ?);  
        logger.info("adding the book in table");  
  
        rows=jdbcTemplate.update(INSERT_BOOK, book.getBookName(  
            book.getISBN(), book.getPublication(),  
            book.getPrice(),  
            book.getDescription(), book.getAuthor()));  
  
        logger.info("book added in the table successfully"+  
            rows+"affected");  
        return rows;  
    }  
}
```

Don't worry we will not add it in our application in each class and then in methods as we already discuss the complexity and repetitive code let's move on to write aspect for logging mechanism with the help of following steps to get the same result as that of the code written above.

# Part III: Writing Logging aspect.

---

1. Create MyLoggingAspect as Java class in com.packt.ch04.aspects package which will have method for before advise.
2. Add a data member of type org.apache.log4j.Logger in it.
3. Add a method beforeAdvise( ) in it. The signature of method can be anything, Here we are adding JoinPoint as argument. Using this argument we can get the information about the class where aspect is getting applied. The code will be as follows:

```
public class MyLoggingAspect {  
    Logger logger=Logger.getLogger(getClass());  
    public void beforeAdvise(JoinPoint joinPoint) {  
        logger.info("method will be invoked :-  
        "+joinPoint.getSignature());  
    }  
}
```

4. The aspect now has to be configured in the XML in three steps:

\*Add namespace for AOP:

```
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:aop="http://www.springframework.org/schema/aop"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/b  
                           http://www.springframework.org/schema/beans/spring-beans.  
                           http://www.springframework.org/schema/aop  
                           http://www.springframework.org/schema/aop/spring-aop.xsd"
```

5. Now we can use AOP's tags by using 'aop' namespace:

\*Add a bean for aspect.

6. Add the bean for aspect which we want to use in application in connection\_new.xml as shown below:

```
<bean id="myLogger"
```

```
class="com.packt.ch04.aspects.MyLoggingAspect" />
```

### \*Configure the Aspect.

7. Each <aop:aspect> enables us writing aspect inside <aop:config> tag.
8. Each aspect will have id and ref as attributes. The 'ref' which refers to the bean whose methods will be invoked for providing advises.
9. Configure the advise for a pointcut expression and which method to be invoked for the advise. The before advise can be configured inside<aop:aspect> using <aop:before>tag.
10. Let's write before advise to apply for 'myLogger' aspect which will be invoked before the addBook() method of BookDAO. The configuration will be as follows:

```
<aop:config>
  <aop:aspect id="myLogger" ref="logging">
    <aop:pointcut id="pointcut1"
      expression="execution(com.packt.ch03.dao.BookDAO.addB
      (com.packt.ch03.beans.Book))" />
    <aop:before pointcut-ref="pointcut1"
      method="beforeAdvise"/>
  </aop:aspect>
</aop:config>
```

11. Execute MainBookDAO\_operation.java to get the following output on console:

```
0 [main] INFO org.springframework.context.support.Cla
66 [main] INFO org.springframework.beans.factory.xml.
842 [main] INFO org.springframework.jdbc.datasource.D
931 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - me
book inserted successfully
book updated successfully
book deleted successfully
```

Here BookDAO\_JdbcTemplate is working as target object whose proxy will be created at runtime by weaving the code of addBook() and beforeAdvise() methods. Now once we know the process let's add different pointcuts and the advises one by one in the application with the help of following steps.

## Note

More than one advises can be applied on the same joinpoint but for simplicity to understand the pointcuts and advises, we will keep single advise each time and comment already written.

## **Adding after advise.**

Let's add after advise for all the methods from BookDAO.

1. Add a method afterAdvise() in MyLoggingAspect for after advise as shown below:

```
public void afterAdvise(JoinPoint joinPoint) {  
    logger.info("executed successfully :-  
    "+joinPoint.getSignature());  
}
```

2. Configure the pointcut expression to target all the methods inside BookDAO class and after advise in the connection\_new.xml inside 'myLogger' aspect as shown below:

```
<aop:pointcut id="pointcut2"  
    expression="execution(com.packt.ch03.dao.BookDAO.*(..))"  
<aop:after pointcut-ref="pointcut2" method="afterAdvise"/>
```

3. Execute MainBookDAO\_operations.java to get following output:

```
999 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - method w  
1360 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - execute  
book inserted successfully  
1418 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - execute  
book updated successfully  
1466 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - execute  
book deleted successfully
```

The underlined statements makes it clear that the advise got invoked after all the methods.

## **Adding after returning advise.**

Though we written the after advise but we are not able to get the value

returned from the business logic method. After-returning will help us to get the returned value with the help of following steps.

1. Add a method `returnAdvise()` in `MyLoggingAspect` which will get invoked for after returning. The code is as shown below:

```
public void returnAdvise(JoinPoint joinPoint, Object val) {  
    logger.info(joinPoint.getSignature() + " returning val" +  
}
```

The argument '`val`' will hold the returned value.

2. Configure the advise under '`myLogger`'. We don't have to configure the pointcut as we will be reusing already configured. In case if you want to use different set of join point, first you need to configure a different pointcut expression. Our configuration will be as shown below:

```
<aop:after-returning pointcut-ref="pointcut2"  
    returning="val" method="returnAdvise" />
```

where,

\* `returning` - represents the attribute to specify the name of the parameter to which the return value will be passed. In our case this name is '`val`' which has bound in advice arguments .

3. To make the output easy to understand comment before and after advise configuration and then execute `MainBookDAO_operations.java` to get following lines on console output:

```
1378 [main] INFO com.packt.ch04.aspects.MyLoggingAspect -  
    returning val:-1  
1426 [main] INFO com.packt.ch04.aspects.MyLoggingAspect -  
1475 [main] INFO com.packt.ch04.aspects.MyLoggingAspect -  
    boolean com.packt.ch03.dao.BookDAO.deleteBook(long)  
    returning val:-true
```

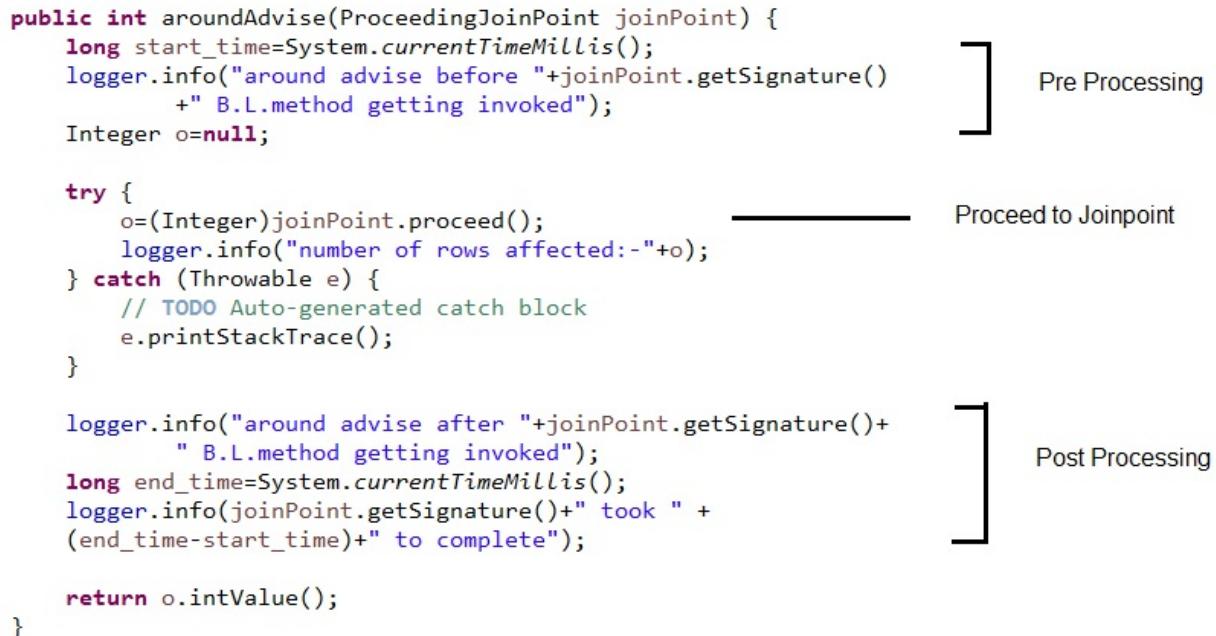
Each of the statements shows the value returned from the joinpoint.

## **Adding Around advise.**

As we already discuss around advise is invoked both before and after the business logic method, only if the execution is successful. Let's add around advise in application as:

1. Add a method aroundAdvise () in MyLoggingAspect. This method must have one of its argument as ProceedingJoinPoint to facilitate the flow of application to join point. The code will be as follows:

```
public int aroundAdvise(ProceedingJoinPoint joinPoint) {  
    long start_time=System.currentTimeMillis();  
    logger.info("around advise before "+joinPoint.getSignature()  
        +" B.L.method getting invoked");  
    Integer o=null;  
  
    try {  
        o=(Integer)joinPoint.proceed();  
        logger.info("number of rows affected:-"+o);  
    } catch (Throwable e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
  
    logger.info("around advise after "+joinPoint.getSignature()  
        +" B.L.method getting invoked");  
    long end_time=System.currentTimeMillis();  
    logger.info(joinPoint.getSignature()+" took " +  
    (end_time-start_time)+" to complete");  
  
    return o.intValue();  
}
```



The part before proceed() will get invoked before B.L. method which we are referring as 'Pre processing'. The ProceedJoinPoint's proceed() method take the flow to the respective join point. If the join point executes successfully the part after proceed will be executed which we are referring as 'Post processing'. Here we are finding time taken to complete the process by taking the difference of time at pre processing and post processing.

The join point where we want to weave the aspect returns int so the aroundAdvise() method is also returning value of the same type. If in case we add void instead of int we will get the following exception:

```
Exception in thread "main"  
org.springframework.aop.AopInvocationException: Null return  
abstract int  
com.packt.ch03.dao.BookDAO.addBook(com.packt.ch03.beans.Boo
```

2. Let's now add around advise in 'myLogger' as shown below,

```
<aop:around pointcut-ref="pointcut1" method="aroundAdvise"
```

3. Execute the MainBookDAO to the following log on console while commenting the advises configured previously,

```
1016 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - a  
B.L.method getting invoked  
1402 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - n  
1402 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - a  
advise after int com.packt.ch03.dao.BookDAO.addBook(Book)  
B.L.method getting invoked  
1403 [main] INFO com.packt.ch04.aspects.MyLoggingAspect - i  
com.packt.ch03.dao.BookDAO.addBook(Book) took 388 to comple
```

## Adding after throwing advise

As we know the after throwing advise will be triggered once the matching join point will throw an exception. While performing JDBC operation if we try to add the duplicate entry in the book table the DuplicateKeyException will be thrown we just want to log it with the help of after throwing advise with the help of following steps,

1. Add the method throwingAdvise() in MyLoggingAspect as follows:

```
public void throwingAdvise(JoinPoint joinPoint,  
                           Exception exception)  
{  
    logger.info(joinPoint.getTarget().getClass().getName() +"  
               got and exception" + "\t" + exception.toString());  
}
```

The developers are free to choose the signature but as the join point method is going to throw exception the method written for the advise will have one of its argument of type Exception so that we can log it. We also are adding argument of type JoinPoint as we want to deal with method signatures

2. Add the configuration in the connection\_new.xml in 'myLogger' configuration. The configuration to add is:

```
<aop:after-throwing pointcut-ref="pointcut1"
method="throwingAdvise" throwing="exception" />
```

The <aop:after- throwing> will take:

- \* **pointcut-ref** - name of pointcut-ref where we want to weave the joinpoints
- \* **method** - name of method which will invoke if the exception is thrown
- \* **throwing** - name of the argument to be bound from the advise method signature to which the exception will be passed. The name of argument in the method signature used by us is 'exception'.

3. Execute the MainBookDAO\_operations and purposely add the book whose ISBN already exists in the Book table. Before execution comment the previous configurations added for other advises. We will get the following output:

```
1322 [main] ERROR com.packt.ch04.aspects.MyLoggingAspect -
com.packt.ch03.dao.BookDAO.addBook(Book) got and exception
org.springframework.dao.DuplicateKeyException:
PreparedStatementCallback; SQL [insert into book
values(?, ?, ?, ?, ?, ?)]; Duplicate entry '9781235' for key 1;
exception is
com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry '9781235' for key 1
```

4. If you add book with different ISBN which is not already in book table the above log for ERROR will not be displayed as no exception no advise will be triggered.

The above demonstration gives clear picture about how the aspect will be written and configured using XML. Let's move on to writing annotation based aspect.

# Annotation based aspect.

---

The aspect can be declared as a Java class annotated with AspectJ annotations to support writing of pointcuts and advises. Spring AspectJ OP implementations provide following annotations for writing aspect:

- **@Aspect** - used to declare a Java class as an Aspect.
- **@Pointcut** - used to declare point cut expression using AspectJ expression language.
- **@Before** - used to declare the before advise which is applied before the business logic (B.L.) method. @Before supports following attribute,
  - **value** - name of the method annotated by @Pointcut
  - **argNames** - to specify the name of parameters at join point
- **@After** - used to declare the after advise which is applied after the B.L. method before returning the result. @After also support same attributes as that of @Before advise.
- **@ AfterThrowing** - used to declare the after throwing advise which is applied after the exception is thrown by the B.L. method. @ AfterThrowing supports following attribute:
  - **pointcut**- the pointcut expression to select the join point
  - **throwing**- the name of the argument which will be bound with exception thrown by B.L. method.
- **@AfterReturning** - used to declare the after returning advise which is applied after the B.L. method but before the result is returned. The advise helps in getting the value of the result from B.L method.  
@AfterReturning supports following attribute,
  - **pointcut**- the pointcut expression to select the join point
  - **returning**- the name of the argument bounded with the value returned from B.L. method.
- **@Around** - used to declare the around advise which is applied before as well as after the B.L. method. @Around support same attribute as that of @Before or @After advise.

We must declare the configuration in the Spring context to enable proxy

creation of the bean. The `AnnotationAwareAspectJAutoProxyCreator` class helps in this. We can register the class in simple way for `@AspectJ` support by including the following configuration in the XML file:

```
<aop:aspectj-autoproxy/>
```

Adding the namespace 'aop' in XML which already had discussed.

We can follow the following steps to declare and use Annotation based aspect:

1. Declare a java class and annotate it by `@Aspect`.
2. Add the method annotated by `@Pointcut` to declare pointcut expression.
3. Add the methods for advises and annotate them by `@Before`, `@After`, `@Around` etc as per the requirements.
4. Add the configuration for namespace 'aop'.
5. Add the aspect in the configuration as a bean.
6. Enable the auto proxy support in the configuration.

Let's add the annotation based aspect in the `JdbcTemplate` application. Follow steps of part I and II to create base application named `Ch04_JdbcTemplateLoggingAspect_Annotation`. You can refer to the `Ch04_JdbcTemplateLoggingAspect` application. Now use the following steps to develop annotation based logging aspect:

1. Create class `MyLoggingAspect` in `com.packt.ch04.aspects` package.
2. Annotate it with `@Aspect`.
3. Add a data member of type `org.apache.log4j.Logger` in it.
4. Add the method `beforeAdvise()` for applying advise before the business logic method `addBook()`. Annotate it with `@Before`. The code will be as shown below:

```
@Aspect
public class MyLoggingAspect {
    Logger logger=Logger.getLogger(getClass());
    @Before("execution(*
        com.packt.ch03.dao.BookDAO.addBook(
        com.packt.ch03.beans.Book) ")
    public void beforeAdvise(JoinPoint joinPoint) {
```

```

        logger.info("method will be invoked :-  

        "+joinPoint.getSignature());
    }
}

```

5. Edit connection\_new.xml to add 'aop' namespace if you already have not done that.
6. Add bean for MyLoggingAspect as shown below:

```
<bean id="logging"  

      class="com.packt.ch04.aspects.MyLoggingAspect" />
```

Alternative to the above configuration will annotating the MyLoggingAspect by @Component annotation.

1. Unable the AspectJ autoproxy by adding the configuration in connection\_new.xml as:

```
<aop:aspectj-autoproxy/>
```

2. Execute the MainBookDAO-operation.java to get the log on console as:

```
23742 [main] INFO com.packt.ch04.aspects.MyLoggingAspect  

method will be invoked :-int  

com.packt.ch03.dao.BookDAO.addBook(Book)
```

To write the pointcut expression for each advise may be a tedious and unnecessarily repetitive task. We can declare the pointcut seperately in a marker method as shown below:

```
@Pointcut(value="execution(*  

com.packt.ch03.dao.BookDAO.addBook(com.packt.ch03.beans.Boo  

public void selectAdd() {}
```

And then refer the above from the advise method. We can update the beforeAdvise () method as:

```
@Before("selectAdd()")
public void beforeAdvise(JoinPoint joinPoint) {
    logger.info("method will be invoked :-  

    "+joinPoint.getSignature());
}
```

3. Once we know the basis of Aspect declaration let's now add the methods for other aspect and pointcut as already discussed in aspect declaration using XML. The aspect will be as shown below:

```
@Aspect
public class MyLoggingAspect {

    Logger logger=Logger.getLogger(getClass());
    @Pointcut(value="execution(*com.packt.ch03.dao.BookDAO.ad
com.packt.ch03.beans.Book))")
    public void selectAdd(){      }

    @Pointcut(value="execution(*
        com.packt.ch03.dao.BookDAO.*(..))")

    public void selectAll(){      }

    // old configuration
    /*
    @Before("execution(*
        com.packt.ch03.dao.BookDAO.addBook(
            com.packt.ch03.beans.Book))")
    public void beforeAdvise(JoinPoint joinPoint) {
        logger.info("method will be invoked :-
                    "+joinPoint.getSignature());
    }
    */
    @Before("selectAdd()")
    public void beforeAdvise(JoinPoint joinPoint) {
        logger.info("method will be invoked :-
                    "+joinPoint.getSignature());
    }
    @After("selectAll()")
    public void afterAdvise(JoinPoint joinPoint) {
        logger.info("executed successfully :-
                    "+joinPoint.getSignature());
    }
    @AfterThrowing(pointcut="execution(*
        com.packt.ch03.dao.BookDAO.addBook(
            com.packt.ch03.beans.Book)),
        throwing="exception")
    public void throwingAdvise(JoinPoint joinPoint,
        Exception exception)
    {
        logger.error(joinPoint.getSignature()+" got and excepti
            + "\t" + exception.toString());
    }
}
```

```

@Around("selectAdd()")
public int aroundAdvise(ProceedingJoinPoint joinPoint) {
    long start_time=System.currentTimeMillis();
    logger.info("around advise before
"+joinPoint.getSignature()
+" B.L.method getting invoked");
Integer o=null;
try {
    o=(Integer)joinPoint.proceed();
    logger.info("number of rows affected:-"+o);
} catch (Throwable e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
logger.info("around advise after
"+joinPoint.getSignature()+
" B.L.method getting invoked");
long end_time=System.currentTimeMillis();
logger.info(joinPoint.getSignature()+" took " +
(end_time-start_time)+" to complete");
return o.intValue(); }

@AfterReturning(pointcut="selectAll()", returning="val")
public void returnAdvise(JoinPoint joinPoint, Object val)
    logger.info(joinPoint.getSignature()+
    " returning val:-" + val);
}
}

```

#### 4. Execute MainBookDAO.java to get logging messages on console.

The class is implementing the interface by default JDK's dynamic proxy mechanism will be used for proxy creation. But sometime the target object will not implement the interface then JDK's proxy mechanism will fail. In such cases CGLIB can be used to for proxy creation. To enable CGLIB proxies we can write the following configuration:

```

<aop:config proxy-target-class="true">
    <!-- aspect configuration à
</aop:config>

```

Also, to force the AspectJ and auto proxy support we can write the following configuration:

```

<aop:aspect-autoproxy proxy-target-class="true"/>

```

# Introduction

---

In enterprise application sometimes the developers come across the situations where they need to introduce set of new functionalities but without changing the existing code. Using introduction not necessarily all the interface implementations needs to be changed as it becomes very complex.

Sometimes developers work with third party implementations where the source code is unavailable introduction plays very important role. The developers may have an option to use decorator or adapter design pattern so that the new functionalities can be introduced. But method level AOP helps in achieving the introduction of new functionalities without writing decorators or adapters.

The Introduction is an advisor which allows introducing new functionalities while handling the cross cutting concerns. Introduce the new implementation the developers have to either use `<aop:declare-parents>` for schema based configuration or `@DeclareParents` if using annotation based implementations.

Using schema to add introduction the `<aop:declare-parent>` declares a new parent for the bean which is being advised. The configuration will be as:

```
<aop:aspect>
  <aop:declare-parents types-matching="" implement-interface=""
    default-impl="" />
</aop:aspect>
```

Where,

- **types-matching**- specifies the matching type of the been getting advised
- **implement - interface** - the newly introduced interface
- **default-impl** - the class implementing newly introduced interface

In case of using annotations the developers can use `@DeclareParents` which is equivalent to the `<aop:declare-parents>` configuration. `@DecalreParents` will be applied to the property which is the new interface introduced. The syntax of `@DeclarePrets` is as shown below:

```
@DeclareParents(value=" ", defaultImpl=" ")
```

Where,

- **value**- specifies the bean to be introduced with interface
- **defaultImpl** - is equivalent to default-impl of the <aop:declare-parent>'s attribute which specifies the class that provides implementation of the interface

Let's use Introduction in the JdbcTemplate application. The BookDAO doesn't have the method to get description of the book so let's add it. We will use Ch03\_JdbcTemplate as the base application. Follow the steps to use introduction:

1. Create a new Java application and name it as Ch04\_Introduction.
2. Add all the jar required for Spring core, Spring -jdbc, Spring AOP as we did in earlier applications.
3. Copy com.packt.ch03.beans package.
4. Create or copy com.packt.ch03.dao with BookDAO.java and BookDAO\_JdbcTemplate.java classes.
5. Copy connection\_new.xml in classpath and delete the bean having id as 'namedTemplate'.
6. Create new interface BookDAO\_new in com.packt.ch03.dao package as shown below to declare getDescription() method:

```
public interface BookDAO_new {  
    String getDescription(long ISBN);  
}
```

7. Create class BookDAO\_new\_Impl implementing BookDAO\_new interface which will deal with JDBC using JdbcTemplate. The code will be as shown below:

```
@Repository  
public class BookDAO_new_Impl implements BookDAO_new {  
    @Autowired  
    JdbcTemplate jdbcTemplate;  
    @Override  
    public String getDescription(long ISBN) {  
        // TODO Auto-generated method stub  
        String GET_DESCRIPTION=" select description from book w
```

```

        String description=jdbcTemplate.queryForObject(
            GET_DESCRIPTION, new Object[]{ISBN}, String.class);
        return description;
    }
}

```

8. Create an aspect class MyIntroductionAspect in com.packt.ch04.aspects package which will introduce the new interface to use getDescription() method. The code is as shown below:

```

@Aspect
public class MyIntroductionAspect {
    @DeclareParents(value="com.packt.ch03.dao.BookDAO+", 
    defaultImpl=com.packt.ch03.dao.BookDAO_new_Impl.class)
    BookDAO_new bookDAO_new;
}

```

The annotation provides introduction of BookDAO\_new which has additional methods than those available in BookDAO interface. The default implementation to be used for introduction is BookDAO-new\_Impl.

9. Register the aspect in connection\_new.xml as:

```
<bean class="com.packt.ch04.aspects.MyIntroductionAspect"><
```

10. Add the following configuration to enable autoproxy,

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

The proxy-target-class is used to force the proxy to be subclass of our class.

11. Copy or create MainBookDAO\_operations.java to test the code. Use getDescription() method to find description of the code. The underline statements in the following code are the additional statements to add:

```

public class MainBookDAO_operations {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ApplicationContext context=new
            ClassPathXmlApplicationContext("connection_new.xml");
        BookDAO bookDAO=(BookDAO)
            context.getBean("bookDAO_jdbcTemplate");
        //add book
    }
}

```

```

int rows=bookDAO.addBook(new Book("Java EE 7 Developer
Handbook", 97815674L,"PacktPub
publication",332,"explore the Java EE7
programming","Peter pilgrim"));
if(rows>0)
{
    System.out.println("book inserted successfully");
}
else
    System.out.println("SORRY!cannot add book");

//update the book
rows=bookDAO.updateBook(97815674L,432);
if(rows>0)
{
    System.out.println("book updated successfully");
}
else
System.out.println("SORRY!cannot update book");
String desc=((BookDAO_new)bookDAO).getDescription(97815
System.out.println(desc);

//delete the book
boolean deleted=bookDAO.deleteBook(97815674L);
if(deleted)
{
    System.out.println("book deleted successfully");
}
else
System.out.println("SORRY!cannot delete book");
}
}
}

```

As the BookDAO doesn't have getDescription() in order to use it, we need to cast the obtained object to BookDAO\_new.

12. On execution we will get the output on console as:

```

book inserted successfully
book updated successfully
explore the Java EE7 programming
book deleted successfully

```

The output clearly show though we are able to introduce the getDescription() method without changing BookDAO and its implementations.

# Chapter 5. Be Consistent: Transaction Management

In previous chapter we discuss in depth about Aspect Oriented Programming using logging mechanism as a cross cutting technology. Transaction management is another cross cutting technology which plays a very important role in application while dealing with persistency. In this chapter we explore transaction management by discussing the following points:

- What is transaction management?
- Importance of transaction management.
- Types of transaction management
- Spring and transaction management
- Annotation based transaction management in Spring framework

Number of developers frequently talk about the fancy term transaction management. How many of us find our self comfortable working with it or its customization. Is it really so difficult to understand? Does adding transaction to the code needs to add lots of complex code? No!! Actually, it's the easiest thing to understand as well as to develop. The transaction management is very much common while discussing, designing, developing a 'persistence layer' which deals with the handling of data to and from the database. The 'transaction' is a unit of sequential multiple database operations where either all the operations in it executed successfully or none of them. Transaction management is the technique which handles transaction by managing its parameters. The transaction maintains the consistency of the database depending upon given transaction parameters so that by either the transactional unit will be successful or failure. The transaction can never ever be partially successful or failed.

Now you may be thinking what's a big deal if anyone of them fails? And why it's been so important? Let's take a real time scenario to understand transaction. We want to open a account in one of the website for online shopping. We need to fill up the form giving some personal information and need to select the username using which we will do our online shopping. The

information will be collected by the application and then saved in two tables. One, for the users with has username as Primary Key and the second user\_info where user's personal information will be stored. After collection of data from user, the developers perform insertion operation for user's information in user\_info, followed by inserting the data in users table. Now consider a scenario the data collected from user gets inserted in user\_info table successfully but unfortunately the username was already existing in the table so the second operation failed. The database is in inconsistent state. Logically the data should be either added in both the tables or in none of them. But in our case data got inserted in one table but not in second. This happened because without checking whether the row got inserted or not we performed insertion operation permanently which now cannot be undo even on the failure of second operation. The transaction management helps the developers to maintain the consistency and the integrity of the database by either making all the operation reflected correctly in the database tables or none of them. If any operation in the unit fails, all the changes made before that will be cancelled. Of course, it won't happen automatically but the developers need to play a key role in that. In JDBC, instead of going with auto committing the operations developers choose to go with committing the transaction or rollback if any operation within it fails. These two are very important terms when it comes to transaction management. The commit reflects the changes in the database permanently. The rollback undoes all the changes made by all the previous operations before the failure happens and making the database back to its original state.

Following are the ACID properties which in 1970's Jim Gray defined to describe a transaction. The properties are later on known as ACID properties. Gray also describes the ways to achieve ACID properties. Let's discuss them one by one:

- **Atomicity:** While carrying out multiple operations one after another on the database either all the operations will be executed successfully or none of them. The developers can take the control on their hand to decide whether to change the database permanently by committing them or to rollback them. The rollback will undo all the changes done by the operations. Once the data is committed it can't be rolled back again.
- **Consistency:** To save the data in properly arranged and easily

maintainable format the rules, data types, associations and triggers has been set when table is created in the database. Consistency makes sure that when getting data transited from one state to another it will be changed keeping all the rules intact set on it.

- **Isolation:** In concurrency multiple transactions take place leading to the problem of data mismanagement. Isolation helps to keep the data in consistent state by locking mechanism. Unless one of the transaction is dealing with data is not getting completed it will keep the lock on it. Once the transaction completes its operations another transaction is allowed to use the data.

Following are the isolation levels defined by ANSI or ISO standards:

- **Dirty read:** Let's consider two transactions A and B, running on the set of data. Transaction A does some changes but yet not committed them. Meanwhile transaction B read the data along with the uncommitted changed data. If transaction A successfully completes its operation, both the transaction has same state of data. But if transaction A fails the data changed by it will be rolled back. The set of data with A and that with B will be different as B read the uncommitted data. The transaction B is using stale data leading to failure of the business logic of the application.
- **Non repeatable read:** Let's again consider transaction A and B which are running to complete few operations. Both of them reads the data, transaction A, changes some of the values and committed them successfully. Transaction B is still working on the previous set of the data which is stale leading to the undesirable effect. The situation can be avoided by keeping the lock on the data unless the first transaction is not completed.
- **Phantom read:** Transaction A and B has the set of data. With one of the criteria transaction A has performed searching operation. Let's say, A is searching data based on the name of the book. There are 8 rows in the database which has been returned to the transaction A. Meanwhile transaction B inserted a row in the table having the same value for the name which A was searching. A got stale data as in actual there are 9 rows the table but A got just 8.
- **Serializable:** This is the highest isolation level which locks the selected used data so that the problem occurred in phantom read will be avoided.

- Following are the default isolation levels supported by databases:

Database	Default isolation level
Oracle	READ_COMMITTED
Microsoft SQL Server	READ_COMMITTED
MySQL	REPEATABLE_READ
PostgreSQL	READ_COMMITTED
DB2	CURSOR STABILITY

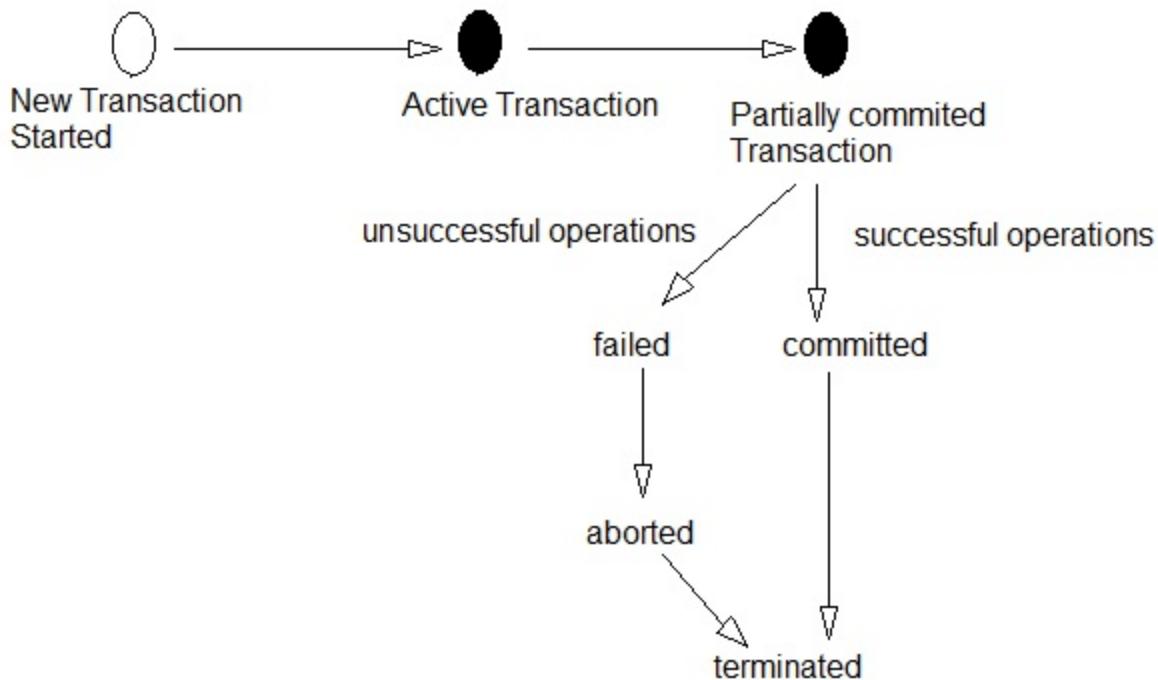
- **Durability:** The transaction keeps on changing by numerous operations simultaneously. Durability specifies once the data in the database is changed, added or updated it must be permanent.

Once we know the properties which describes transaction, knowing the stages in the progress of transaction helps us to use transaction management effectively.

# Life cycle of transaction management

---

Following diagram shows the stages in the progress of each transaction:



The newly started transaction will progress through the following stages:

1. **Active:** The transaction has just been stated and progressing ahead.
2. **Partially committed:** Once the operation has been performed successfully the generated values during it will be stored in the volatile storage.
3. **Failed:** The values generated before the failure are no longer required and will be removed from the volatile storage area by rollback them.
4. **Aborted:** The operation has been failed and is no longer continued further. It will be stopped or aborted.
5. **Committed:** All the operations successfully performed and all the temporary values generated during the operation will be stored permanently once the transaction is committed.

6. **Terminated:** When the transaction is committed or aborted, it reaches to its final stage the termination.

To handle transaction along with lifecycle steps and properties, one cannot neglect very important fact to know types of transaction. A transaction can be divided into either local transaction or global transaction

## Local transaction

The local transaction allows the application to connect to a single database and once all the operations in the transactions are completed successfully, it will be committed. The local transactions are specific to the resource and don't need any server to handle them. The configured DataSource object will return the connection object. This connection object further allows the developers to perform database operation as required. By default, such connections are auto committed. To take the control in hands, the developers can manually handle transactions using commit or rollback. The JDBC connection is the best example of local transactions.

## Global or distributed transaction

The global transactions are managed by the application servers like Weblogic, WebSphere. Global transaction facilitates to handle more than one resource and servers. The global transaction is comprises of many local transactions who access the resource. EJB's container managed transaction uses global transactions.

## Spring and Transaction management

The Spring framework excellently supports the integration of transaction managers. It supports Java Transaction API, JDBC, Hibernate and Java Persistent APIs. The framework supports abstract transaction management known as transaction strategy. The transaction strategy is defined through service provider interface (SPI) through PlatformTransactionManager interface. The interface has the methods to commit and rollback the

transaction. It also has the method to get the transaction specified by the TransactionDefinition. All of these methods throws TransactionException which is a runtime exception.

The getTransaction() method returns TransactionStatus depending upon the TransactionDefinition parameters. The TransactionStatus returned by the method represent a new transaction or the existing one. Following parameters can be specified to define the TransactionDefinition:

- **Propagation:** The propagation behaviour comes in discussion when one transactional method invokes other. In such invocation propagation behaviour states what transaction behaviour it will be performed. The invoking method may have started transaction, what the invoked method should do in such cases? Whether the invoked method start a new transaction, used the current one or it doesn't support transaction? The propagation behaviour can be specified using following values:
  - **REQUIRED:** It says the transaction is must. If No transaction exists it will create a new one.
  - **REQUIRES\_NEW:** It specifies to have a new transaction every time. The current transaction will be suspended. If no transaction exists it will create a new.
  - **MANDATORY:** It states the current transaction will be supported but if in case of no ongoing transaction an exception will be thrown.
  - **NESTED:** It states, if the current transaction exists the method will be executed within a nested transaction. If no transaction exists it will act as PROPAGATION\_REQUIRED.
  - **NEVER:** The transaction is not supported and if it exists an exception will be thrown.
  - **NOT\_SUPPORTED:** It states the transaction is not supported. If transaction exists opposite to NEVER it won't throw exception but suspends it.
- **Isolation:** We already had discussed in depth about isolation levels.
- **Timeout:** The timeout value for transaction mentioned in seconds.
- **Read only:** The attribute states the transaction will allowed to only read the data and no operation leading to updating the data will be supported.

Following are the advantages of using Spring framework's transaction management.

Spring facilitates the use of Transaction management by two ways as follow:

- Programmatic transaction management.
- Declarative transaction management.

Whether we are using Programmatic transaction or Declarative transaction the foremost important component is to define the PlatformTransactionManager using Dependency Injection(DI). One should have a clear idea to use local transaction or global as it is essential to define PlatformTransactionManager. Following are the few configuration which can be used to define PlatformTransactionManager:

- Using DataSource PlatformTransactionManager can be defines as:

```
<bean id="dataSource"
    <!--DataSource configuration -->
</bean>
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTran
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

- Using JNDI and JTA to define PlatformTransactionManager as shown below:

```
<jee: jndi-lookup id="dataSource" jndi-name="jdbc/books"/>
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransacti
    </bean>
```

- Using HibernateTransactionManager defines PlatformTransactionManager as:

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionfa
        <!--define parameters for session factory -->
    </bean>

<bean id=" transactionManager"
```

```
class="org.springframework.orm.hibernate5.HibernateTrans  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

Let's start with using the transaction management in Spring one by one,

## **Programmatic Transaction management**

The programmatic transaction management in Spring can be done by using either TransactionTemplate or PlatformTransactionManager.

### **Using PlatformTransactionManager**

The PlatformTransactionManager is at the centre of while discussing Spring's transaction management API. It has the functionalities to commit, rollback. It also provides a method which returns the currently active transaction. As it's an interface it can easily mocked or stubbed whenever required. Spring provides DataSourceTransactionManager, HibernateTransactionManager, CciLocalTransactionManager, JtaTransactionManager and OC4JJtaTransactionManager as few of the implementation of PlatformTransactionManager. To use PlatformTransactionManager any implementation of it can be injected in the bean to use for transaction management. Further, the objects of TransactionDefinition and TransactionStatus can be used to rollback or commit the transaction.

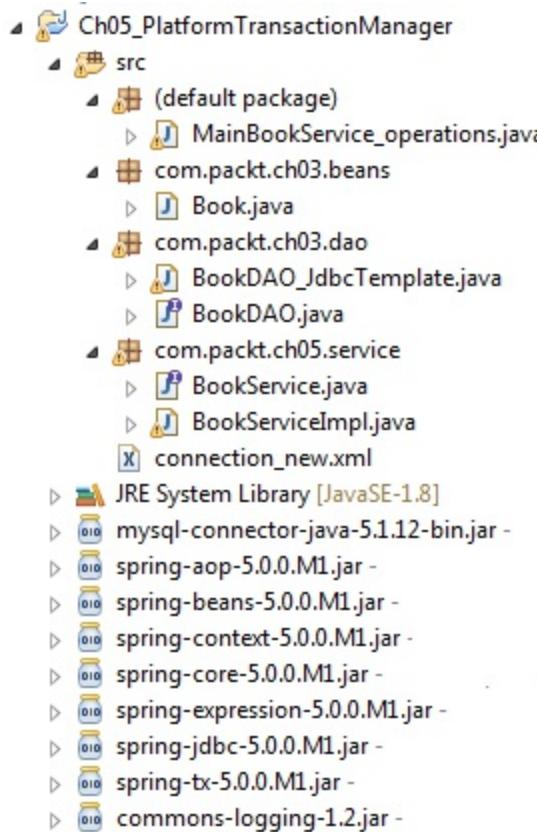
Before heading ahead, we need to discuss a very important point. Generally the application requirement decides whether to apply transaction to service layer or to DAO. But still it's a debatable question whether to apply transaction to DAO layer or to service layer. Applying transaction to DAO layer though keeps the transaction shorter, the biggest problem will occur of multiple transactions. And the concurrency has to make with very much care and unnecessarily the complexity will increase. The transaction when applied to service layer the DAO's will be using a single transaction. We will apply the transaction to service layer in our application.

To apply transaction management in application we can think about the following points,

- Whether to apply transaction to DAO layer or Service layer?
- Decide whether to use Declarative transaction or Programmatic transaction Management
- Define the PlatformtransactionManager to use in beans configuration.
- Decide Transaction attributes as Propagation Behaviour, Isolation level, Read Only, Timeout etc to be defined for the transaction.
- According to programmatic or declarative transaction management add the attributes to the transaction either in code.

Let's use transaction for better understanding. We will use JDBC operations in Ch03\_JdbcTemplate application developed in the third chapter as our base application. Let's follow the steps to use transaction using PlatformTransactionManager,

1. Create a new Java application named Ch05\_PlatformTransactionManager and add all the required jar for Spring core, Spring-jdbc, Spring-transaction, Spring-aop, commons-logging and mysql-connector.
2. Copy or create Book.java in com.packt.ch03.beans package.
3. Copy or create BookDAO.java and BookDAO\_JdbcTemplate.java in com.packt.ch03.dao package. The final outline of application will be as shown below:



4. We will add a new method in BookDAO to search the book as before adding it will be important to find out if there is any book in the 'Book' table with the same ISBN. If it's already exists, we don't want to unnecessary go ahead for adding it again. The newly added method will be as follows:

```
public Book serachBook(long ISBN);
```

5. The BookDAO\_JdbcTemplate.java needs to override the newly added method in the interface as:

```
@Override
public Book serachBook(long ISBN) {
    // TODO Auto-generated method stub
    String SEARCH_BOOK = "select * from book where ISBN=?";
    Book book_serached = null;
    try {
        book_serached = jdbcTemplate.queryForObject(SEARCH_BOOK,
            new Object[] { ISBN },
            new RowMapper<Book>() {
```

```

    @Override
    public Book mapRow(ResultSet set, int rowNum)
    throws SQLException {
        Book book = new Book();
        book.setBookName(set.getString("bookName"));
        book.setAuthor(set.getString("author"));
        book.setDescription(set.getString("description"));
        book.setISBN(set.getLong("ISBN"));
        book.setPrice(set.getInt("price"));
        book.setPublication(set.getString("publication"))
        return book;
    }
}
return book_serached;
} catch (EmptyResultDataAccessException ex) {
    return new Book();
}
}
}

```

We have added an anonymous inner class which is implementing RowMapper to bind object the fetched data from the database using queryForObject() method to the data members of the Book object. The code is searching for the book and then the column values from ResultSet will be bounded to the Book object. We returned an object with default values just for our business logic.

6. Add BookService interface as a service layer in com.packt.ch05.service package with following method signatures:

```

public interface BookService {
    public Book searchBook(long ISBN);
    public boolean addBook(Book book);
    public boolean updateBook(long ISBN, int price);
    public boolean deleteBook(long ISBN);
}

```

7. Create BookServiceImpl implementing BookService. As it's for service annotate the class with `@Service`.
8. Add two data members to the class first of type PlatformTransactionManager to handle transactions and second of type BookDAO to perform JDBC operations. For dependency injection annotate both of them by `@Autowired`.
9. Let's first off all develop searchBook() method of service layer for

handling read-only transaction in two steps as follows:

- Create an instance of TransactionDefinition().
- Create an instance of TransactionStatus obtained from TransactionManager who uses an instance of TransactionDefinition created in previous step. The TransactionStatus will provide the status information of transaction which will be used to commit or .rollback the transaction.

Here make the transaction read-only by setting the property to true as we just want to search the book and don't want to perform any updating on the DB side. The code developed till this step will be as:

```
@Service(value = "bookService")
public class BookServiceImpl implements BookService {
    @Autowired
    PlatformTransactionManager transactionManager;

    @Autowired
    BookDAO bookDAO;

    @Override
    public Book searchBook(long ISBN) {
        TransactionDefinition definition = new
            DefaultTransactionDefinition();
        TransactionStatus transactionStatus =
            transactionManager.getTransaction(definition);
        //set transaction as read-only
        ((DefaultTransactionDefinition)
        definition).setReadOnly(true);
        Book book = bookDAO.serachBook(ISBN);
        return book;
    }
    // other methods from BookService
}
```

The way we updated read-only property of transaction, we can set other properties as isolation level, propagation, timeout in the same way.

10. Let's add addBook() method to service layer to find out whether the book with same ISBN already exist and if not insert a row in table. The code will be as:

```
@Override
```

```

public boolean addBook(Book book) {
    // TODO Auto-generated method stub
    TransactionDefinition definition = new
        DefaultTransactionDefinition();
    TransactionStatus transactionStatus =
        transactionManager.getTransaction(definition);

    if (searchBook(book.getISBN()).getISBN() == 985645671) {
        System.out.println("no book");
        int rows = bookDAO.addBook(book);
        if (rows > 0) {
            transactionManager.commit(transactionStatus);
            return true;
        }
    }
    return false;
}

```

`transactionManager.commit()` will commit the data permanently to the book table.

11. In the same way let's add `deleteBook` and `updateBook()` methods as shown below,

```

@Override
public boolean updateBook(long ISBN, int price) {
    TransactionDefinition definition = new
        DefaultTransactionDefinition();
    TransactionStatus transactionStatus =
        transactionManager.getTransaction(definition);
    if (searchBook(ISBN).getISBN() == ISBN) {
        int rows = bookDAO.updateBook(ISBN, price);
        if (rows > 0) {
            transactionManager.commit(transactionStatus);
            return true;
        }
    }
    return false;
}

@Override
public boolean deleteBook(long ISBN)
{
    TransactionDefinition definition = new
        DefaultTransactionDefinition();
    TransactionStatus transactionStatus =

```

```

        transactionManager.getTransaction(definition);
        if (searchBook(ISBN).getISBN() != 985645671) {
            boolean deleted = bookDAO.deleteBook(ISBN);
            if (deleted) {
                transactionManager.commit(transactionStatus);
                return true;
            }
        }
        return false;
    }
}

```

12. Copy or create connection\_new.xml for the bean configurations. Add a bean for DataSourceTransactionManager as we had seen earlier while discussing how to configure PlatformTransactionManager using DataSource.
13. Update package scanning from XML as we want to consider newly added package as well. The updated configuration will be as follows:

```

<context:component-scan base-package="com.packt.*">
</context:component-scan>

```

14. The final set will be to add Main code in MainBookService\_operation.java which will invoke methods from service layer using BookServiceImpl object as we did earlier for BookDAO\_JdbcTemplate object. The code will be as shown below:

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    ApplicationContext context = new
        ClassPathXmlApplicationContext("connection_new.xml");
    BookService service = (BookService)
        context.getBean("bookService");
    // add book
    boolean added = service.addBook(new Book("Java EE 7
        Developer Handbook", 97815674L, "PacktPub
        publication", 332, "explore the Java EE7
        programming", "Peter pilgrim"));
    if (added) {
        System.out.println("book inserted successfully");
    } else
        System.out.println("SORRY!cannot add book");
    // update the book
    boolean updated = service.updateBook(97815674L, 800);
    if (updated) {

```

```

        System.out.println("book updated successfully");
    } else
        System.out.println("SORRY!cannot update book");
        // delete the book
        boolean deleted = service.deleteBook(97815674L);
        if (deleted) {
            System.out.println("book deleted successfully");
        } else
            System.out.println("SORRY!cannot delete book");
    }
}

```

## The TransactionTemplate

Use of thread safe TransactionTemplate helps developers to get rid from boilerplate code as already discussed with JdbcTemplate. It make the programmatic transaction management simple yet powerful with the help of callback methods. Using TransactionTemplate becomes easy as it has different setter methods to customize various transaction properties as isolation level, propagation behaviour etc. The first step to use Transaction template will be to get it's instance by providing transaction manager. The second step will be to get instance of TransactionCallback which will be passed to the execute method. The following example will demonstrate how to use the template where we don't have to create TransactionDefinition as we did in earlier application,

1. Create a Java application naming Ch05\_TransactionTemplate and copy all the required jar which we added in earlier application.
2. We will keep the outline of the application same as that of Ch05\_PlatformTransactionManager application so you can copy the beans, dao and service package as it is. The only change which we will make is to use TransactionTemplate instead of using PlatformTransactionManager in BookServiceImpl.
3. From BookServiceImpl delete data member PlatformTransactionManager and add TransactionTemplate.
4. Annotate it with @Autowired to use DI.
5. We will update the searchBook() method to use TransactionTemplate by setting it as a read-Only transaction using setReadOnly(true). The TransactionTemplate has a callback method as 'execute()' where the business logic can be written to execute. The method is expecting an

instance of TransactionCallback and it will return searched book. The code will be as shown below:

```
@Service(value = "bookService")
public class BookServiceImpl implements BookService {
    @Autowired
    TransactionTemplate transactionTemplate;

    @Autowired
    BookDAO bookDAO;

    public Book searchBook(long ISBN) {
        transactionTemplate.setReadOnly(true);
        return transactionTemplate.execute(new
            TransactionCallback<Book>()
        {
            @Override
            public Book doInTransaction(TransactionStatus status)
                // TODO Auto-generated method stub
                Book book = bookDAO.serachBook(ISBN);
                return book;
        }
    );
}
```

To perform the task, we have created instance of TransactionCallback by using concept of inner class. The generic type specified here is, Book as it is the return type of the searchBook() method. The class is overriding doInTransaction()method to invoke the business logic from DAO's searchBook()method.

One more implementation of TransactionCallback can be written using TransactionCallbackWithoutResult. It can be used in case where the service method is not returning anything or having void as its return type.

6. Let's now add addBook(). The very first thing we have to find whether the book exists in table or not using searchBook(). If Book doesn't exist add the book. But as searchBook() has made transaction read-only we need to change the behavior. As Add book has boolean as its return type we will use TransactionCallBack of Boolean type. The code will be as shown below:

```
@Override
```

```

public boolean addBook(Book book) {
    // TODO Auto-generated method stub
    if (searchBook(book.getISBN()).getISBN() == 985645671)
    {
        transactionTemplate.setReadOnly(false);
        return transactionTemplate.execute(new
            TransactionCallback<Boolean>()
        {
            @Override
            public boolean doInTransaction(TransactionStatus stat
                try {
                    int rows = bookDAO.addBook(book);
                    if (rows > 0)
                        return true;
                } catch (Exception exception) {
                    status.setRollbackOnly();
                }
                return false;
            }
        });
    }
    return false;
}

```

The code clearly shows the TransactionTemplate gives us the power of changing the properties of the Transaction yet to internally managing the transaction without writing the boilerplate code as PlatformTransactionManager has to.

7. In the same way we can add the code for deleteBook and updateBook(). You can find the complete code in source code.
8. Copy connection\_new.xml from Ch05\_PlatformTransactionmanager in classpath and add a bean for TransactionTemplate as follows:

```

<bean id="transactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager"
              ref="transactionManager"/>
</bean>

```

We already had a bean for transactionManager so we will not add it here again.

9. Copy MainBookService\_operations.java in default package to test the

code. We will get the code executed successfully.

10. Before moving ahead just modify the code of doInTransaction() of searchBook() method as follows;

```
public Book doInTransaction(TransactionStatus status) {  
    //Book book = bookDAO.serachBook(ISBN);  
    Book book=new Book();  
    book.setISBN(ISBN);  
    bookDAO.addBook(book);  
    return book;  
}
```

11. On execution we will get the stack trace which denotes read-only operations are not allowed to modify the data as follows:

```
Exception in thread "main"  
org.springframework.dao.TransientDataAccessResourceException:  
PreparedStatementCallback; SQL [insert into book values(?,?)]  
Connection is read-only. Queries leading to data modification  
allowed; nested exception is java.sql.SQLException:  
Connection is read-only.
```

## Declarative transaction management

The Spring framework uses AOP to facilitate declarative transaction management. The best things about declarative transaction is, it necessarily does not required to be managed by application server and it can be applied to any class. The framework also facilitates developers to customize the transactional behavior by using AOP. The declarative transaction can either be XML based or annotation based configuration.

### XML based declarative transaction management:

The framework offers the rollback rules to specify on which types of exception the transaction will be rollback. The rollback rules can be specified in XML as follows,

```
<tx:advise id=:transactionAdvise" transaction-manager="transactionManager">  
    <tx:attributes>  
        <tx:method name="find*" read-only="true"  
            rollback-for ="NoDataFoundException">
```

```
</tx:attributes>
</tx:advise>
```

The configuration can even specify attributes as,

- **'no-rollback-for'** - to specify the exception when we don't want the transaction to rollback.
- **propagation** - to specify the propagation behavior of transaction with 'REQUIRED' as its default value.
- **isolation** - to specify the isolation level.
- **timeout** - transaction timeout value in seconds with '-1' as default.

As now a days we more tend to use Annotation based transaction management without wasting time let's move on to annotation based transaction management.

### Annotation based transaction management

The `@Transaction` annotation facilitates to develop annotation based declarative transaction management which can be applied to interface level, class level as well as method level. To enable the annotation based support one need to configure the following configuration along with the transaction manager,

```
<bean id="transactionManager" class=" your choice of transaction :
  <!--transaction manager configuration -->
</bean>
<tx:annotation-driven transaction-manager="transcationManager"/>
```

The attribute 'transaction-manager' can be omitted if the bean written for PlatformTransactionManager has the name as 'transactionManager'.

Following are the attributes which can be use to customize the behavior of transaction,

- **value** - to specify the transaction manager to be used.
- **propagation** - to specify the propagation behavior.
- **isolation** - to specify the isolation levels.
- **readonly** - to specify the read or write behavior.

- **timeout** - to specify the transaction timeout.
- **rollbackForClassName** - to specify the array of exception classes who causes the transaction to rollback.
- **rollbackFor** - to specify the array of exception classes who causes the transaction to rollback.
- **noRollbackFor** - to specify the array of exception classes who doesn't causes the transaction to rollback.
- **noRollbackForClassName** - to specify the array of exception classes who doesn't causes the transaction to rollback.

Let's use the `@Transactional` to demonstrate declarative transaction management in the application instead of programmatic transaction management with the help of following steps:

1. Create Ch05\_Declarative\_Transaction\_Management and add the required jars as we did in earlier application.
2. Copy com.packt.ch03.beans and com.packt.ch03.dao from Ch05\_PlatformTransactionManager application.
3. Copy the interface BookService.java in com.packt.ch05.service packages.
4. Create a class BookServiceImpl in com.packt.ch05.service package and add a data member of type BookDAO.
5. Annotate the data member of type BookDAO with `@Autowired`.
6. Annotate `searchBook()` with `@Transactional(readOnly=true)` and write the code to search data using `JdbcTemplate`. The class will be as follows:

```
@Service(value = "bookService")
public class BookServiceImpl implements BookService {

    @Autowired
    BookDAO bookDAO;

    @Override
    @Transactional(readOnly=true)
    public Book searchBook(long ISBN)
    {
        Book book = bookDAO.serachBook(ISBN);
        return book;
    }
}
```

7. Copy connection\_new.xml from Ch05\_PlatformTransactionManager in the classpath.
8. Now, we need to tell the Spring to find out all the beans which has been annotated by `@Transactional`. It will be simply done by adding the following configuration in XML:

```
<tx:annotation-driven />
```

9. To add the above configuration we first have to add 'tx' as a namespace in XML. Update the schema configuration from connection\_new.xml as follows:

```
<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"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd">
```

10. Now, we can add the following configuration:

```
<tx:annotation-driven />
```

11. Copy MainBookService\_operation.java and execute it to get the output.
12. Now add the method addBook() to understand readOnly=true. The code will be as follows:

```
@Transactional(readOnly=true)
public boolean addBook(Book book) {
    if (searchBook(book.getISBN()).getISBN() == 985645671) {
        System.out.println("no book");
        int rows = bookDAO.addBook(book);

        if (rows > 0) {
            return true;
        }
    }
    return false;
}
```

13. Execute the MainBookService\_operation.java and execute it to get the following output specifying read-only transaction not allowed to modify data:

```
Exception in thread "main"
org.springframework.dao.TransientDataAccessResourceException
PreparedStatementCallback; SQL [insert into book values(?,?
Connection is read-only. Queries leading to data modification
allowed; nested exception is java.sql.SQLException:Connecti
Queries leading to data modification are not allowed
```

14. Edit addBook() for remove read-only transaction by specifying read-only=false which is default behavior of Transaction.
15. The main code will execute successfully performing the operations.

## Note

Use Programmatic transaction management if the application has few transaction operations using TransactionTemplate. In case of having numerous transactional operations to keep it simple and cohesive choose declarative transaction management.

# Summary

---

We discussed in this chapter about Transaction and why it is important. We also discussed about transaction management and its life cycle. We discuss about transaction attributes as read-only, isolation level, propagation behavior and time out. We see declarative and programmatic as two ways to handle transaction where one gives the other get rid from the plumbing code and other gives fine control of the operations. We also discuss both these techniques with help of an application for better understanding. Up till now we had discuss about how to handle the data which was imaginary. We need a means to get this for the actual users.

In the next chapter we will explore how to develop the web layer of an application which facilitates us have to have user interaction.

# Chapter 6. Explore Spring MVC

Up till now we have discussed about the Spring framework to handle, initialize and use of the data considering console as our output. We haven't taken any effort on either of its presentation or any user interaction. It seems very boring in today's world to work with old style window based, very bland presentation. We want something much more interesting and exciting. Internet is the something which made the world closer than ever before as well as interesting. Today's world is the world of web, so how could we be apart from it? Let's dive into an amazing world of internet to explore the power of Spring with the help of following point:

- Why is it necessary to learn web application development using Spring?
- How to develop web application using Spring MVC
- What are different components of Spring MVC?
- How to pre populate the form and bind the data to an object?
- We will also discuss about how to perform validations in spring

In the 1990s the world of internet opened the doors of a complete new world to us. It was the ocean of data which no one had seen ever before. Before internet, the data was only available through hard copies; mainly books and magazines. In early days the internet was used just to share the static data, but over the years the dimensions, meaning and the use of internet had changed a lot. Now a day, we cannot imagine the world without internet. It is simply next to impossible. It has become a part of our day today life and a very major source of our business industry as well. As a developer for us also it's very important to know about the web application, its development, the challenges and how to overcome that.

In Java, the basic web application can be created using Servlet and JSP, but then lot many evolutions happened. These evolutions are mainly due to high demand of changing world in less of the time. Not only presentation, but also the overall web experienced has changed using HTML5, CSS, JavaScript, AJAX, Jquery and many similar technologies. The Servlets handle the web request and use the data from the request parameters to extract the data for

the dynamic web applications.

While using Servlets and JSPs the developers have to take lots of efforts to perform the data conversion and bind the data to the objects. Apart from their main role in performing the business logic, now they have to handle the extra burden of request and response presentation as well.

The developers mainly work on the data extracted from the request in web application. They develop the complex, lengthier business logic based on the rules to perform the task. But it all will be useless if the data extracted from request parameters is incorrect. It's obviously not the fault of the developers but still their business logic suffered and there is no point in carrying out business logic with such data values. The developers now have to take all the care to first find out whether the data extracted from the request is correct or not before performing business logic. The developers also have to extensively involve in the presentation of the data to the response. To present the data first the developers needs to bind the data to the response and then further how to extract it on the presentation side.

Each one of the above discussed tasks adds extra burden on the development side within limited time. The Spring framework facilitates the developers for easy and faster development by the following features:

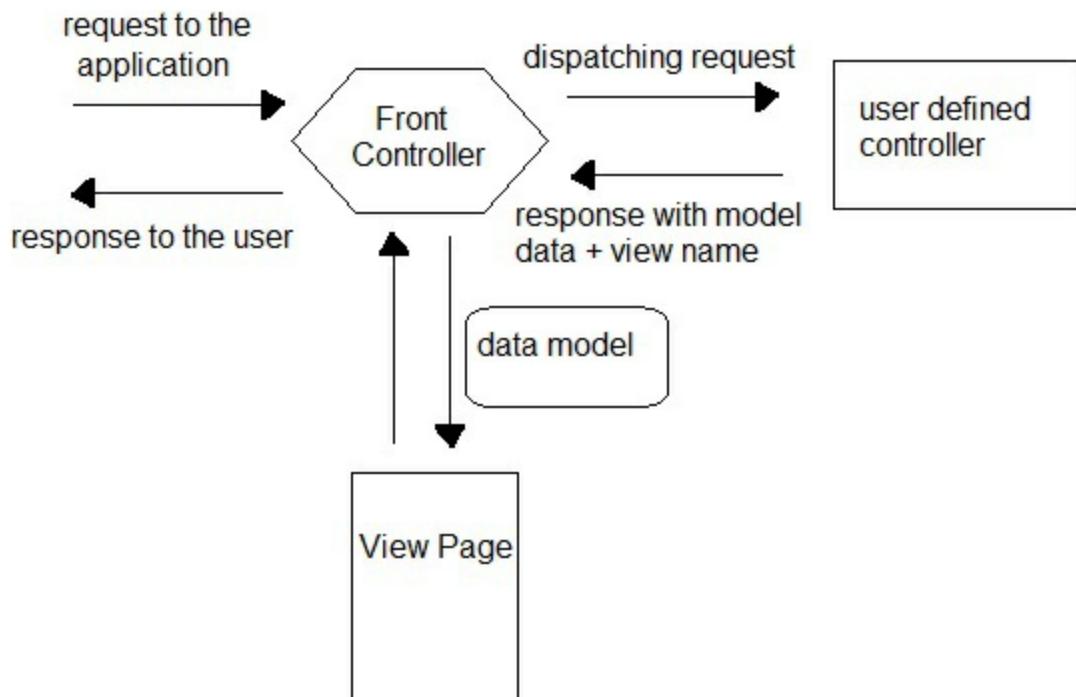
- Spring framework supports MVC architecture which gives clear separation of Model, View and Controller.
- The framework facilitates the developers with power of beans for easy handling of data by binding the request parameters to the command object.
- It provides easy validation of request parameters performing validation either with Validator interface or using annotations. It can also supports customization of validation rules.
- It provides the annotations such as `@RequestParam`, `@RequestHeader` which facilitates the request data binding to the method parameters without getting involved in Servlet APIs.
- It has support for a wide range of view templates such as JSTL, Freemarker, Velocity and many other.
- The easy transfer of the data from Controller to the view made easy with the use ModelMap object.

- It can be easily pluggable to other frameworks like Apache Struts2.0, JSF and many more.

Generally, the web applications are deployed on the web Servers. Each resource in the application is mapped with the URL and users use these URLs to access the resources. The Servlet or JSP reads the data from the request object, performs the business logic on it and then returns the response as a result. We all are well aware of this general flow takes places in any web application. In this flow the first and for most concern is that these web applications don't have any Servlet or controller who manages the flow of entire application. Yes, the first attainer to the application is absent. The entire application and its flow has to be maintained by the development side. This is where the major different in between Servlets and Spring lies.

---

The Spring framework works on MVC design pattern, it provides a Front controller which handles or attains each request hitting to the application. The following figure shows how the Spring MVC handles the request and all the components are part of Spring MVC:



The following steps gives us the orientation of the flow of the Spring MVC web application:

- Each incoming request will first hit the Front Controller which is the heart of the application. The Front Controller dispatches the request to the handlers and allows the developers to use different features of the framework.
- The Front Controller has its own WebApplicationContext which has been inherited from the root WebApplicationContext. The beans configured in the root application can be accessed and shared between the context and the Servlet instance of the application. As applicable to all the Servlets the Front Controller gets initialized on the first request.
- Once the Front Controller is initialized, it looks further for a XML file

named as `servlet_name -servlet.xml` under the WEB-INF folder. It contains the MVC specific components.

- This configuration file is by default named as `XXX-servlet.xml` under WEB-INF folder. This file contains the mapping information of the URL to the controllers which can handle the incoming request. Before Spring 2.5 the mapping was must for discovery of the handlers, which now we don't need. We now can directly use the annotation based controllers.
- The `RequestMappingHandlerMapping` searches all the controllers to look for `@RequestMapping` annotation under `@Controller`. These handlers can be used to customize the way URLs are searched by customizing the properties like `interceptor`, `defaultHandler`, `order`, `alwaysUseFullPath`, `urlDecode`.
- After scanning all the user defined Controllers the appropriate controller based on URL mapping will be chosen and appropriate method will be invoked. The method selection took place based on the URL mapping and the HTTP method which it supported.
- On execution of business logic written in the controller method, now it's time to generate the response. This is different than our usual `HTTPResponse` as it won't be served to the user directly. Instead the response will be sent to the Front Controller. Here the response contains the logical name of the view, the logic name of the model data and the actual data to bind. Usually instance of the `ModelAndView` is returned to the `FrontController`.
- The logical view name is with Front Controller, but it doesn't give any information about the actual view page to return to the user. The bean for `ViewResolver` configured in the `XXX-servlet.xml` file will be the mediator to map view name to the actual page. There is a wide range of view resolvers supported by the framework, we will discuss them shortly.
- The `ViewResolver` helped to get the actual view which Front Controller can return as a response. The `FrontController` will render it by extracting the values from the bounded model data and will return it to the user.

In the flow discussion we have used many names as Front Controller, `ModelAndView`, `ViewResolver`, `ModelMap` etc. Let's discuss them classes in

depth.

## DispatcherServlet

The DispatcherServlet acts as the Front Controller in Spring MVC applications where first off all each incoming request will hit. It is basically used to handle the HTTP requests as it has been inherited from HttpServlet. It delegates the request to the controllers, resolves which view to send back as response. The following configuration shows the Dispatcher mapping in the web.xml (deployment descriptor),

```
<servlet>
    <servlet-name>books</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>books</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

The above configuration state all the request with \*.htm as URL pattern will be handled by the Servlet named 'books'.

Sometimes the application demands for multiple configuration files, few of them are in root WebApplicationContext handling beans for Database and few in Servlet application context containing beans defined to be used in controllers. The following configuration can be used to initialize beans from multiple WebApplicationContexts. The following configuration can be used to load multiple configuration files from the context as,

```
<servlet>
    <servlet-name>books</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>books</servlet-name>
    <url-pattern>*.htm</url-pattern>
```

```
</servlet-mapping>
```

## Controller

The Spring controllers handles the request for carrying out the business logic, these controllers can also be referred as 'handlers' and their methods as handler methods. The Spring provides AbstractUrlViewController, ParameterizableViewController, ServletForwardingController, ServletWrappingControllerBefore as controllers. One of these controllers needs to be subclassed to customize the controller in Spring 2.5 based web application. But now, Spring supports annotation driven controllers by @Controller annotation. The following configuration enables the annotation based controllers:

```
<mvc:annotation-driven />
```

The annotation based controllers need to be discovered to execute the handler method. The following configuration provides information to the framework about which packages to scan to discover the controllers:

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

The @RequestMapping annotation is used to annotate either a class or a method for the declaration of specific URL which it can handle. Sometimes more than one method can be annotated for the same value of the URL which supports different HTTP methods. The 'method=RequestMethod.GET' attribute of the @RequestMapping is used to specify which HTTP method will be handled by the method.

## ModelAndView

The ModelAndView plays vital role for response generation. The instance of the ModelAndView facilitates binding of model data to its logical name, logical view name. The object which holds the data to be used in view is generally referred as model data. The following snippet makes clear how binding happens:

```
new ModelAndView(logical_name_of_view,logical_name_of_model_data,  
actual_value_of_model_data);
```

We can even use the following snippet code:

```
ModelAndView mv=new ModelAndView();  
mv.setViewName("name_of_the_view");  
mv.setAttribute(object_to_add);
```

## ModelMap

The ModelMap interface is the subclass of LinkedHashMap used in building of the model data using key and value pair. It has addAttribute() method providing the binding of model and logical name of the model. The attribute set in ModelMap can be used by the views for form data binding on form submission. We will discuss this in depth shortly.

## ViewResolver

The logical view name and other details returned by the user defined controller to the Front Controller. The view name is a String which needs to be resolved by ViewResolver.

Following are few ViewResolvers which can be used to render the view:

- **XmlViewResolver:** The XmlViewResolver helps in viewing the file written in XML. It uses the default configuration from WEB-INF/views.xml which contains the view beans having the same DTD as that of Spring beans configuration file has. The configuration can be written as shown below:

```
<bean id="myHome"  
      class="org.springframework.web.servlet.view.JstlView">  
    <property name="url" value="WEB-INF/jsp/home.jsp"/>  
<bean>
```

- The logical view name 'myHome' is mapped to the actual view 'WEB-INF/jsp/home.jsp'.
- One bean can also be referred the view mapped for some other bean as:

```

<bean id="logout"
      class="org.springframework.web.servlet.view.RenderView">
    <property name="url" value="myHome"/>
<bean>

```

- The 'logout' bean is not mapped for any actual view file, but it is using the bean 'myHome' to give the actual view file.
- **UrlBasedViewResolver:** It gives the direct mapping of the URL's to the logical view name. It will be preferred where the logical names match to the view resource. It has prefix and suffix as its properties helps in getting the actual view names with its location. The class is unable to resolve the views which are based on the current locale. To enable URLBasedViewResolver the following configuration can be written as:

```

<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolve
<property name="viewClass" value=
        "org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
<bean>

```

- The JstlView is used to render the view page. The page name and location is our case is 'prefix+ view\_name\_from\_controller+suffix'.
- **InternalResourceViewResolver:** The InternalResourceViewresolver is subclass of UrlBasedViewResolver used to resolve the internal resource which can serve as views using properties like prefix and suffix similar to its parent class. AlwaysInclude, ExposeContextBeansAsAttributes, ExposedContextBeanNames are few extra properties of the class adding advantage of using it more frequently than its parent class. The following configuration is similar to the way we configure UrlBasedViewResolver in previous example:

```

<bean id="viewResolver" class=
"org.springframework.web.servlet.view.InternalResourceViewResol
<property name="viewClass" value=
        "org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
<bean>

```

- It can verify the existence of the page only when it lands to it and not before that.
- **ResourceBundleViewResolver:** The ResourceBundleViewResolver uses the definition from the ResourceBundle specified in the configuration. The default file is used to define the configuration is views.properties. The configuration will be as,

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceViewResolver">
    <property name="base" value="web_view"/>
</bean>
```

- The view.properties will specify the details of the View class to be used and the url mapping to the actual view as follows:

home.(class)= org.springframework.wev.servlet.view.JstlView

- The following line states the mapping of view named homepage:

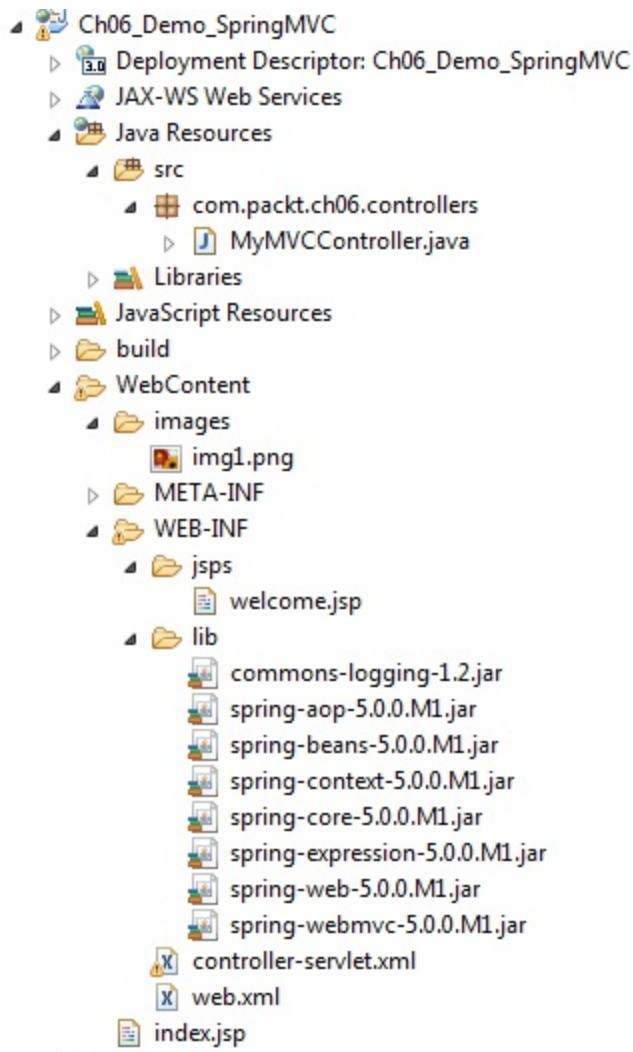
homepage.url= /WEB-INF/page/welcome.jsp

- **TilesViewResolver:** Tiles framework is used to define layout templates of the pages which can be reused keep consistent look and feel of the application. The page definitions defined in 'tiles.def' file as tile, header, footer, menus which will assembled in the page as runtime. The logical name returned by the controller matches the name of the tiles template which will be rendered by the view resolver.

Apart from the above discussed view resolvers Spring also has FreeMarkerViewResolver, TileViewResolver, VelocityLayoutViewResolver, VelocityViewResolver, XsltViewResolver.

Before continuing the discussion ahead let's first develop a sample demo to understand the flow of the application in detail which gives the orientation of the above discussion with the help of following steps,

1. Create Ch06\_Demo\_SpringMVC as a dynamic web application.
2. Copy the jars for spring-core, spring-context, commons-logging, spring-web and spring-webmvc as shown in the below project outline:



3. Create `index.jsp` in `WebContent` folder which works as the home page. The name can be customized as per your requirement the way we do in any Servlet application.
4. Add a link in `index.jsp` which gives navigation to the controller as shown below:

```

<center>
    
    <br>
</center>
<a href="welcomeController.htm">Show welcome message</a>

```

5. Whenever the user clicks the link the request will be generated having URL 'welcomeController.htm', it will be attended by the Front

Controller.

6. It's time to configure the Front Controller in `web.xml` as follows:

```
<servlet>
    <servlet-name>books</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>books</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

7. The information to find and invoke the methods of the Controllers the Front controller will look for the file having name as `servlet_name-servlet.xml` in WEB-INF. In our case 'books' is the name of the Servlet. So let's create file with name '`books-servlet.xml`' under WEB-INF folder.
8. The file should have configuration for which package Spring container will be scanning to find the Controllers. The configurations will be as shown below:

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

The above configuration says all the controllers will be scanned from '`com.packt`' package.

9. Create a class `MyMVCCController` in the package `com.packt.ch06.controllers`.
10. Annotate the class by `@Controller`. Annotating the class facilitates it to use the feature of handling the request.
11. Let's add `welcome()` method to handle the request by annotating it with `@RequestMapping` as shown below:

```
@Controller
public class MyMVCCController {
    @RequestMapping(value="welcomeController.htm")
    public ModelAndView welcome()
    {
        String welcome_message="Welcome to the wonderful
```

```

        world of Books";
        return new ModelAndView("welcome", "message", welcome_mes
    }
}

```

The controllers can have multiple methods, which will be invoked as per the URL mapping. Here we are declaring the method which will be invoked for the 'welcomeController.htm' URL.

The method performs the business logic of generating the welcome message and generates the response with the help of `ModelAndView` as shown below,

```

new ModelAndView("welcome", "message", welcome_message);
The ModelAndView instance is specifying,
Logical name of the view - welcome
Logical name of the model data - message
Actual value of the model data - welcome_message

```

Alternative to the above code you can also use the code shown below:

```

 ModelAndView mv=new ModelAndView();
mv.setViewName("welcome");
mv.addObject("message", welcome_message);
return mv;

```

We can have multiple methods mapped to the same URL supporting different HTTP methods as shown below:

```

@RequestMapping(value="welcomeController.htm",
    method=RequestMethod.GET)
public ModelAndView welcome() {
    //business logic
}
@RequestMapping(value="welcomeController.htm",
    method=RequestMethod.POST)
public ModelAndView welcome_new() {
    //business logic
}

```

12. Configure the bean for `ViewResolver` in `books-servlet.xml` as shown below:

```

<bean id="viewResolver" class=
"org.springframework.web.servlet.view.InternalResourceViewReso

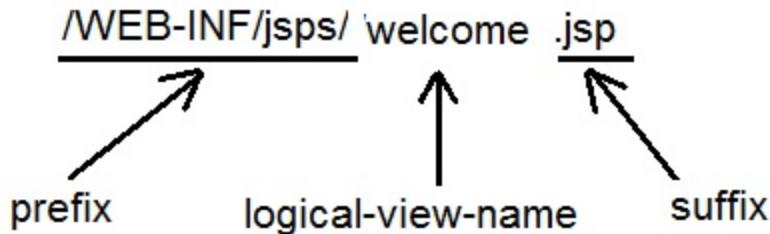
```

```

<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

```

The ViewResolver helps the Front Controller to get the actual view name and the location. The response page which Font Controller returns to the browser in our case it will be:



13. Create a folder named `jsp`' in WebContent.
14. Create a `welcome.jsp` page in the `jsp` folder, which will display the welcome message using Expression Language as:

```

<body>
  <center>
    
    <br>
  </center>
  <center>
    <font color="green" size="12"> ${message } </font>
  </center>
</body>

```

The attribute '`message`' is used in EL as it's the logical model name used by us in the controller method bounded to the object of `ModelAndView`.

15. Configure the tomcat server and run the application. The link will be shown on the browser. Clicking on the link we will get the output as shown the screen shot below:



## Welcome to the wonderful world of Books!!!

The demo introduced us to the Spring MVC flow. Let's now develop the book application step by step covering the following cases:

- Reading the request parameters
- Handling Form Submission

### Case1: Reading request parameters

Let's start with reading the request parameters with help of following steps:

1. Create ReadMyBooks as dynamic web application, and add all required jars to it as we did earlier.
2. Each application has home page. So, let's add index.jsp as a home page from the earlier application. You can copy and paste it directly.
3. Copy images folder from the earlier application.
4. Add one more link to search book on author's name as shown below,

```
<a href="searchByAuthor.jsp">Search Book By Author</a>
```

5. Let's add searchByAuthor.jsp page facilitating user to request for list of books by entering author's name as shown below:

```
<body>
    <center>
        
        <br>

        <h3>Add the Author name to search the book List</h3>
```

```

<form action="/searchBooks.htm">
    Author Name:<input type="text" name="author">
    <br>
    <input type="submit" value="Search Books">
</form>
</center>
</body>

```

6. Add the configuration for DispatchServlet as front controller in web.xml as we did earlier. Name the servlet as 'books'.
7. Create or copy books -servlet.xml for configuring handler mapping and other web component mappings from earlier application.
8. Add the configuration for scanning the controllers using 'context' namespace.
9. We need Book bean to handle data to and fro the controller. So, before developing controllers code add Book.java to com.packt.ch06.beans package from our earlier applications having the data members as shown below:

```

public class Book {
    private String bookName;
    private long ISBN;
    private String publication;
    private int price;
    private String description;
    private String author;
    //getters and setters
}

```

10. Now create a class SearchBookController as a controller in com.packt.ch06.controllers package and annotate it by @Controller.
11. To search the books, add method named searchBookByAuthor() and annotate it by @RequestMapping for the URL 'searchBooks.htm'. We can either use the Servlet API or Spring API, but we will use Spring APIs here.
12. Let's now add the code to searchBookByAuthor() for:
  - \* Reading request parameters
  - \* Search list of books
13. Creating instance of ModelAndView to bind book list as model data,

logical model name and logical view name together.

The code will be as shown below:

```
@Controller
public class SearchBookController {
    @RequestMapping(value = "searchBooks.htm")
    public ModelAndView searchBookByAuthor(
        @RequestParam("author") String author_name)
    {
        // the elements to list generated below will be added by
        business logic
        List<Book> books = new ArrayList<Book>();
        books.add(new Book("Learning Modular Java Programming",
            9781235, "packt pub publication", 800,
            "Explore the Power of Modular Programming ",
            "T.M.Jog"));
        books.add(new Book("Learning Modular Java Programming",
            9781235, "packt pub publication", 800,
            "Explore the Power of Modular Programming ",
            "T.M.Jog"));
        mv.addObject("auth_name", author);
        return new ModelAndView("display", "book_list", books);
    }
}
```

The `@RequestParam` is used to read a request parameter and bind it to the method argument. The value of the 'author' attribute is bounded to `author_name` argument without exposing servlet APIs.

Here, we have added a dummy list. Later on it can be replaced by the actual code to get the data from persistency layer.

14. It's time to configure ViewResolver and package scanning in books - servlet.xml as we did in earlier application. We can copy paste books-servlet.xml in WEB-INF from earlier application.
15. Create jsps folder under WebContent which will contain the jsp pages.
16. Create display.jsp in jsps folder to display the list of book using JSTL tag as shown below:

```
<%@ taglib prefix="jstl"
    uri="http://java.sun.com/jsp/jstl/core"%>
<html>
```

```

<head>
    <meta http-equiv="Content-Type" content="text/html;
        charset=ISO-8859-1">
    <title>Book List</title>
</head>
<body>
    <center>
        
        <br>
    </center>
    <jstl:if test="${not empty book_list }">
        <h1 align="center">
            <font color="green">
                Book List of ${auth_name }
            </font>
        </h1>
        <table align="center" border="1">
        <tr>
            <th>Book Name</th>
            <th>ISBN</th>
            <th>Publication</th>
            <th>Price</th>
            <th>Description</th>
        </tr>
        <jstl:forEach var="book_data"
            items="${book_list}" varStatus="st">
        <tr>
            <td>
                <jstl:out value="${ book_data.bookName }">
                </jstl:out>
            </td>
            <td>
                <jstl:out value="${ book_data.ISBN }">
                </jstl:out>
            </td>
            <td>
                <jstl:out value="${ book_data.publication }">
                </jstl:out>
            </td>
            <td>
                <jstl:out value="${ book_data.price }">
                </jstl:out></td>
            <td>
                <jstl:out value="${ book_data.description }">
                </jstl:out>
            </td>
        </tr>
    </jstl:forEach>
</table>

```

```

        </jstl:forEach>
    </table>
</jstl:if>
<jstl:if test="${empty book_list }">
    <jstl:out value="No Book Found"></jstl:out>
</jstl:if>
</body>

```

If the list doesn't have elements, there is no point to display such list. jstl:if tag is used to take the decision whether to display the list or not and jstl:forEach is used to display the book information by iterating over the list.

- Run the application and click link on home page to get the form to enter author's name. If the author's name exists, on the submission of the form we will get list of books as shown below:

The screenshot illustrates the search process. It starts with a search form on the left, followed by a transition arrow pointing to a results table on the right.

**Search Form (Left):**

- URL: <http://localhost:8080/ReadMyBooks/searchByAuthor.jsp>
- Illustration: A cartoon illustration of three children sitting on a grassy hill, reading books.
- Text: "Add the Author name to search the book List"
- Form fields:
  - Author Name:
  - Search Books

**Results Table (Right):**

- URL: <http://localhost:8080/ReadMyBooks/searchBooks.htm?author=T.M.Jog>
- Illustration: The same cartoon illustration of children reading books.
- Text: "Book List"
- Table:
 

Book Name	ISBN	Publication	Price	Description
Learning Modular Java Programming	9781235	packt pub publication	800	explore the power of modular Programming
Learning Modular Java Programming	9781235	packt pub publication	800	explore the power of modular Programming

Here, we have used `@RequestParam` to bind the individual request parameters to the method arguments. But, if in case name of the request parameters are matching to the name of method arguments, no need to use the annotation. The update code can be written as shown below:

```

@RequestMapping(value = "searchBooks.htm")
public ModelAndView searchBookByAuthor( String author) {
    List<Book> books = new ArrayList<Book>();
    books.add(new Book("Learning Modular Java Programming",
                      9781235, "packt pub publication", 800,
                      "explore the power of modular Programming ",
                      author));
    books.add(new Book("Learning Modular Java Programming",

```

```

    9781235, "packt pub publication", 800,
    "explore the power of modular Programming",
    author));
ModelAndView mv=
    new ModelAndView("display", "book_list", books);
mv.addObject("auth_name",author);
return mv;
}

```

Reading individual request parameters one by one and then to bind them to the object of bean becomes clumsy and unnecessarily lengthy. A better option has been provided by the framework by handling 'form backing object'.

## **Case 2: Handling Form submission**

The form submission is very common task in the application development. Each time on form submission the developer needs to perform the following steps:

1. Reading request parameters
2. Converting the request parameter values according to required data types
3. Setting the values to an object of the bean.

The above steps can be bypassed to get directly an instance of the bean on the form submission. We will discuss the form handling in two cases as:

- Form submission
- Form Preprocessing

### **Form Submission**

In normal web application the form will be loaded on clicking a link by the user and then manually the above discussed steps will be carried out. As the process needs to be automated instead of displaying the form directly, it should be loaded from the controller which already will have an instance of a bean. On the form submission, the values entered by the user will be bounded

to this instance. This instance now can be used in controller to carry out business logic. Spring 2.0 onwards provides the set of tags which are aware data binding in form handling in the view making the development easy.

Let's add a form to ReadMyBooks application to understand form submission using Spring provided form tags. We will do this in two steps one displaying the form and second post processing the submitted form.

### Displaying the form

As the form has to be loaded from the controller let's add the code using following steps,

1. Add a link on home page to get the form loaded. The code to get the form is as shown below:

```
<a href="showBookForm.htm">Show Form to add new Book</a>
```

2. Add `showBookForm()` method in `AddBookController` which will be invoked on clicking the link written in step1. The method will return a form page facilitating use of an object of `Book` where entered data will be bounded. The method has the below code,

```
@RequestMapping("/showBookForm.htm")
public ModelAndView showBookForm(ModelMap map)
throws Exception {
    Book book=new Book();
    map.addAttribute(book);
    return new ModelAndView("bookForm");
}
```

The method should have `ModelMap` as one of its argument to add an instance of bean which can be used by the views. Here we had added 'book' attribute having value as an instance of `book`. By default the reference name will be used as an attribute name. The 'book' instance can be referred as 'form backing' object as well. In order to customize name of form backing object to be used in the view we can use the following code:

```
map.addAttribute("myBook",book);
```

3. As the view name 'bookForm' has been returned by the controller, add bookForm.jsp in the jsps folder which has the form to display.
4. The values entered by the user needs to be bound to the form. Spring framework provides powerful tags to handle user input. To use Spring tags we need to add 'taglib' directive as shown below:

```
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
```

5. The Spring provides similar tags as that of html to handle form, input, checkbox, button and many more with a major difference of implicit binding of their values to the bean data member. The following code will allow user to enter the book name and on submission of form bind it to 'bookName' data member of the Book bean:

```
<form:input path="bookName" size="30" />
```

The 'path' attribute maps the entered value to the bean data member. The values have to be specified as per the names of the data members.

6. Let's add the form in bookForm.jsp as shown below to facilitate user to enter the values of new book:

```
<form:form modelAttribute="book" method="POST"
action="addBook.htm">
<h2>
    <center>Enter the Book Details</center>
</h2>

<table width="100%" height="150" align="center" border="0
<tr>
    <td width="50%" align="right">Name of the Book</td>
    <td width="50%" align="left">
        <form:input path="bookName" size="30" />
    </td>
</tr>
<tr>
    <td width="50%" align="right">ISBN number</td>
    <td width="50%" align="left">
        <form:input path="ISBN" size="30" />
    </td>
</tr>
<tr>
```

```

<td width="50%" align="right">Name of the Author</td>
<td width="50%" align="left">
    <form:input path="author" size="30" />
</td>
</tr>
<tr>
    <td width="50%" align="right">Price of the Book</td>
    <td width="50%" align="left">
        <form:select path="price">
            <!--
                We will add the code to have
                predefined values here
            -->
        </form:select>
    </td>
</tr>
<tr>
    <td width="50%" align="right">Description of the
        Book</td>
    <td width="50%" align="left">
        <form:input path="description" size="30" />
    </td>
</tr>
<tr>
    <td width="50%" align="right">Publication of the
        Book</td>
    <td width="50%" align="left">
        <form:input path="publication" size="30" />
    </td>
</tr>
<tr>
    <td colspan="2" align="center"><input type="submit"
        value="Add Book"></td>
</tr>
</table>
</form:form>

```

The attribute 'modelAttribute' takes the value of the logical name of the attribute of ModelMap set by the controller.

7. Run the application and click '**Show Form to add new book**'.
8. You will be navigated to the bookForm.jsp page where you can enter your own values. On submission, you will get 404 error as no resource has been written by us to handle the request. Don't worry!! In following steps we will process the form.

## Post Processing the form

1. Let's add a method in AddController which will be invoked on form submission for the url 'addBook.htm' as shown below:

```
@RequestMapping("/addBook.htm")
public ModelAndView addBook(@ModelAttribute("book") Book bo
throws Exception {
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("display");
    //later on the list will be fetched from the table
    List<Book>books=new ArrayList();
    books.add(book);
    modelAndView.addObject("book_list",books);
    return modelAndView;
}
```

When user submits the form, the values entered by him will get bounded to the bean data members giving an instance of the Book bean. Annotating the 'book' argument by `@ModelAttribute` facilitates developers to use bean instance who has values bounded to it. Now, there is no need to read the individual parameter and further things to get and set an instance of Book.

As we already have display.jsp page to display the books so we are just reusing it here. The book details entered by the user can later be added to the book table.

2. Run the application, click the link to get the form. Fill the form and submit it to get following output:

The image consists of two side-by-side screenshots of a web application. The left screenshot shows a page titled 'Book Shelf' with a cartoon illustration of three children reading books. The title 'Enter the Book Details' is displayed. The form contains fields for Name of the Book (Spring 5.0), ISBN number (1234), Name of the Author (T.M.Jog), Price of the Book (dropdown menu), Description of the Book (explore Spring 5.0), and Publication of the Book (Packt Pub). A blue 'Add Book' button is at the bottom. An arrow points from this button to the right screenshot. The right screenshot shows a page titled 'Book List of T.M.Jog' with a similar cartoon illustration. It displays a table with one row of data:

Book Name	ISBN	Publication	Price	Description
Spring 5.0	1234	Packt Pub	0	explore Spring 5.0

The output list shows the book details but without price as in the output. The price is having no value simply because we haven't set them. We want the pricelist with some predefined values. Let's move ahead to discuss pre-processing of the form.

## Form Pre processing

In some situations the form contains few predefined values like name of countries or categories of book in drop down menu, radio buttons with colors available to choose and many more. These values can be hard coded leading to the frequent change as values to show changes. Instead of using constant values, the values can be rendered and can be populated in the form. This generally called as form pre processing. The preprocessing can be done in two steps.

### Defining the attribute with values to add in the view

The `@ModelAttribute` is used to add an instance of the model data to an instance of Model. Each method annotated by `@ModelAttribute` will be

called before any other method of the Controller and on its execution the model data will be added to Spring Model. The syntax of using the annotation is as follows:

```
@ModelAttribute("name_of_the_attribute")
accessSpecifier returnType name_of_method(argument_list) { //
```

The following code adds the 'hobbies' attribute which can be used in the view:

```
@ModelAttribute("hobbies")
public List<Hobby> addAttribute() {
    List<Hobby> hobbies=new ArrayList<Hobby>();
    hobbies.add(new Hobby("reading",1));
    hobbies.add(new Hobby("swimming",2));
    hobbies.add(new Hobby("dancing",3));
    hobbies.add(new Hobby("painting",4));
    return hobbies;
}
```

Hobby is user defined class with hobbyName and hobbyId as data members.

### **Populating the values of the attribute in the form**

The form can display the available list of the choices to the user using checkboxes, drop down menus or radio buttons. The values in the view can be populated using List, Map or array for the values of the drop down menu, check boxes or radio buttons .

The general syntax of the tags is as shown below:

```
<form:name-of_tag path="name_of_data_memeber_of_beans"
    items="name_of_attribute" itemLabel="value_to_display"
    itemValue="value_it_holds">
```

The following code can be used to display hobbies of the user in checkboxes using 'hobbies' as modelattribute for binding the value to the hobby data member of the bean:

```
<form:checkboxes path="hobby" items="${hobbies}"
    itemLabel="hobbyName" itemValue="hobbyId"/>
```

In the same way we can generate dropdown menu and options for the select tag at runtime.

## Note

The `itemLabel` and `itemValue` attribute can be skipped while dealing while handling String values.

The complete example can be referred from the application Ch06\_Form\_PrePopulation.

Let's update `ReadMyBooks` application to predefine some values of price in the `bookForm.jsp` using '`ModelAttribute`' to discuss form pre- processing with the help of following steps:

1. As the form is retuned by the `AddController` to the Front Controller where we want to set predefined values, ass `addPrices()` method in it. Annotate the method by `@ModelAttribute` as shown below:

```
@ModelAttribute("priceList")
public List<Integer> addPrices() {
    List<Integer> prices=new ArrayList<Integer>();
    prices.add(300);
    prices.add(350);
    prices.add(400);
    prices.add(500);
    prices.add(550);
    prices.add(600);
    return prices;
}
```

The above code is creating an attribute '`pricelist`' which can be available to the view for use.

2. Now the `pricelist` attribute can be used in the view to display the predefined values. In our case it's a form for addition of the new book, update the `bookForm.jsp` to display pricelist as shown below:

```
<form:select path="price">
<form:options items="${priceList}" />
```

```
</form:select>
```

- Run the application and click the link, you can observe the predefined prices will appear in the drop sown list as shown below:

http://localhost:8080/ReadMyBooks/showBookForm.htm

**Book Shelf**

### Enter the Book Details

Name of the Book

ISBN number

Name of the Author

Price of the Book

Description of the Book

Publication of the Book

300  
350  
400  
500  
550  
600

The users will enter values in the form and submit it.

The values can be obtained in the handler method. But still, we can't be sure of only valid values will be entered and submitted. The business logic performed on wrong values will always fail. Also there is a chance of getting wrong data type values entered by the user leading to exception. Let's take an example of email ids. The email id always follows a particular format, if the format is wrong the business logic ultimately fails. Whatever may be the case, we have to be sure of submitting only valid values either for their data types, range or formation. To validate whether the correct data will be submitted or not is the process of 'form validation'. The form validation plays key role to make sure the correct data submission. The form validation can be

done on client side as well as server side. The Java Script is used to perform client side validations, but it's possible to disable it. In such cases server side validation is always preferable.

---

Spring has flexible validation mechanism which can be extensible to write custom validators as per application requirements. Spring MVC framework by default supports JSR 303 specification on addition of JSR303 implementation dependencies in the application. The following two approaches can be used to validate the form fields in Spring MVC,

- JSR 303 specification based validation
- Spring based implementation using Validator interface.

## Custom validator based on Spring Validator interface

Spring provides Validator interface who has validate method where the validation rules will be checked. The interface not only supports validation of the web tier but it can also be used in any tier to validate the data. In case the validation rules fail, the user has to be made aware of it by showing appropriate informative messages. The BindingResult, a child of an Errors holds the validation result bounded by the Errors while performing validation on the model in validate() method. The bounded messages for the errors will be displayed using<form:errors> tag in the view to make the user aware of them.

Let's add a custom validator in our ReadMyBooks application with the help of following steps:

1. Add validation-api-1.1.0.final.api.jar file in the lib folder of the application.
2. Create BookValidator class in com.packt.ch06.validators package.
3. The class implements org.springframework.validation.Validator interface.
4. Override supports () method as shown in the code below,

```
public class BookValidator implements Validator {  
    public boolean supports(Class<?> book_class) {  
        return book_class.equals(Book.class);  
    }  
}
```

The support method assures the object is matching to the object being validated by the validate method

5. Now override the validate() method where the check on the data member as per the rule. We will do it in three steps as:
  1. Set the rules for validation

We will crosscheck the following rules:

- The length of book's name must be greater than 5.
  - The author's name must not be empty.
  - The description must not be empty.
  - The description length must be of minimum 10 and maximum 40 characters.
  - The ISBN should not be less than 150.
  - The price should not be less than 0.
  - The publication must not be empty.
2. Write condition to check validation rules.
  3. If validation fails add the message to an instance of errors using rejectValue () method

The method using the above steps can be written as shown below:

```
public void validate(Object obj, Errors errors) {  
    // TODO Auto-generated method stub  
    Book book=(Book) obj;  
    if (book.getBookName().length() < 5) {  
        errors.rejectValue("bookName", "bookName.required",  
        "Please Enter the book Name");  
    }  
    if (book.getAuthor().length() <=0) {  
        errors.rejectValue("author", "authorName.required",  
        "Please Enter Author's Name");  
    }  
    if (book.getDescription().length() <= 0)  
    {  
        errors.rejectValue("description",  
        "description.required",  
        "Please enter book description");  
    }  
    else if (book.getDescription().length() < 10 ||  
    book.getDescription().length() < 40) {
```

```

        errors.rejectValue("description", "description.length
            Please enter description within 40 charaters only");
    }
    if (book.getISBN()<=1501) {
        errors.rejectValue("ISBN", "ISBN.required",
            "Please Enter Correct ISBN number");
    }
    if (book.getPrice()<=0 ) {
        errors.rejectValue("price", "price.incorrect", "Pleas
            enter a Correct correct price");
    }
    if (book.getPublication().length() <=0) {
        errors.rejectValue("publication",
            "publication.required",
            "Please enter publication ");
    }
}
}

```

The Errors interface is used to store the binding information about the validation of the data. The `errors.rejectValue()` is one of the very useful method provided by it which registers the errors for an object along with their error messages. Following are the available signatures of `rejectValue()` method from Error interface,

```

void rejectValue(String field_name, String error_code);
void rejectValue(String field_name, String error_code, Stri
    default_message);
void rejectValue(String field_name, String error_code,
    Object[] error_Args, String default_message);

```

4. Add a data member of type `org.springframework.validation.Validator` in `AddBookController` and annotate it by `@Autowired` as shown below:

```

@.Autowired
Validator validator;

```

5. Update the `addBook()` method of `AddController` to invoke `validate` method and check whether validation error occurred or not . The updated code is as shown below:

```

public ModelAndView addBook(@ModelAttribute("book") Book bo
    BindingResult bindingResult) throws Exception {
    validator.validate(book, bindingResult);
    if(bindingResult.hasErrors())

```

```

{
    return new ModelAndView("bookForm");
}
ModelAndView modelAndView = new ModelAndView();
modelAndView.setViewName("display");
//later on the list will be fetched from the table
List<Book>books=new ArrayList();
books.add(book);
modelAndView.addObject("book_list",books);
modelAndView.addObject("auth_name",book.getAuthor());
return modelAndView;
}

```

The method signature of `addBook()` should have `BindingResult` as one of its argument. The instance of `BindingResult` has the list of errors which has occurred while performing the validation. The `hasErrors()` method returns true if validation has failed on the data members. If `hasErrors()` returns true, we are returning the '`bookForm`' view facilitating user to enter the correct values. In case of no validation violation the '`display`' view will be returned to the Front Controller.

6. Register the `BookValidator` as a bean in `books-servlet.xml` as shown below:

```
<bean id="validator"
      class="com.packt.ch06.validators.BookValidator" />
```

You can also use `@Component` instead of the above configuration.

7. The Validation violation messages have to be shown to the user by updating the `bookForm.jsp` as shown in code below:

```

<tr>
    <td width="50%" align="right">Name of the Book</td>
    <td width="50%" align="left">
        <form:input path="bookName" size="30" />
        <font color="red">
            <form:errors path="bookName" />
        </font>
    </td>
</tr>

```

Only the underline code has to be added in the `bookForm.jsp` to show the

message in Red color.

The <form:errors> is used to display the messages if validation failed. It takes the below shown syntax:

```
<form:errors path="name of the data_member" />
```

8. Update the bookForm.jsp for all the inputs by specifying name of data members as value for path attribute.
9. Run the application. Click on the link Show form to add new Book.
10. Without entering any data in the text fields submit the form. We will get the form displaying the messages which denotes which validation rules has been violated as below:



## Enter the Book Details

Name of the Book	<input type="text"/>	Please Enter the book Name
ISBN number	<input type="text" value="0"/>	Please Enter Correct ISBN number
Name of the Author	<input type="text"/>	Please Enter Author's Name
Price of the Book	<input type="text" value="300"/> <input type="button" value="▼"/>	
Description of the Book	<input type="text"/>	Please enter book description Please enter description within 40 charaters only
Publication of the Book	<input type="text"/>	Please enter publication
<input type="button" value="Add Book"/>		

The code written for validation above though is working fine, we are not taking complete advantage of Spring framework. The invocation of the validate method is explicit as the framework is not aware to carry out validation implicitly. The @Valid annotation provides information to the framework to perform validation implicitly using custom validators. The framework facilitates binding of the custom validator to WebDataBinder

giving awareness to framework to use the validate() method.

## Using @InitBinder and @Valid for validation

Let's update the code of AddController.java step by step as shown below:

1. Add a method to bind validator to WebDataBinder and annotate it by `@InitBinder` as shown below:

```
@InitBinder  
private void initBinder(WebDataBinder webDataBinder)  
{  
    webDataBinder.setValidatorvalidator);  
}
```

`@InitBinder` annotation helps in identifying the methods which performs the WebDataBinder initialization.

2. To enable the annotations to be considered by framework the book-servlet.xml has to be updated as,
3. Add mvc namespace as shown in the configuration below:

```
<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"  
       xmlns:mvc="http://www.springframework.org/schema/mvc"  
       xsi:schemaLocation="http://www.springframework.org/schema/  
                           http://www.springframework.org/schema/beans/spring-bean  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/  
                           spring-context.xsd  
                           http://www.springframework.org/schema/mvc  
                           http://www.springframework.org/schema/mvc/spring-mvc.xs
```

You can only copy the underlined statements in your existing code.

4. Add the configuration as shown below:

```
<mvc:annotation-driven/>
```

5. Update the `addBook()` method to add `@Valid` annotation to perform Book validation and remove the `validator.validate()` invocation as it will be

executed implicitly. The updated code is as shown in the below:

```
@RequestMapping("/addBook.htm")
public ModelAndView addBook(@Valid @ModelAttribute("book")
Book book,BindingResult bindingResult)
throws Exception {
    //validator.validate(book, bindingResult);
    if(bindingResult.hasErrors())
    {
        return new ModelAndView("bookForm");
    }
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.setViewName("display");
    //later on the list will be fetched from the table
    // rest of the code is same as the earlier implemenation
}
```

6. Run the application to get the similar result when you submit blank form. The messages will be displayed in the view has been hard coded in the `rejectValue()` method. The framework provides a support for externalizing the messages in the properties file. Let's update the validator for externalizing messages.

### **Externalizing the messages**

We will use the externalization of the messages without changing the logic of validation with the help of following steps:

1. Add a new class `BookValidator1` in `com.packt.ch06.validators` package implementing `Validator` interface.
2. Override the `supports` method as did in earlier application.
3. Override the `validate` method where we are not providing the default error messages. We will only provide name of the bean property and error code associated with it as shown below:

```
public void validate(Object obj, Errors errors) {
    Book book=(Book) obj;
    if (book.getBookName().length() < 5) {
        errors.rejectValue("bookName", "bookName.required");
    }

    if (book.getAuthor().length() <=0) {
```

```

        errors.rejectValue("author", "authorName.required");
    }

    if (book.getDescription().length() <= 0) {
        errors.rejectValue("description", "description.required");

    if (book.getDescription().length() < 10 || book.getDescription().length() < 40) {
        errors.rejectValue("description", "description.length")

    if (book.getISBN()<=1501) {
        errors.rejectValue("ISBN", "ISBN.required");
    }

    if (book.getPrice()<=0 ) {
        errors.rejectValue("price", "price.incorrect");
    }

    if (book.getPublication().length() <=0) {
        errors.rejectValue("publication", "publication.required"
    }
}

```

- Let's add messages\_book\_validation.properties file in WEB-INF to map error code to their associated message as shown below:

```

bookName.required=Please enter book name
authorName.required=Please enter name of the author
publication.required=Please enter publication
description.required=Please enter description
description.length=Please enter description of minimum 10 a ISBN.required=Please enter ISBN code
price.incorrect= Please enter correct price

```

The syntax to write properties file to map key-value pair is,

```

name_of_Validation_Class . name_of_model_to_validate
.name_of_data_memeber = message_to_display

```

- Update the books-servlet.xml as,
- Comment the bean written for BookValidator as we are no using it
- Add a new bean for BookValidator1 as shown below:

```

<bean id="validator"
      class="com.packt.ch06.validators.BookValidator1" />

```

8. Add a bean for MessageSource to load the messages from the properties file as:

```
<bean id="messageSource"
      class="org.springframework.context.support.
      ReloadableResourceBundleMessageSource">
    <property name="basename"
              value="/WEB-INF/messages_book_validation" />
</bean>
```

9. No need to change the AddController.java. Run the application, on submission of the blank from the messages pulled from properties file will be displayed.

We successfully externalize the messages, Congratualtion!!!

But don't you think the validation code is unnecessarily performing the basic validations here. The framework provides ValidationUtils as a utility class which facilitate the developers carrying out the basic validations like empty or null values.

### Using ValidationUtils

Let's add BookValidator2 which will use ValidationUtils as follows:

1. Add BookValidator2 as a class in com.packt.ch06.validators package which is implementing Validator in ReadMyBooks application.
2. Override supports() method as did earlier.
3. Override validate() which will perform validation using ValidationUtils class as shown below:

```
public void validate(Object obj, Errors errors) {
    Book book = (Book) obj;
    ValidationUtils.rejectIfEmptyOrWhitespace(errors,
        "bookName", "bookName.required");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "author",
        "authorName.required");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors,
        "description", "description.required");
    if (book.getDescription().length() < 10 ||
        book.getDescription().length() < 40) {
```

```

        errors.rejectValue("description", "description.length",
            "Please enter description within 40 charaters only");
    }
    if (book.getISBN() <= 1501) {
        errors.rejectValue("ISBN", "ISBN.required", "Please
            Enter Correct ISBN number");
    }
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "price"
        "price.incorrect");
    ValidationUtils.rejectIfEmptyOrWhitespace(errors,
        "publication", "publication.required");
}

```

4. As we are reusing the same error codes no need to add them again in the properties file.
5. Comment the bean for BookVlidator1 and add a bean for BookVlidator2 in books-servlet.xml as shown below:

```
<bean id="validator"
    class="com.packt.ch06.validators.BookValidator2" />
```

6. Execute the application and submit the blank form to get validation messages displayed from the properties file.

## JSR annotation based validation

JSR 303 is a bean specification which defines the metadata and APIs to validate the beans in J2EE applications. The latest in market is JSR 349 which is an extension of JSR 303 providing features as openness, Denepndency injection and CDI, method validation, group conversion, integrations with other specification. Hibernate Validator is a well known reference implementation available. The javax.validation.\* package provides the APIs for validation purpose.

Following are few common annotations used in validation:

- **@NotNull:** Checks the annotated value is not null but it cannot check for empty strings.
- **@Null:** It Checks the annotated value is null
- **@Pattern:** It checks whether the annotated string matches the given

regular expression.

- `@Past`: Checks the annotated value is the date in past.
- `@Future`: Checks the annotated value is the date in future.
- `@Min`: It makes sure that the annotated element is a number and whose value is equal or greater than the specified value.
- `@Max`: It makes sure that the annotated element is a number and whose value is equal or less than the specified value.
- `@AssertFalse`: It assures the annotated element is false.
- `@AssertTrue`: It assures the annotated element is true.
- `@Size`: It assures the annotated element is between the max and min values.

Apart from the above annotations defined by Bean Validation API the following additional annotations has been provided by Hibernate Validator:

- `@CreditCardNumber`: It checks that the annotated value follows the character sequence passed to it.
- `@Email`: Used to check the specified character follows valid email address according to the specified expression
- `@Length`: It checks that the annotated element has number of characters limited by min and max attribute specified by the annotation.
- `@NotBlank`: It checks for the annotated element is not null and has the length greater than zero.
- `@NotEmpty`: It makes sure the annotated element is neither null nor empty.

Let's create a copy of ReadMyBooks application to implement JSR based validation by following steps:

## Part1: Creating basic application

1. Create ReadMyBooks\_JSR\_Validation as a dynamic web application.
2. Add all required jar which we had added in earlier application.
3. Along with these jar also add hibernate-validator-5.0.1.final.jar, classmate-0.5.4.jar, jboss-logging-3.1.0.GA.jar and validation-api-1.1.0.final.jar
4. Copy com.packt.ch06.beans and com.packt.ch06.controllers package

along with its content.

5. Copy index.jsp and searchByAuthor.jsp in WebContent.
6. Add DispatcherServlet mapping in web.xml file.
7. Copy books-servlet.xml in WEB-INF
8. Copy images in WebContent and jsps folder in WEB-INF along with its content.

## Part2: Applying validation

1. Let's apply validations provided by hibernate-validator API on Book.java as shown below:

```
public class Book {  
    @NotEmpty  
    @Size(min = 2, max = 30)  
    private String bookName;  
  
    @Min(150)  
    private long ISBN;  
  
    @NotEmpty  
    @Size(min = 2, max = 30)  
    private String publication;  
  
    @NotNull  
    private int price;  
  
    @NotEmpty  
    @Size(min = 10, max = 50)  
    private String description;  
  
    @NotEmpty  
    private String author;  
  
    //default and parameterized constructor  
    //getters and setters  
}
```

2. Let's update AddBookController as,
3. Delete the Validator data member.
4. Delete initBinderMethod.
5. Keep @Valid annotation applied on Book argument of addBook()

method as it is.

6. Remove bean for validator from books-servlet.xml as its no longer required.
7. Comment bean for messageResource from XML we will use it later on.
8. Make sure to have `<mvc:annotation-driven />` entry in book-servlet.xml to enable framework to consider annotation in controllers.
9. Run the application. On submission of blank form you will get the following response displaying the default validation messages:



## Enter the Book Details

Name of the Book	<input type="text"/>	may not be empty
ISBN number	<input type="text" value="0"/>	must be greater than or equal to 150
Name of the Author	<input type="text"/>	may not be empty
Price of the Book	<input type="text" value="300"/> <input type="button" value="▼"/>	
Description of the Book	<input type="text"/>	size must be between 10 and 50
Publication of the Book	<input type="text"/>	size must be between 2 and 30
<input type="button" value="Add Book"/>		

The customization of the messages can be done either by using the 'message' attribute or we can externalize the messages using properties file. Let's do one by one.

### Using 'message' attribute

Each of the annotations used in bean class to validate the data has 'message' attribute. The developers can use it to pass the appropriate message as shown in the code below:

```

public class Book {
    @NotEmpty(message="The book name should be entered")
    private String bookName;

    @Min(value=150,message="ISBN should be greater than 150")
    private long ISBN;

    @Size(min = 2, max = 30, message="Enter Publication between
        limit of 2 to 30 only")
    private String publication;

    @NotNull(message="Enter the price")
    private int price;
    @Size(min = 10, max = 50,message="Enter Publication between lim
        10 to 50 only")
    private String description;

    @NotEmpty(message="Enter the price")
    private String author;
    /default and parameterized constructor
    //getters and setters
}

```

Keeping all other code as it is and changing the Book.java as shown above run the application. If any violation of the validation rule occurred, the messages configured for 'message' attribute will be displayed.

### **Using properties file**

The developers can externalize the messages in the properties file from where it will be loaded on the violation of the validation as we did in the earlier application.

Let's add properties file using the below steps in the application:

1. Create a file messages\_book\_validation.properties in WEB-INF and add to mapping of violation rules and messages to display as shown below:

```

NotEmpty.book.bookName=Please enter the book name F1.
NotEmpty.book.author=Please enter book author F1.
Min.book.ISBN= Entered ISBN must be greater than 150 F1
Size.book.description=Please enter book description having
minimum 2 and maximum 30charatcters only F1.

```

NotNull.book.price=Please enter book price F1.

F1 at the end of each file has been purposely added to know whether the messages are pulled from bean class or properties file. You don't have to add them in actual file. Purposely we haven't added any message for 'publication' data member to understand pulling of messages.

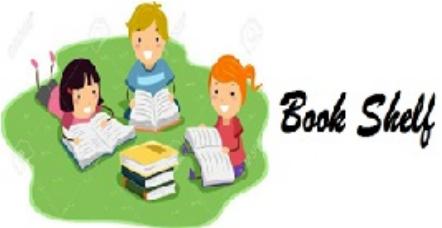
The syntax to write the properties file is as follows:

```
Name_of_validator_class.name_of_model_attribute_to_validate  
name_of_data_member= message_to_display
```

2. Uncomment the bean 'messageResource' in book-servlet.xml or add one if you don't have as shown below:

```
<bean id="messageSource"  
      class="org.springframework.context.support.  
      ReloadableResourceBundleMessageSource">  
  <property name="basename"  
    value="/WEB-INF/messages_book_validation" />  
</bean>
```

3. Run the application, on submission of the blank form the messages from the properties file will be loaded apart from 'publication' as shown below:



## Enter the Book Details

Name of the Book	<input type="text"/>	Please enter the book name F1.
ISBN number	<input type="text" value="0"/>	Entered ISBN must be greater than 150 F1.
Name of the Author	<input type="text"/>	Please enter book author F1.
Price of the Book	<input type="text" value="300"/> <input type="button" value="▼"/>	
Description of the Book	<input type="text"/> 30charatcters only F1.	Please enter book description having minimum 2 and maximum 30charatcters only F1.
Publication of the Book	<input type="text"/>	Enter Publication between limit of 2 to 30 only <input type="button" value="▼"/>
<input type="button" value="Add Book"/>		<b>Message pulled from bean class's message attribute</b>

# Summary

---

We had a discussion about web tier in this application. We discussed about the working of Spring MVC framework declaring user defined Controller. We had a discussion about the views are used to display the values using model object from ModelAndView. We also discussed about how the view are discovered by the framework and how they are rendered from the logical name set in ModelAndView using ViewResolvers. The discussion continued to the form handling where we had in depth discussion about form submission to use form backing object and pre population of the form using @ModelAttribute annotation. The from containing incorrect values may leads to exceptions or failure in business logic. The solution to the problem is form validation. we discussed form validation with the help of Spring custom validators and annotation based validation provided by hibernate validators. We also had discovered how externalization of messages to carry out using messageresource bundle. In the next chapter we will continue our discussion about how to carry out testing of an application to minimize the risk of its failure when the application goes live.

# **Chapter 7. Be assured take a test drive**

The application development is a lengthier, time consuming and expensive process. The development is depending upon the requirements collected from the clients and market requirements. But what if after the completion of the work, something goes wrong and everything gets collapsed. The collision is not because of the incorrect solution, but because it's based on wrong assumptions which developers assumed before the start of the work. This collision occurred just before the date of delivery to the client. Now nothing can be recovered! Let's not go into the details of why and what went wrong. But I am interested in, can this be avoided? Is there something which can be done to prevent this last moment collision? We always heard 'prevention is better than cure'. This phrase is applicable to application development as well. The situation of failure can be avoided with bit of extra efforts taken by the developers step by step. The cross checking of the code developed is according to the requirements helps developers to be assured of correct working of the code. This cross checking is called as testing of the application. In this chapter, we will discuss in depth about the testing with the help of following points:

- Why testing?
- Problems in testing of Spring Controllers.
- Mock testing.
- Spring TestContext Framework.
- Using Mockito to test Spring Controller.
- Introduction to Spring Controller testing using Arquillian

# 'Testing' an important step

---

The application development is an expensive and time consuming process. The errors and mistakes occurred at the final deployment leads to very serious consequences. The coding is done by the developers according to the requirements, is based on the rules which may be based on few assumptions. Being a human we may make mistakes in either collection of requirement or making up the assumptions. If this is the work done by us, who can better understand it than us? The unit testing tests the code and helps in assuring it is working.

The developers did development. Their development was based on some assumption and they may leave out few blind spots as well. The development is followed by the testing by them. It's a high risk to carry out the tests by the same person as they may repeat the same mistakes. Ideally someone else should do the check assuring they know what they are testing.

Following are the few major factors which makes the testing as one of the unforgettable part in application development,

- It helps in early detection of the defect and errors which has been done while development.
- It assures least failures in application executions
- It helps in improving the consistency of an application
- It helps in assuring better application quality
- It helps in improving the security by checking authentication and authorization
- Helps in saving money and more importantly time

Each application undergoes rigorous testing before it is released to ensure the application is matching the requirements and the correctness of all its functionalities. Unit testing, Integration testing, System testing and Acceptance testing are four major stages through which each application needs to pass.

## **Unit Testing**

Unit testing focuses on the unit of the component ensuring the correctness of the function. The unit can be referred to an individual function or a procedure. The unit testing mainly aims to make sure the unit is working as per the design. It allows the raised issues to be resolved quickly. As the unit is the very smallest part of the application the code can be modified easily. It's generally done by the developer who had developed the code.

## **Integration Testing**

Once the unit testing is successfully carried out, most of the issues occurred while testing the unit has been changed to match the requirements. The integration testing gives the opportunity to test the group of these units within a program execution. It helps in determining how multiple units are running together. The unit may work fine, but the same unit when combined with other unit may leads to some side effects which need to be resolved. The integration test helps in catching such errors giving an opportunity to correct it.

## **System Testing**

In previous two stages the individual unit or interaction of two units with one another has been tested. This is the first stage where for the first time the complete application will be tested. The system testing is generally done by the independent tester and in a close to production environment. The system testing makes sure whether all functional and business requirements for which the application developed has been met or not.

## **User Acceptance Testing**

This is the final stage in testing which determines whether the system is ready for the final release. The acceptance test is generally carried out by the end users to determine the application is meeting the requirements and have covered all the necessary functionalities to give the final acceptance. It gives

the feel of final application in the production environment.

In this chapter we will carry out Unit testing, Integration testing and System Testing in three phases. But before moving ahead let's have an overview about the testing tools available in market.

# Testing Tools

---

Following are the available test tools for Java platform,

## JTest

JTest is automated software testing, coding standard compliance tool for Java platform developed by Parasoft since 1997. The tool leverages unit as well as integration testing. The tool facilitates analysing the classes, generation and execution of test cases in the same format as that of JUnit test cases.

Following are the features JTest:

- Along with testing, it covers and exposes the runtime exceptions which normally a developer doesn't catch.
- The tool also verifies if the class is following **Design by Contract(DbC)** basis.
- It ensures the code follows 400 standard rules of coding and checks the code against 200 violation rules.
- It also can identify the problems like functional errors, memory leakage, and security vulnerabilities.
- **Jcontract** is tools from JTest which verifies the functionality requirements during the integration testing without hampering the performance of the application.

## Grinder

Grinder is **load testing tool** for Java programming language available under the BSD-style open source licence. It aims to simplify the running of a distributed testing using load injector machines. It has the capabilities to do load testing, capability testing, functional testing and stress testing. It has minimum system resource requirements along with it manages its own thread in test context which can split over the process if required.

Following are the features Grinder:

- Easy to use Java Swing based user interface
- It can be used for load testing of anything which has Java API. It can be used for Web servers, web services based on SOAP and Rest API, application servers
- The Jython and Clojure languages supports in writing flexible, dynamic test scripts.
- It manages client connections as well as cookies

## **JWalk**

JWalk is one more unit testing tool for Java platform supporting the Lazy Systematic Unit testing paradigm. It has been developed by Anthony Simons. JWalk tests a single class and produces the test report by notions of 'lazy specification' and 'systematic testing'. It is more favourable for agile development where no formal specification needs to be produced. It saves lots of time and efforts by constructing and presenting the automatic test cases.

Following are the features of JWalk:

- Systematic proposal of all probable test cases.
- No need to confirm the sub set of the test outcome by tester.
- Can predict test outcomes.
- Generates new test case if the class has been modified.
- Suits for TDD for extreme programming in software development.

## **PowerMock**

PowerMock is open source created as an extension of the EasyMock and Mockito frameworks by adding few methods and annotations. It facilitates creation mock objects of the implementations from Java code. Sometimes the architecture of the application is designed in such a way that it uses final classes, private methods or static methods to design the classes. Such methods or classes cannot be tested as their mocks can't be created. The

developer has the choice to choose between a good design or testability. The PowerMock facilitates the mock creation of static methods and final classes by using custom classloader and bytecode manipulation.

## TestNG

TestNG a powerful testing framework inspired by JUnit and NUnit testing useful unit testing, functional testing, integration testing. It facilitates a parameterised testing which is not possible by JUnit. It is empowered with many useful annotations like before and after )every test method(@BeforeMethod, @AfterMethod) and before and after class(@BeforeClass, @AfterClass) to carry out pre or post data processing.

Following are the features of TestNG:

- Easy testcase writing
- It can generate HTML report
- It can generate logs
- Good integration test support

## Arquillian Framework

Arquillian is a testing framework for Java based applications. The framework facilitates the developers to deploy the application in the runtime environment to execute the test cases using JUnit and TestNG. The management of the runtime environment from within the test is made possible as Arquillian manages the following things in management of test the life cycle:

- It can manage more than one containers
- It bundles the classes, resources and the test cases using ShrinkWrap
- It deploys the archive to the container
- Executes the test case inside the container
- Returns the result to the test runner

## The ShrinkWrap

The framework comprises of three major components,

## Test Runners

To execute the test case JUnit or TestNG uses Arquillian test runner. This facilitates the use of component model in the test case. It also manages the container life cycle and dependency injections which make the model available to use.

## Java Container

Java containers are the major components of the test environment. Arquillian tests can be executed in any compatible container. The Arquillian selects the container to choose which container adapter is made available in the classpath. These container adapters controls and helps in communicating with the containers. Arquillian test cases even can be executed without JVM based container also. We can use the annotation `@RunsClientto` execute the test cases outside of the Java Container.

## Integration of test cases into the Java container

The framework has external dependency to use known as ShrinkWrap. It helps in defining the deployments and descriptors of the application to be loaded in the Java Container. The test cases run against these descriptors. Shrinkwrap supports to generate dynamic Java archive files of type JAR, WAR and EAR. It also can be used for addition of deployment descriptor as well as creation of the DD programmatically.

Arquillian can be suits to use in the following scenarios,

- The part of your application to test needs deployment of application within embedded server
- The test to be executed on hourly, after certain interval or when someone commits the code
- Automation of acceptance test of the application through external tools

# JUnit

JUnit is the most popular open source framework for Java in test driven development. The JUnit helps in unit testing of the component. It also widely support for tools such as ANT, Maven, Eclipse IDE. The unit test class is an ordinary class like any other classes with a major difference of use of `@Test` annotation. The `@Test` annotation lets the JUnit test runner that this annotated method needs to be executed to perform testing.

The class `org.junit.Assert` provides a series of static `assertXXX()` methods which performs the testing by comparing the actual output to the assumed output of the method under test. If the comparison of the test returns normally, it indicates the test has passed. But if the comparison fails the execution stops indicating test has failed.

The unit test class normally called as Unit Test Case. The Test case can have multiple methods which will be executed one after another in the order in which they have written. The JUnit facilitates setting up the test data for a public unit under testing and does the testing against it. The initialization of data can be done in `setUp()` method or in the method which has been annotated by `@Before` annotation. By default it uses JUnit runner to run the test case. But, it has Suite, Parameterised, and Categories as few more built in runners. Along with these runner JUnit also support third party runners like `SpringJUnit4ClassRunner`, `MokitoJUnitRunner`, `HierarchicalContextRunner`. It also facilitates the use of `@RunWith` which facilitates using the custom runners. We will discuss in depth about the annotation along with Spring testing framework shortly.

Following are few assertion methods which facilitates the testing using comparison,

- `assertEquals` : The method tests the equality of two objects with the help of `equals()` method.
- `assertTrue` and `assertFalse` : It is used to test boolean values against true or false conditions.
- `assertNull` and `assertNotNull` : The method tests the value to be either null or not null.

- `assertSame` and `assertNotSame` : It is used to test the two references passed as an argument point to the same object or not.
- `assertArrayEquals` : It is used to test the two array contains equal elements and each element from one of the array is equal to the element from other array with the same index.
- `assertThat` : It tests the object matches to an object of `org.harmcrest.Matcher` or not.

# Pase I Unit testingDAO Unit testing by JUnit

---

Now, it's time to write the actual test case. We will start unit testing DAO layer. Following are general steps to write an annotation based test case,

1. Create a class having the name of the class under test prefixed by 'Test'.
2. Write `setUp()` and `tearDown()` methods for initializing the data we needed and releasing the resources used by us respectively.
3. The methods where test will be conducted name them as name of the method under test prefixed by 'test'.
4. The method which the test runner should recognize need to be annotated by `@Test`
5. Use `assertXXX()` methods as per the data under test to compare the values.

Let's write the tests for the DAO layer which we developed in third chapter. We will use Ch03\_JdbcTemplates as base project. You can create the new project or can use Ch03\_JdbcTemplates by adding only test package. Let's follow the steps.

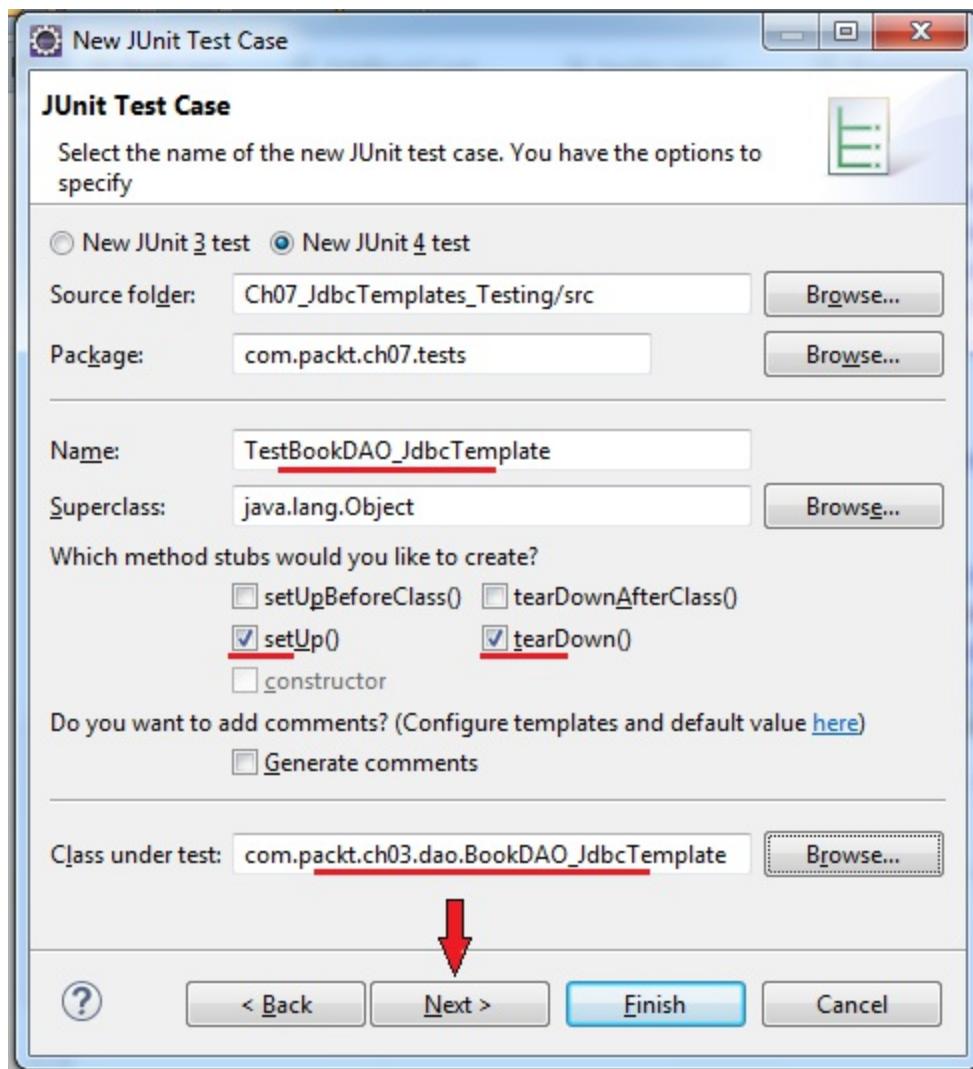
## Create base application.

1. Create Ch07\_JdbcTemplates\_Testing as Java project.
2. Add all the required jars for spring core, spring jdbc and JDBC which we already added for Ch03\_JdbcTemaplates project.
3. Copy com.packt.ch03.beans and com.packt.ch03.dao package from base project. We will only carry out testing for BookDAO\_JdbcTemplate class.
4. Copy connection\_new.xml in classpath

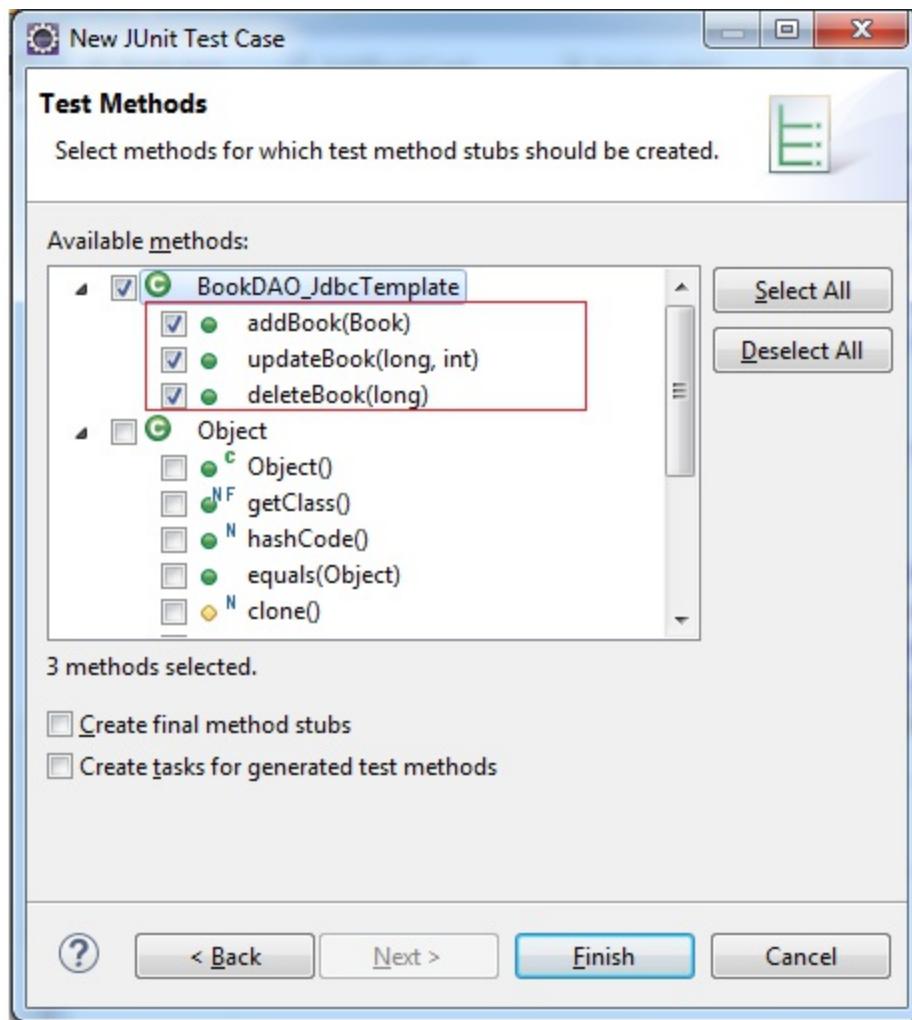
## Performing testing

1. Create com.packt.ch07.tests package
2. Use JUnit test case template from Eclipse IDE as:
  1. Enter the name of test case as TestBookDAO\_JdbcTemplate
  2. Select checkboxes for setUp and teardown for initializing and releasing the test case components.
  3. Click on browse button and select BookDAO\_JdbcTemplate as class under test
  4. Click on next button
  5. In the Test methods Dialogue box select all the methods from BookDAO\_JdbcTemplate class.
  6. Click on finish.
  7. A Dialogue will appear asking to add JUnit4 on build path. Click on Ok button.

The steps can be summarized as shown in the figure below:



8. After Clicking Next Button you will get the next dialogue:



9. In the test case declare a data member as `BookDAO_JdbcTemplate`.
10. Update the `setUp()` method to initialize the data member of test case using `ApplicationContext` container.
11. Update `tearDown()` to release the resources.
12. Update `testAddBook()` as,
13. Create an object of type `Book` with some values, make sure the value of the ISBN is not available in the Book table.
14. Invoke `addBook()` from `BookDAO_JdbcTemplate` class
15. Test the result using `assertEquals()` method as shown in the code below:

```
public class TestBookDAO_JdbcTemplate {  
    BookDAO_JdbcTemplate bookDAO_JdbcTemplate;  
  
    @Before
```

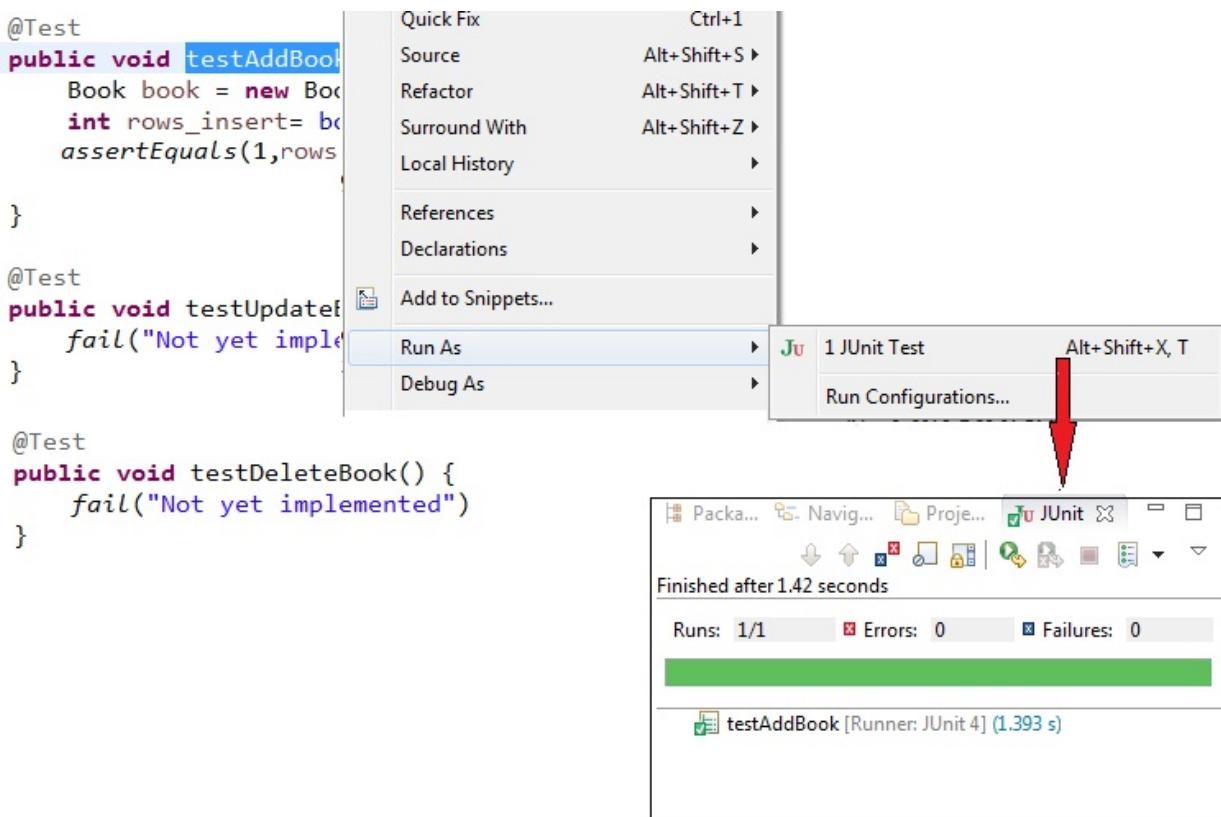
```

public void setUp() throws Exception {
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("connection_new.xml");
    bookDAO_JdbcTemplate = (BookDAO_JdbcTemplate)
        applicationContext.getBean("bookDAO_jdbcTemplate");
}
@After
public void tearDown() throws Exception {
    bookDAO_JdbcTemplate = null;
}

@Test
public void testAddBook() {
    Book book = new Book("Book_Test", 909090L, "Test
Publication", 1000, "Test Book description", "Test
author");
    int rows_insert= bookDAO_JdbcTemplate.addBook(book);
    assertEquals(1, rows_insert);
}
}

```

16. Select the `testAddBook()` method and run it as JUnit test.
17. The JUnit window will be shown with a green mark indicating the code has passed the unit test as shown in figure below:



18. The ISBN is a Primary Key in Book table, if you rerun the same `testAddBook()` it will fail showing red color instead of green. Still it proves our code is working as per logic. If one of the test conditions is failed the test case execution stops by showing `AssertionError`.

## Note

Try to write test condition which will always pass.

19. Let's add `TestAddBook_Negative()` to test what happens if we try to add book with same ISBN. Don't forget to annotate the method by `@Test`. The code will be as shown below:

```
@Test(expected=DuplicateKeyException.class)
public void testAddBook_Negative() {
    Book book = newBook("Book_Test", 909090L, "Test
Publication", 1000, "Test Book description", "Test
author");
    int rows_insert= bookDAO_JdbcTemplate.addBook(book);
```

```
        assertEquals(0, rows_insert);
    }
```

## Note

On adding duplicate key, the code will throw `DuplicateKeyException`. In the `@Test` annotation we had added `DuplicateKeyException` as expected result indicating to the JUnit Runner as it's expected behavior.

20. In the same way let's add the code to other test methods as shown below:

```
@Test
public void testUpdateBook() {
    //with ISBN which does exit in the table Book
    long ISBN = 909090L;
    int updated_price = 1000;
    int rows_insert = bookDAO_JdbcTemplate.updateBook(ISBN,
        updated_price);
    assertEquals(1, rows_insert);
}

@Test
public void testUpdateBook_Negative() {
    // code for deleting the book with ISBN not in the table
}

@Test
public void testDeleteBook() {
    // with ISBN which does exit in the table Book
    long ISBN = 909090L;
    boolean deleted = bookDAO_JdbcTemplate.deleteBook(ISBN);
    assertTrue(deleted);
}

@Test
public void testDeleteBook_negative() {
    // deleting the book with no ISBN present in the table.
}

@Test
public void testFindAllBooks() {
    List<Book> books =
        bookDAO_JdbcTemplate.findAllBooks();
    assertTrue(books.size() > 0);
    assertEquals(4, books.size());
    assertEquals("Learning Modular Java
Programming", books.get(3).getBookName());
}
```

```

    }
    @Test
    public void testFindAllBooks_Author() {
        List<Book> books =
            bookDAO_JdbcTemplate.findAllBooks("T.M.Jog");
        assertEquals("Learning Modular Java
                    Programming", books.get(1).getBookName());
    }
}

```

The above code constructs few objects such as `BookDAO_JdbcTemplate`, which have been constructed using Spring container. In the code we consumed an object of `BookDAO_JdbcTemplate` which we obtained in `setUp()` using Spring Container. Can't we have better choice instead of doing it manually? Yes, we can do it by using custom runner provided by Spring. The `SpringJUnit4ClassRunner` is a custom runner which is an extension of class `JUnit4Runner` provides a facility to use Spring TestContext Framework implicitly taking out the complexity out.

## Spring TestContext Framework

Spring empowers developers with rich Spring TestContext Framework which provides a strong support for unit as well as integration testing. It supports both API based and annotation based test case creation. The framework supports strongly JUnit and TestNG as testing frameworks. The TestContext encapsulates the spring context in which the test cases will be executed. It can also be used to load ApplicationContext, if requested. The `TestContextManager` is the main component which manages TestContext. The event publication is done by the `TestContextManager` and the `TestExecutionListener` provides the action to be taken for a published event.

The class level annotation `@RunWith` instructs the JUnit to invoke the class it is referencing to run the test cases rather than using the built in runner. The Spring provided `SpringJUnit4ClassRunner` facilitates the JUnit to use the functionalities provided by Spring Test Framework using `TestContextManager`. The `org.springframework.test.context` package provides annotation driven support of the testing. Following annotations are used to initialize the context,

## **@ContextConfiguration**

The class level annotation loads the definition to build the Spring container. The context is built either by referring a class or XML files. Let's discuss them one by one:

- Using single XML file:

```
@ContextConfiguration("classpath:connection_new.xml")
public class TestClass{
    //code to test
}
```

- Using configuration class:

```
@ContextConfiguration(class=TestConfig.class)
public class TestClass{
    //code to test
}
```

- Using configuration class as well as XML file:

```
@ContextConfiguration(locations="connection_new.xml",
loader=TestConfig.class)
public class TestClass{
    //code to test
}
```

- Using context initializer:

```
@ContextConfiguration(initializers =
TestContextInitializer.class)
public class TestClass{
    //code to test
}
```

## **@WebAppConfiguration**

The class level annotation is used to instruct how to load the ApplicationContext and used by the WebApplicationContext(WAC) from default location as "file:/src/main/webapp". The following snippet shows loading of resource to initialize WebApplicationContext to be used for

testing:

```
@WebAppConfiguration("classpath: myresource.xml")
public class TestClass{
    //code to test
}
```

Earlier developed test case was using explicit Spring Context initialization. In this demo we will discuss how to use SpringJUnit4ClassRunner and @RunWith. We will use Ch07\_JdbcTemplates\_Testing project and test methods of BookDAO\_JdbcTemplates using following steps,

1. Download spring-test-5.0.0.M1.jar file to use Spring testing APIs.
2. Create SpringRunner\_TestBookDAO\_JdbcTemplate in com.packt.ch07.tests package as a JUnit Test case. Select BookDAO\_JdbcTemplate as class under test and all of its methods under testing.
3. Annotate the class by @RunWith and @ContextConfiguration annotation as shown in the code below.
4. Add a data member of type BookDAO and apply the annotation for autowiring as shown in the code below:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:connection_new.xml")
public class SpringRunner_TestBookDAO_JdbcTemplate {
    @Autowired
    @Qualifier("bookDAO_jdbcTemplate")
    BookDAO bookDAO;

    @Test
    public void testAddBook() {
        Book book = new Book("Book_Test", 909090L, "Test
Publication", 1000, "Test Book description", "Test
author");
        int rows_insert = bookDAO_JdbcTemplate.addBook(book);
        assertEquals(1, rows_insert);
    }
}
```

5. The @RunWith annotation accepts the SpringJUnit4ClassRunner. The @ContextConfiguration accepts the file to initialize the container. Also we use annotation based auto wiring for BookDAO, instance instead of

using Spring API in `setUp()` method as we did in earlier demo. The code for testing in `testAddBook()` remains same as we are not changing the logic.

6. Execute it as JUnit test and if your ISBN is not already available in the Book table the test will pass.

In the above code we did the testing against the actual database, which makes it slower and it always will be. These tests are not isolated from the environment and they will always be dependent upon the external dependencies, in our case on the database. The unit test cases always are rewritten based on few assumptions according to the real time values. To understand the problem and complexity while dealing with the real time values.

We have a function for updating the book details. To update the book the function is having two arguments, first to accept ISBN and second to update the price of the book with specified ISBN as specified below:

```
public int updateBook(long ISBN, int updated_price)
{
    // code which fires the query to database and update the price
    // of the book whose ISBN has specified
    // return 1 if book updated otherwise 0
}
```

We wrote the test case to find whether the book is updated or not as shown below:

```
@Test
public void testUpdatedBook()
{
    long ISBN=2;    // isbn exists in the table
    int updated_price=200;
    int rows_updated=bookDAO.updateBook( ISBN, updated_price);
    assertEquals(1, rows_updated);
}
```

We assumed ISBN exists in the database to update the book details. So, the test case executed successfully. But, what if in between someone changes the ISBN or someone deletes the row with that ISBN? The test case written by us will fail. The problem is not in our test case, the only problem is we are

assuming the ISBN exists.

Another thing is, sometimes the real time environment may not be accessible. The controller layer testing is highly dependent upon request and response object. These request and response will be initialized by the container once the application is deployed to the server. Either the server may not be available for deployment or the layers on which the controller coding is independent has not been developed yet. All such problems made testing more and more difficult. These problems will easily be solved using Mock object testing.

# Mock Testing

---

Mock testing deals in testing with fake objects, these are not real objects. These fake objects returns the data required to conduct the test. It is helpful to save lots of work while carrying out actual object. The fake objects are usually called as 'Mock object'. The mock objects are used to replace the actual objects to avoid unnecessary complications and dependencies like database connections. These mock objects are isolated from the environment leading to the faster execution. The mock objects are created by setting up the data and then specifying the behavior of the method. The behavior includes the data to be returned on a particular scenario. Mockito is one of the famous testing frameworks using the mock objects.

## Mockito

Mockito is an open source testing framework for Java base applications which has released under the MIT Licence. It allows the developers to create mock objects for **Test Driven Development(TDD)** which are isolated from the framework. It uses Java Reflection API for creating mock objects and has simple APIs for writing test cases. It also facilitates the developers to have a check on the order in which the methods are getting invoked.

Mockito has static `mock()` method, which can be used for creating mock objects. It also facilitates creation of mock objects by using `@Mock` annotation. The `methodMockitoAnnotations.initMocks(this)` instructs to initialize all the annotated fields which has annotated by `@Mock`. If we forget to do so the objects would be null. The `@RunWith(MockitoJUnitRunner.class)` also does the same. The `MockitoJUnitRunner` is the custom runner which is used by the JUnit.

Mockito works on the principle of returning predefined values when a function is called, **Mokito**, `when()` method facilitates to provide the information about which method will be called and `Mokito,thenXXX()` is used to specify what values the function will return. Following are the

methods which is used to specify what values to be returned,

- `thenReturn` - used to return a specified value
- `thenThrow`- throws specified exception
- `then` and `thenAnswer` returns an answer by the user defined code
- `thenCallRealMethod`- gives a call to the real method

Mock testing is simple three step process as,

1. Initialization the dependencies byMock object for the class under test
2. Execute the operation to test
3. Write test condition to check whether the operation gives expected result or not

Let's use Mockito to create mock object of the BookDAO and use it in testing step by step as,

1. Download mokito-all-1.9.5.jar and add it to the In Ch07\_JdbeTemplate\_Testing project which we use as our base project.
2. Create `Spring_Mokito_TestBookDAO_JdbcTemplate` as Junit test case in `com.packt.ch07.unit_tests` package.
3. Add a data member of type `BookDAO` and annotate it with `@Mock` annotation.
4. To initialize the mock object invoke `initMocks()` method of the Mockito in `setUp()` method as shown below:

```
public class Spring_Mokito_TestBookDAO_JdbcTemplate {  
    @Mock  
    BookDAO bookDAO_JdbcTemplate;  
  
    @Before  
    public void setUp() throws Exception  
    {  
        MockitoAnnotations.initMocks(this);  
    }  
}
```

5. Let's now add code to test `addBook()` where we will first define the values which we are expecting the function under test to return. Then we will use `assertXXX()` methods to test the behavior as shown below:

```

    @Test
    public void testAddBook() {
        Book book = new Book("Book_Test", 909090L, "Test
Publication", 1000, "Test Book description",
"Test author");
        //set the behavior for values to return in our case addBo
//method
        Mockito.when(bookDAO_JdbcTemplate.addBook(book)).thenReturn

        // invoke the function under test
        int rows_insert = bookDAO_JdbcTemplate.addBook(book);

        // assert the actual value return by the method under tes
//the expected behaiour by mock object
        assertEquals(1, rows_insert);
    }
}

```

6. Execute the test case and test the behavior. We will get all test case executed successfully.
7. Let's add the code for other findAllBooks(String) and deleteBook()methods as well:

```

    @Test
    public void testDeleteBook() {

        //with ISBN which does exit in the table Book
        long ISBN = 909090L;
        Mockito.when(bookDAO_JdbcTemplate.deleteBook(ISBN)) .
            thenReturn(true);
        boolean deleted = bookDAO_JdbcTemplate.deleteBook(ISBN);
        assertTrue(deleted);
    }

    @Test
    public void testFindAllBooks_Author() {
        List<Book> books = new ArrayList();
        books.add(new Book("Book_Test", 909090L, "Test
Publication", 1000, "Test Book description", "Test
author"));

        Mockito.when(bookDAO_JdbcTemplate.findAllBooks("Test
author")).thenReturn(books);
        assertTrue(books.size() > 0);
        assertEquals(1, books.size());
        assertEquals("Book_Test", books.get(0).getBookName());
    }
}

```

In the previous demo, we discussed about unit testing of DAO layer in both real time environment as well as using mock objects. Let's now test the controller using Spring MVC test framework in the following sections.

## **Spring MVC controller testing using Spring TestContext framework**

The Mockito facilitates the developers to create mock objects of the DAO layer in the earlier discussion. We were not having DAO object but even without it the testing was made possible. The Spring MVC layer testing without mock objects is not possible as they are highly dependent upon the request and response object which gets initialized by the container. The spring-test module supports creation of mock object for Servlet API which makes testing of the web component without the actual container deployment. The following table shows the list of packages which has been provided by Spring TestContext framework for mock creation:

<b>Package name</b>	<b>Provides mock implemenetation of</b>
org.springframework.mock.env	Environment and PropertySource
org.springframework.mock.jndi	JNDI SPI
org.springframework.mock.web	Servlet API
org.springframework.mock.portlet	Portlet API

The org.springframework.mock.web provides the MockHttpServletRequest, MockHttpServletResponse, MockHttpSession as mock objects for HttpServletRequest, HttpServletResponse and HttpSession for use. It also provides the class ModelAndViewAssert to test the ModelAndView objects from Spring MVC framework. Let's test our SearchBookController step by step as follows:

1. Add spring-test.jar to the ReadMyBooks application which we will use for testing.
2. Create `com.packt.ch06.controllers.test_controllers` package to add test cases for the controllers.
3. Create `TestSearchBookController` as JUnit case in the package created in earlier step.
4. Annotate it by `@WebAppConfiguration`.
5. Declare data members of type `SearchBookController` and autowire it as shown in the code below:

```
@WebAppConfiguration
@ContextConfiguration({ "file:WebContent/WEB-INF/book-
    servlet.xml" })
@RunWith(value = SpringJUnit4ClassRunner.class)
public class TestSearchBookController {
    @Autowired
    SearchBookController searchBookController;
}
```

6. Let's test add `testSearchBookByAuthor()` to test `searchBookByAuthor()` method. The method accepts author's name entered by the user in the web form and returns list of the books written by the author. The code will be written as:
  1. Initialize the data required by the method under testing
  2. Invoke the method under test
  3. Assert the values.
7. The final code will be as shown below:

```
@Test
public void testSearchBookByAuthor() {
    String author_name = "T.M.Jog";
    ModelAndView modelAndView =
        searchBookController.searchBookByAuthor(author_name);
    assertEquals("display", modelAndView.getViewName());
}
```

4. We are testing the name of the view 'display' which has been written from the controller method.
5. The Spring facilitates `ModelAndViewAssert` which provides the method to test `ModelAndView` returned by the controller method as shown in

the code below:

```
@Test  
public void testSerachBookByAuthor_New()  
{  
    String author_name="T.M.Jog";  
    List<Book>books = new ArrayList<Book>();  
    books.add(new Book("Learning Modular Java Programming",  
        9781235, "packt pub publication", 800,  
        "explore the power of modular Programming ", author_name);  
    books.add(new Book("Learning Modular Java Programming",  
        9781235, "packt pub publication", 800,  
        "explore the power of modular Programming ", author_name);  
    ModelAndView modelAndView =  
        searchBookController.searchBookByAuthor(author_name);  
    ModelAndViewAssert.assertModelAttributeAvailable(  
        modelAndView, "book_list");  
}
```

6. Execute the test case, the green colour indicates the test case has passed.
7. We successfully tests the SearchBookController which has easy coding without any form submission, form model attribute binding, form validation and many more. Such complex code testing becomes more complex with mock objects which we just handled.

## Spring MockMvc

Spring provides the MockMVC, as the main entry point which is empowered with methods to start with the server side testing. The implementation of the MockMVCBuilder interface will be used to create a MockMVC object. The MockMVCBuilders provides the following static methods which gives opportunity to get implementation of the MockMVCBuilder:

- `xmlConfigSetUp(String ...configLocation)` - will be used when the application context is configured using XML configuration files as shown below:

```
MockMvc mockMvc =  
    MockMvcBuilders.xmlConfigSetUp("classpath:myConfig.xml").bu
```

- `annotationConfigSetUp(Class ... configClasses)` - will be used when we

are using Java class to configure the application context. The following code shows how to use the MyConfig.java as a configuration class:

```
MockMvcmockMVC=
    MockMvcBuilders.annotationConfigSetUp(MyConfiog.class).
        build();
```

- standaloneSetUp(Object ... controllers) - will be used when developers configured the test controllers and its required MVC components. The following code shows using MyController for the configuration:

```
MockMvcmockMVC= MockMvcBuilders.standaloneSetUp(
    newMyController()).build();
```

- webApplicationContextSetUp(WebApplicationContext context) - will be used when the developers already had fully initialized WebApplicationContext instance. The following code shows how to use the method:

```
@Autowired
WebApplicationContextwebAppContext;
MockMvcmockMVC= MockMVCBuilders.webApplicationContextSetup(
    webAppContext).build();
```

MockMvc has perform() method which accept the instance of RequestBuilder and returns the ResultActions. The MockHttpServletRequestBuilder is an implementation of RequestBuilder who has methods to build the request by setting request parameters, session. The following table shows the methods which facilitates building the request,

<b>Method name</b>	<b>The data method description</b>
accept	Helps in setting the 'Accept' header to the given media type
buildRequest	Helps in building the MockHttpServletRequest

`createServletRequest` Based on the ServletContext, the method creates a new MockHttpServletRequest

Param	Helps in setting request parameter to the MockHttpServletRequest.
principal	Helps in setting the principal of the request.
locale .	Helps in setting the locale of the request.
requestAttr	Helps in setting a request attribute.
Session, sessionAttr, sessionAttrs	Helps in setting session or session attributes to the request
characterEncoding	Helps in setting character encoding to the request
content and contentType	Helps in setting the body and content type header of request
header and headers	Helps in adding one or all headers to the request.
contextPath	Helps in specifying the part of requestURI which represents the context path
Cookie	Helps in adding cookies to the request

flashAttr	Helps in setting input flash attribute.
pathInfo	Helps to specify the part of the requestURI that represents the pathInfo.
Secure	Helps in setting the secure property of the ServletRequest such as HTTPS.
servletPath	Helps to specify the part of the requestURI which represents the path to which the Servlet is mapped.

The `perfrom()` method of MockMvc return the ResultActions facilitates the assertions of the expected result by following methods:

Method name	Description
andDo	It takes a general action.
andExpect	It takes the expected action
annReturn	It return the result of the expected request which can be directly accessed.

Let's use MockMvc to test AddBookController step by step:

1. Add TestAddBookController as JUnit test case in  
`com.packt.ch06.controllers.test_controllers package.`
2. Annotate the class by `@WebAppConfiguration`,  
`@ContextConfiguration` and `@RunWith` as we did in earlier code.

3. Add the data members of type WebApplicationContext and AddBookController. Annotate both by @Autowired.
4. Add data member of type MockMvc and initialize it in setup() method and release memory in teardown() method as shown below:

```

@WebAppConfiguration
@ContextConfiguration(
    { "file:WebContent/WEB-INF/books-servlet.xml"})
@RunWith(value = SpringJUnit4ClassRunner.class)
public class TestAddBookController {
    @Autowired
    WebApplicationContext wac;

    MockMvc mockMvc;

    @Autowired
    AddBookController addBookController;

    @Before
    public void setUp() throws Exception {
        mockMvc =
            MockMvcBuilders.standaloneSetup(addBookController).bu
    }
}

```

- Lets add the code to test the addBook() method in testAddBook() as:
  1. Initialize the request by setting values of:
    - model attribute 'book' with default values
    - content-type as the form submission results in method invocation
    - URI on which the method will be invoked
    - request parameters of the form
      2. test the result by checking,
    - view name
    - model attribute name
      3. use andDo() to print the result of test actions on cosole

The code for testAddBook() method is as shown below:

```

@Test
public void testAddBook() {
    try {
        mockMvc.perform(MockMvcRequestBuilders.post("/addBook.h
            .contentType(MediaType.APPLICATION_FORM_URLENCODED)

```

```

        .param("bookName", "Book_test")
        .param("author", "author_test")
        .param("description", "adding book for test")
        .param("ISBN", "1234")
        .param("price", "9191")
        .param("publication", "This is the test publication")
        .requestAttr("book", new Book()))
        .andExpect(MockMvcResultMatchers.view().name("display"))
        .andExpect(MockMvcResultMatchers.model().
            attribute("auth_name", "author_test"))
        .andDo(MockMvcResultHandlers.print());
    } catch (Exception e) {
        // TODO: handle exception
        fail(e.getMessage());
    }
}
}

```

The matching of the expected behavior in the `andExpect()` is facilitated by `ResultMatcher`. The `MockMvcResultMatcher` is an implementation of the `ResultMatcher` provides the methods to match the view, cookie, header, model, request and many other parameters. The `andDo()` method prints the `MvcResult` to the `OutputStream`.

4. Run the test case and surprisingly it will fail. The part of the output is as shown below:



5. It shows the validation error, but we had given all input as per the validation rules. Which validation has failed is not clear from the output. No, no need to panic and to check the validation one by one.
6. Instead of creating more confusion let's add test code for validation using `attributeHasErrors()` as shown below by underlined statement:

```

@Test
public void testAddBook_Form_validation() {
    try {
        mockMVC.perform(MockMvcRequestBuilders.post("/addBook.h
            .param("bookName", "Book_test")
            .param("author", "author_test")
            .param("description", "adding book for test")

```

```

        .param("ISBN", "12345")
        .param("price", "9191")
        .param("publication", "This is the test publication")
        .requestAttr("book", new Book()))
        .andExpect(MockMvcResultMatchers.view().name("bookForm"))
        .andExpect(MockMvcResultMatchers .model() .
            attributeHasErrors("book"))
        .andDo(MockMvcResultHandlers.print());
    }
    catch (Exception e) {
        fail(e.getMessage());
        e.printStackTrace();
    }
}
}

```

7. The test runs successfully proving the input has validation error. We can get the field whose validation failed on console output in the 'errors' as:

```

MockHttpServletRequest:
  HTTP Method = POST
  Request URI = /addBook.htm
  Parameters = {bookName=[Book_test],
  author=[author_test],
  description=[adding book for test],
  ISBN=[1234],
  price=[9191],
  publication=[This is the test publication]}
  Headers = {
    Content-Type=[application/x-www-form-urlencoded] }
Handler:
  Type = com.packt.ch06.controllers.AddBookController
  Method = public
org.springframework.web.servlet.ModelAndView
com.packt.ch06.controllers.AddBookController.
addBook(com.packt.ch06.beans.Book,org.
springframework.validation.BindingResult)
throwsjava.lang.Exception
Async:
  Async started = false
  Async result = null

Resolved Exception:
  Type = null
ModelAndView:
  View name = bookForm
  View = null
  Attribute = priceList

```

```

value = [300, 350, 400, 500, 550, 600]
Attribute = book
value = Book_test adding book for test 9191
errors = [Field error in object 'book' on field
'description':
rejected value [adding book for test];
codes
[description.length.book.description,
description.length.description,description.
length.java.lang.String,description.length];
arguments [];
default message [Please enter description
within 40 charaters only]]
FlashMap:
Attributes = null
MockHttpServletResponse:
Status = 200
Error message = null
Headers = {}
Content type = null
Body =
Forwarded URL = bookForm
Redirected URL = null
Cookies = []

```

8. Though the description has characters within the specified limit of 10 to 40. Let's find out the rule to get what mistake we did in Validator2.
9. The code in the validate method for setting validation rule for publication is:

```

if (book.getDescription().length() < 10 || 
    book.getDescription().length() < 40)
{
    errors.rejectValue("description", "description.length",
        "Please enter description within 40 charaters only");
}

```

10. Yes, we set the validation for publicationlength as less than 40 which lead to failure. We made mistake. Let's change the code to set rules as length greater than 40 will not be allowed. The updated code is as shown below:

```

if (book.getDescription().length() < 10 || 
    book.getDescription().length() > 40)
{

```

```

        errors.rejectValue("description", "description.length",
            "Please enter description within 40 charaters only");
    }
}

```

11. Now rerun the testAddController to find what happens.
12. The test case passes successfully. This is why we are carrying out the test cases.
13. Let's now add the code to test field validations in the testAddBook\_Form\_Validation().as:

```

@Test
public void testAddBook_Form_Field_Validation()
{
    try {
        mockMVC.perform(MockMvcRequestBuilders.post("/addBook.h
            .param("bookName", "")
            .param("author", "author_test")
            .param("description", " no desc")
            .param("ISBN", "123")
            .param("price", "9191")
            .param("publication", " ")
            .requestAttr("book", new Book()))
            .andExpect(MockMvcResultMatchers.view().name("bookForm"))
            .andExpect(MockMvcResultMatchers.model())
            .attributeHasFieldErrors("book", "description")).andExp
                MockMvcResultMatchers.model()
            .attributeHasFieldErrors("book", "ISBN")).andExpect(
                MockMvcResultMatchers.model())
            .attributeHasFieldErrors("book", "bookName") .
            andDo(MockMvcResultHandlers.print());
    } catch (Exception ex)
    {
        fail(ex.getMessage());
    }
}

```

14. Run the test case where validation errors failed.

The Controllers and DAOs are working fine. The service layer is using DAOs, so let's conduct the integration testing of service layer. You can conduct mock object testing of service layer as per we discussed and did in DAO layer testing. We will move on to integration testing of the service as next phase.

# Pase II Integration testing

---

## Integration testing of Service and DAO layer

Let's carry out integration testing of the application, Ch05\_Declarative\_Transaction\_Management step by step as follow:

1. Create com.packt.ch05.service.integration\_tests package.
2. Create JUnit test case TestBookService\_Integration by considering BookServiceImpl as class under test. Select all of its methods to test.
3. Declare the data member of type BookService, annotate it by @Autowired annotations as shown in code below:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:connection_new.xml")
public class TestBookService_Integration
{
    @Autowired
    BookService bookService;
}
```

4. Let's test addBook() method, as we did in JUnit testing earlier. You can refer the code below:

```
@Test
public void testAddBook() {
    // Choose ISBN which is not there in book table
    Book book = new Book("Book_Test", 909098L, "Test
Publication", 1000, "Test Book description", "Test
author");
    boolean flag = bookService.addBook(book);
    assertEquals(true, flag);
}
```

5. You can run the test case which will run successfully.

## Note

All other test methods from BookService can be referred from the source code.

Both the layers developed by us are working as we want them to. We developed controllers, services and DAOs separately and tested as well. Now, we will combine them in single application so as one complete application will be with us and then using integration testing we will check whether it is working as per expectation.

## Integration testing of Controller and Service Layer

Let's combine the three layers together in ReadMyBooks from Ch05\_Declarative\_Transaction\_Management as stated by steps below:

1. Add jars for jdbc and spring-jdbc and other required jars in lib folder of ReadMyBooks.
2. Copy the com.packt.ch03.dao and com.packt.ch05.service packages from Ch05\_Declarative\_Transaction\_Management to ReadMyBooks application.
3. Copy connection\_new.xml in the class path of ReadMyBooks application.
4. In the Book class for Form Submission we had commentated the default constructor, the logic in addBook of service is to check against 98564567 as default value.
5. Change the BookService as shown below by underline keeping rest of the code untouched:

```
@Override  
@Transactional(readOnly=false)  
public boolean addBook(Book book) {  
    // TODO Auto-generated method stub  
  
    if (searchBook(book.getISBN()).getISBN() == 0) {  
        //  
    }  
}
```

6. The controllers needs to be updated to get a talk with under lying layer as:
  - Add autowired data member of type BookService in the controllers.
  - Invoke the methods of service layer in method of controllers as per business logic requirements.
7. The addBook() method will be updated as shown below:

```

@RequestMapping("/addBook.htm")
public ModelAndView addBook(@Valid @ModelAttribute("book")
Book book, BindingResult bindingResult)
throws Exception {
    // same code as developed in the controller
    // later on the list will be fetched from the table
    List<Book> books = new ArrayList();

    if (bookService.addBook(book)) {
        books.add(book);
    }
    modelAndView.addObject("book_list", books);
    modelAndView.addObject("auth_name", book.getAuthor());
    return modelAndView;
}

```

## Note

In the same way we can update all the methods from the controller. You can refer the complete source code.

Let's execute the test case TestAddBookController.java to get the result.

The code will execute and gives success message. Also in the table one row with the ISBN and other values which we specified gets added.

We had done testing of all the components successfully. We can now directly start with SystemTesting.

But have patience as we will discuss about the new entry to the testing framework 'Arquillian'.

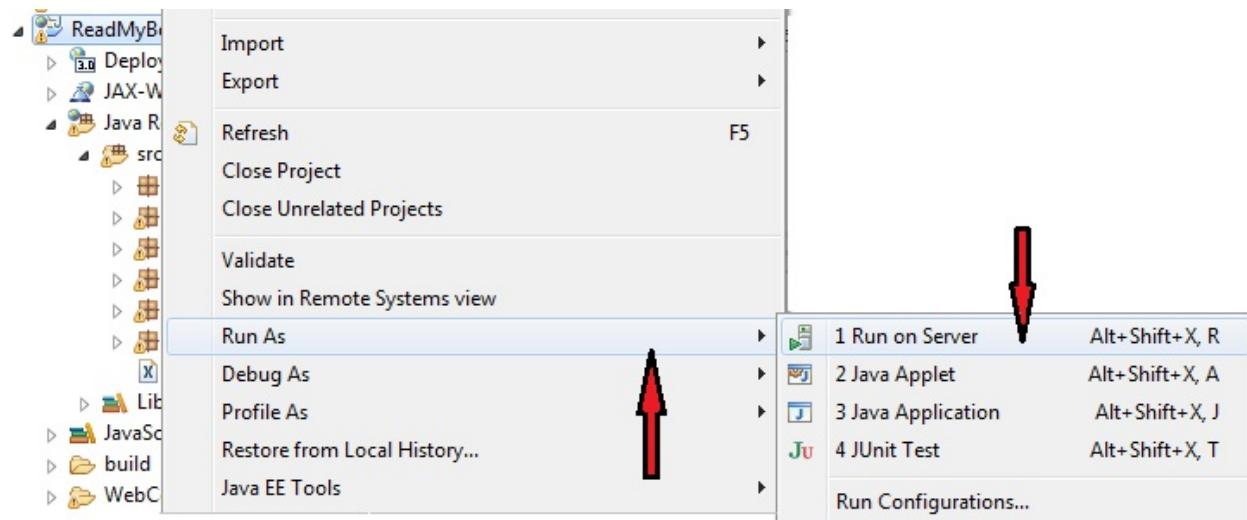
# Pase III System testing

---

All the layers are working as per expectations now it's time to test the application by using the web i.e. will check one by one functionalities by taking at most care to go step by step and not only the result but the presentation also will be observed which will be close to the actual deployment environment . Let's deploy the application to check all the functions are working and giving the correct results both on data base side as well as presentation side by either of the ways as discussed below.

## Using Eclipse IDE for deployment

In eclipse once you are finished with development, configure the server and select the project from the Project Explorer to choose **Run on server** option as shown by the arrows below:

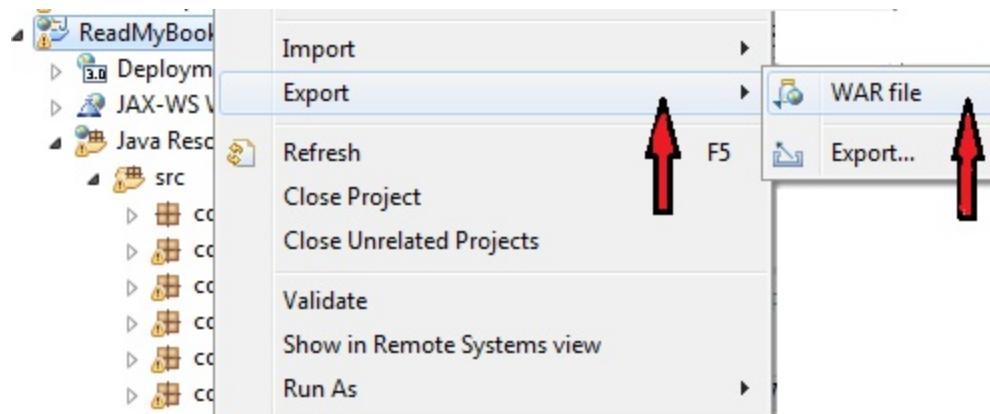


The IDE will package the application in the war file and will deploy it to the container. Now you can check the functionalities one by one to make sure each and everything is as per the expectation or not. We will take care of presentation, look and feel as well as accuracy of the data which is displayed by the presentation.

## Manually deploying the application

To manually deploy the application can be by the following steps:

1. First of all we need to get the jar file of it. We can use the Eclipse IDE to get the war file for deployment easily by right clicking the application and selecting **Export** as shown by arrows below:



2. Select the destination where the war you want to create the war file. If you want you can change the war file name. I will keep ReadMyBooks as it is.
3. Click on **finish** to complete the process. You will get a war file on the selected destination.
4. Copy the WAR file which we created in earlier step and paste it in 'webapps' folder under Tomcat directory.
5. Start tomcat by clicking `startup.bat` file from `bin` folder
6. Once the tomcat started, open the browser and type home url in the format of [http://host\\_name:port\\_number\\_of\\_tomcat/war\\_file\\_name](http://host_name:port_number_of_tomcat/war_file_name). in our case it is <http://localhost:8080/ReadMyBooks>.
7. Before moving ahead make sure the database parameters are correctly set otherwise the application will fail.
8. The home page will open where we can test the application for the functionalities and look and feel.

# Summary

---

In this chapter we discussed about what is testing and why it is so important. We also had a discussion about the unit testing, integration testing and User acceptance test as phases of testing. There are many testing tools available in market, we had taken an overview of these tools so that the choice of tool can be wisely done by you. One of the very important tool of testing is 'JUnit testing', which we used to carry out unit testing of the DAO layer which was the starting of testing phase1. But JUnit uses realtime database, we discuss the difficulties in testing upon the external parameters. We resolve the issue by using mock objects. Mockito is one of the Mock object creation tool which we explore to test the DAO layer. After DAO layer we tested Web layer which also was having dependency upon the web container which initializes request and response objects. We discuss in depth about Spring TestContext framework whose MockMVC module facilitates creation of Mock objects of the web related components like request and responses. We used the the framework for testing the form validation as well. After Unit testing we carry out the integration testing of DAO and service layer and then web and service layers. The story won't end here we carry out successful deployment and final checking of the product by carrying out System Testing. All the components developed by us are working fine and we had proved it by successfully executing the System Testing !!

In next chapter we will go one step ahead and discuss about the role of security in an application along with the ways provided by Spring framework to implement security. Keep reading!!!!

# **Chapter 8. Explore the Power of Restful Web Services**

In earlier chapter we discussed about building Spring MVC application. These applications facilitate the user to serve through web only for Java platform. What if some other platform wants to use the functionalities developed by us? Yes we need functionalities which are platform independent. In this chapter we will discuss how to develop such platform independent services using Restful web services to address following topics:

- What is web service?
- Importance of web services.
- Types of web services
- Restful web services
- Developing Spring restful web services.
- How to use RestTemplate and POSTMAN to test the web services?
- The presentation of the data using message converters and content negotiation.

# **Web services**

---

Web service is the way of communication between two or more applications which have developed for different platforms. These services are independent of browsers and operating systems which make easy communication and enhanced performance to target more users. This service can be as easy as a function, a collection of standards or protocols which has been deployed on the server. It is a communication between a client and server or communication between two devices through network. Let's say we developed a service in Java and published it on internet. Now this service can be consumed by any Java based applications, but more importantly any .NET based or Linux based applications can also consume it with the same ease. This communication is done through set of XML based messages over the HTTP protocol.

## **Why we Need of web service?**

Interoperability is one of the best thing which can be achieved by web services along with which they provides following

### **Usability**

Many applications invest their valuable time in developing complex function which is already available in other application. Instead of redeveloping it web services allow developers to explore such services exposed over web. It also leverages to develop customized client side logic reusing the web services saving valuable time.

### **Reusing the developed application**

The technologies and market so moving so fast, the developers has to keep on matching the client requirements. It's very common in development to redevelop an application using a new platform to support new features with

ease. Instead of developing the complete application from scratch, the developers can now add enhanced functionalities with whatever the platform they want, and use the old modules using web services.

## **Loosely coupled modules**

Each service developed as web service is totally independent of any other services which supports ease of modifying them without any effect on other part of the application.

## **Ease in deployment**

The web services are deployed over the servers to facilitates the use through internet. The web service can be deployed over the fire walls to the server through internet with the same ease as they can be deployed in local servers.

## **Types of web services**

### **SOAP web service**

### **RESTful web service**

RESTful web service is Representational state transfer which is an architectural style. The RESTfuul resources are revolver around the transfer of data in some represenatational format. The REST resources will be in the form which needed suits the consumer. It can be in representation forms like XML, JSON or HTML. In RESTful web services the state of the resource is more important than the action took against the resource.

Advantages of RESTful web services:

- RESTful web services are fast as it consumes less resources and band width.
- It can be written and execute on any platform.
- The most important thing is it allows different platforms such as HTML, XML, Plain text and JSON.

## Spring and RESTful web services

Spring supports writing of RestController which can handle requests for HTTP requests using @RestController annotation. It also provides @GetMapping, @PostMapping, @DeleteMapping, @PutMapping annotations to handle HTTP get, post, delete and put methods. The @PathVariable annotation facilitates access of the values from the URI template. Current most of the browsers support using GET and POST as HTTP methods as html actions methods. The HiddenHttpMethodFilter now enables submission of form for PUT and DELETE methods using <form:form> tags. Spring facilitates the selecting the suitable view depending upon requested media type using ContentNegotiatingViewResolver. It implements the ViewResolver which already has used in Spring MVC. It delegates the request to the appropriate view resolvers automatically. Spring framework has introduced @ResponseBody and @RequestBody to bind the method parameters either to request or response. The request and response communication from the clients read and write data with variety of formats which may need message converters. Spring provides many message converters like StringHttpMessageConverter, FormHttpMessageConverter, MarshallingHttpMessageConverter to perform reading and writing. The RestTemplate provides easy client side consumption of RESTful web services.

Before moving ahead let's develop a RESTController to understand the flow and URI consumptions with the help of following steps:

1. Create Ch09\_Spring\_Restful as dynamic web application and add the jars which we added for Spring web MVC application.
2. Add DispatcherServlet as a font controller mapping in web.xml file as shown below to map all the URLs:

```
<servlet>
    <servlet-name>books</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>books</servlet-name>
```

```
<url-pattern>/*</url-pattern>
</servlet-mapping>
```

3. Add books-servlet.xml to configure base package name to scan for controller and view resolvers which we have added in every Spring web MVC application.
4. Create MyRestController class in com.packt.ch09.controllers package.
5. Annotate the class by @RestController.
6. Add the method getData() for consuming '/welcome' URI as shown in the code below:

```
@RestController
public class MyRestController {

    @RequestMapping(value="/welcome",method=RequestMethod.GET)
    public String getData()
    {
        return("welcome to web services");
    }
}
```

The getData() method will be serving the request for '/welcome' URL for GET as HTTP method and returns a String message as the response.

7. Deploy the application to the container and once the service is successfully deployed it's time to test the application by creating the client.
8. Let's write client using RestTemplate provided by Spring as shown below:

```
public class Main {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String URI=
            "http://localhost:8080/Ch09_Spring_Restful/welcome";
        RestTemplate template=new RestTemplate();
        System.out.println(template.getForObject(URI, String.class));
    }
}
```

Executing the main function will display "welcome to web services" on your console.

## RestTemplate

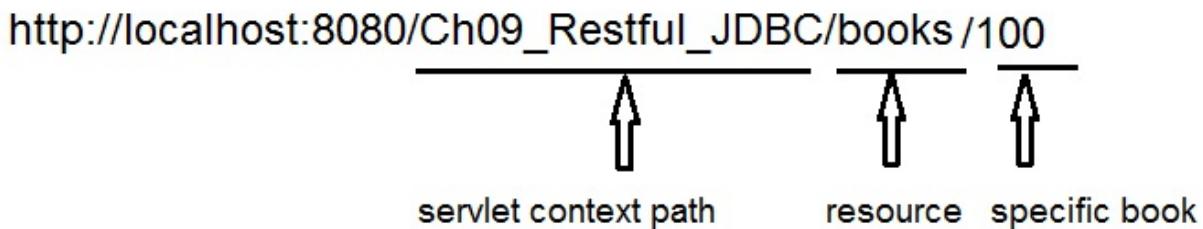
Similar to many other template classes like JdbcTemplate and HibernateTemplate the RestTemplate class is also designed to perform complex functionalities to give a call to REST services. The following table sumaries the methods provided by RestTemplate to map HTTP methods:

RestTemplate method	HTTP method	Description
getForEntity and getForObject	GET	It retrieves the representation on the specified URI
postForLocation and postForObject	POST	It creates a new resource by posting the new object on the specified URI location and it returns the header having value as Location.
put	PUT	It creates or updates the resource at the specified URI
delete	DELETE	It delete the resource specified by the URI
optionsForAllow	OPTIONS	The method returns the value of allowed headers for the specified URL.
execute and exchange	Any	Execute the HTTP method and returns the response as ResponseEntity

We cover most of them in upcoming demo. But before diving into RESTful webservices let's discuss the most important part of the RESTful web service

'URL. RestContollers handle the request only if it has been requested by the correct URL. The Spring MVC controllers also handle the web request which are request parameter and request query oriented while the URLs handled by the RESTful web services are resource oriented. The identification about the resource to map is done by the entire base URL without any query parameters.

The URLs written are based upon plural nouns in it and try to avoid using verbs or query parameters as we did in earlier demo for Spring MVC. Let's discuss the way URLs are formed. The following is RESTful URL for the resource which is a combination of the Servlet context, the resource noun to fetch and path variable:



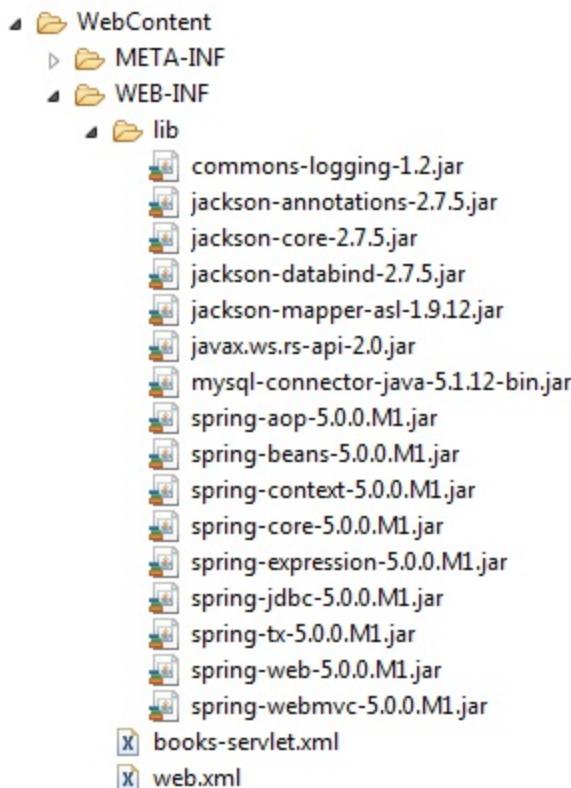
Observe the following table to know more about the RESTful URLs:

Supported HTTP methods	GET method	POST method	PUT method	DELETE method
Resource to fetch				
/books	Returns the list of books	Add a new book	Update the book or books	Delete the books
/books/100	Returns the book	405	Update the book	Delete the book

Let's develop an application to use different HTTP methods and URLs to

have better understandings by the steps below. We will use Ch03\_JdbcTemplate as our dao layer in this application from where you can directly copy the required codes.

1. Create Ch09\_Restful\_JDBC and add all the required jars as shown in the outline of WebContent folder:



2. Add front controller and web component mapping file as we did in earlier application in web.xml and books-servlet.xml. You can copy the same from earlier applications. Don't forget to add 'contextConfigLocation' as we are writing more than one bean configuration files.
3. Add Book.java in com.ch03.beans as POJO which we had used in all JDBC applications.
4. Add com.packt.ch03.dao package containing BookDAO and BookDAO\_JdbcTemplate class.
5. Add connection\_new.xml in classpath.
6. Create MyBookController class in com.packt.ch09.controllers package and annotate it by @RestController.

7. Add BookDAO as data member and annotate it by `@Autowired` annotation.
8. Now add we will add `getBook()`method to handle web service request to search the book. Annotate the method by `@GetMapping` mapped for the URL '`/books/{ISBN}`' as shown in the following code:

```

@RestController
@EnableWebMvc
public class MyBookController {

    @Autowired
    BookDAO bookDAO;
    @GetMapping("/books/{ISBN}")
    public ResponseEntity<Book> getBook(@PathVariable long ISBN)

        Book book = bookDAO.getBook(ISBN);
        if (null == book) {
            return new ResponseEntity<Book>(HttpStatus.NOT_FOUND);
        }

        return new ResponseEntity(book, HttpStatus.OK);
    }
}

```

The `@GetMapping` set the method to handle GET requests for the URLs in the form of '`books/{ISBN}`'. The `{name_of_variable}` acts as the place holder so that a data can be passed to the method for use. We also have used `@PathVariable` annotation applied to the first parameter in the method signature. It facilitates the binding of value of the URL variable to the parameter. In our case ISBN have the value passed by the URL's ISBN.

The `HttpStatus.NO_CONTENT` states that the status of the response to be set which indicates the resource has been processed but the data is not available.

The `ResponseEntity` is an extension of the `HttpEntity` where additional information about the `HttpStatus` to the response.

Let's add the Client code to use the mapped resource using `RestTemplate` as shown below:

```

public class Main_Get_Book {
    public static void main(String[] args) {

```

```

// TODO Auto-generated method stub

RestTemplate template=new RestTemplate();
Book book=
template.getForObject(
"http://localhost:8081/Ch09_Spring_Rest_JDBC/books
Book.class");
System.out.println(book.getAuthor()+"\t"+book.getISBN
)
}
}

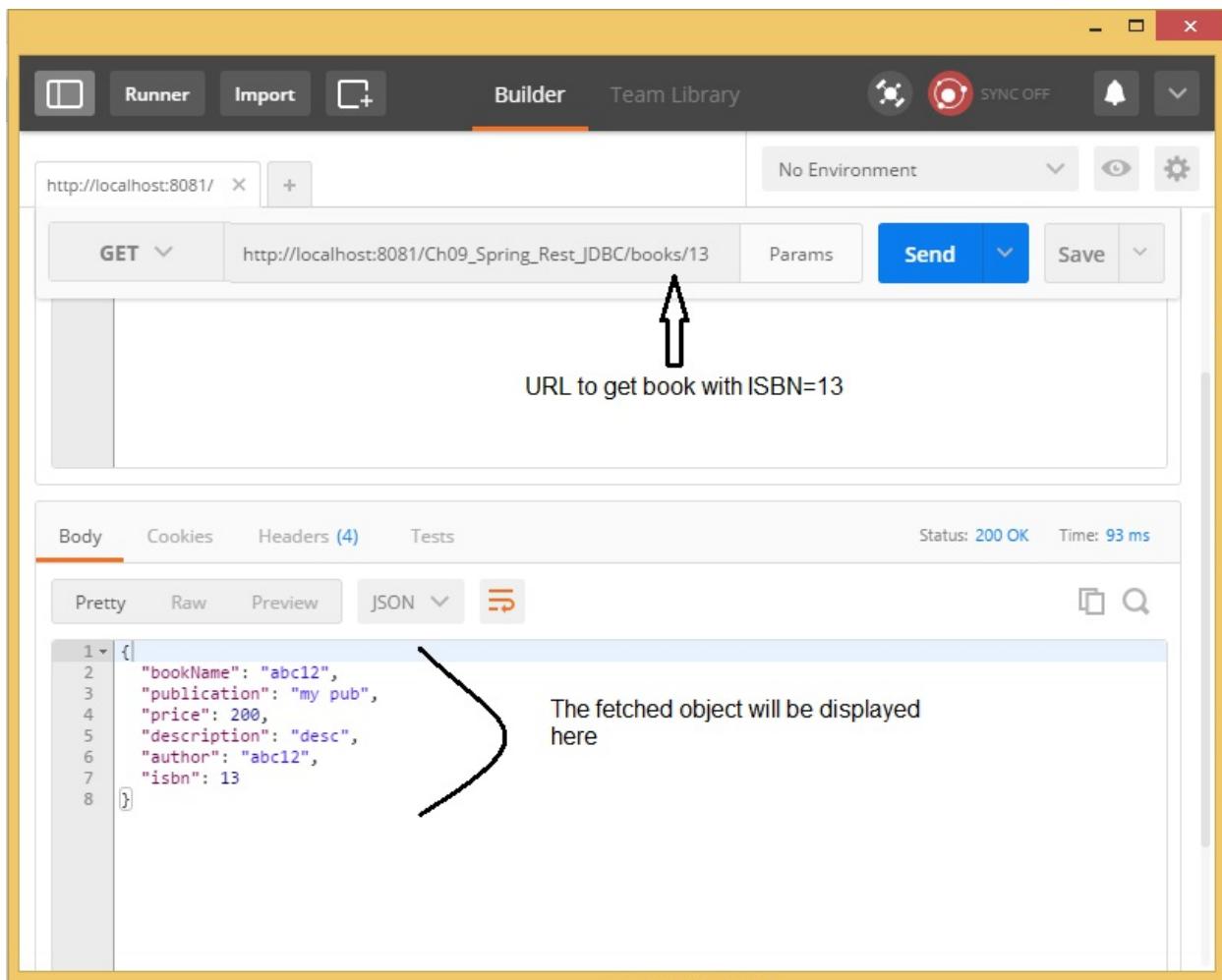
```

Here we are getting the book with ISBN=14. Make sure this ISBN is available in the table, if not you can add your value.

Execute the Main\_Get\_Book to get the book details on console.

We can test the RESTful web services using POSTMAN tool available in Google Chrome using following steps:

1. You can install in your Google from  
<https://chrome.google.com/webstore/detail/postman-rest-client/fdmmgilgnpjigdojopojpjoooidkmcomcm>
2. Once you install launch it by clicking the Postman icon.
3. Now from drop down select GET method and enter the URL  
http://localhost:8081/Ch09\_Spring\_Rest\_JDBC/books/13 in the textfield.
4. Click on **send** button.
5. We will get the list displayed in the body as shown below by the image:



The URL specifies only which the handler method will be handling the request, but it cannot decide what action to be taken at the resource. As in the discussed demo we use handled HTTP GET method to get the data.

Once we know how to get the data now let's update the data by adding the method using following steps:

1. Add updateBook() method in the MyBookController which will be annotated by `@PutMapping` to handle URL as follows:

```

@PutMapping("/books/{ISBN}")
public ResponseEntity<Book> updateBook(@PathVariable lo
ISBN, @RequestBody Book book)
{
    Book book_searched = bookDAO.getBook(ISBN);
    if (book_searched == null) {
        return new ResponseEntity(HttpStatus.NOT_FOUND);
    }
}

```

```

        }
        bookDAO.updateBook(ISBN, book.getPrice());

        book_searched.setPrice(book.getPrice());
        return new ResponseEntity(book_searched, HttpStatus.OK)
    }
}

```

Here the URL is mapped for `PUT` method.

The `updateBook()` has:

- The argument as ISBN which has been bounded for the value by `@PathVariable` annotation.
- Second argument is of type Book annotated by `@ResponseBody`. The `@ResponseBody` annotation marker for the HTTP response body which is used to bind HTTP response body to the domain object. This annotation uses the standard HTTP Message Converters by Spring framework to convert the response body to the respective domain object.

In this case the `MappingJacksonHttpMessageConverter` will be chosen to convert the arrived JSON message to Book object. To use the converter we had added related libraries in the lib folder. We will discuss in detail about message converters in upcoming pages.

2. The client code to update the Book as shown below:

```

public class Main_Update {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        RestTemplate template = new RestTemplate();

        Map<String, Long> request_parms=new HashMap<>();
        request_parms.put("ISBN",131);

        Book book=new Book();
        book.setPrice(200);
        template.put
            ("http://localhost:8081/Ch09_Spring_Rest_JDBC/books
             book,request_parms");
    }
}

```

The `put` method has the signature as:

```
void put(URL_for_the_resource, Object_to_update, Map_of_va
```

- Let's now test it from POSTMAN by entering the URL, selecting PUT method from drop down and value for the body as shown below and click on send:

The screenshot shows the POSTMAN application interface. At the top, the URL is set to `http://localhost:8081/Ch09_Spring_Rest_JDBC/books/13`. The method dropdown is set to `PUT`. The `Body` tab is selected, showing a JSON payload:

```
1 {  
2   "price": "1200"  
3 }
```

Below the request, the response section is visible, showing a `200 OK` status and a response time of `106 ms`. The `Body` tab is selected, displaying the response JSON:

```
1 {  
2   "bookName": "abc12",  
3   "publication": "my pub",  
4   "price": 1200,  
5   "description": "desc",  
6   "author": "abc12",  
7   "isbn": "13
```

After getting and updating the data now let's add the code for the resource for adding a Book using following steps:

- Add a method `addBook()` in the controller annotated by `@PostMapping`.
- We will use `@RequestBody` annotation to bind the HTTP request body to the domain object 'book' as shown in the code below:

```
@PostMapping("/books")
public ResponseEntity<Book> addBook(@RequestBody Book boo
    System.out.println("book added" + book.getDescription()
    if (book == null) {
        return new ResponseEntity<Book>(HttpStatus.NOT_FOUND)
```

```

    }
    int data = bookDAO.addBook(book);
    if (data > 0)
        return new ResponseEntity(book, HttpStatus.OK);
    return new ResponseEntity(book, HttpStatus.NOT_FOUND);
}

```

The `@RequestBody` annotation binds the request body to the domain object, here in our case it's Book object.

## 6. Now let's add Client code as shown below:

```

public class Main_AddBook {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        RestTemplate template = new RestTemplate();

        Book book=new Book("add book",12341,"adding
                           book",1000,"description adding","abcd");
        book.setDescription("new description");
        Book book2= template.postForObject(
            "http://localhost:8081/Ch09_Spring_Rest_JDBC/books"
            ,book,Book.class);
        System.out.println(book2.getAuthor());
    }
}

```

The post method take `url for resource`, Object to add at resource and type of object as the arguments.

## 7. In the POSTMAN we can add the resource URL and select POST method as shown in the figure below:

The screenshot shows the Postman Builder interface. The URL for the resource is `http://localhost:8081/Ch09_Spring_Rest_JDBC/books`. The method is set to `POST`. The request body is set to `raw` type, specifically `JSON (application/json)`. The body content is a JSON object:

```

1 {
2   "ISBN": "13",
3   "bookName": "Book Name",
4   "publication": "publication1",
5   "price": "200",
6   "description": "new book",
7   "author": "author1"
8 }

```

The response status is `200 OK` and the time taken is `106 ms`.

- In the same way we will add a resource for getting all the books as shown below:

```

@GetMapping("/books")
public ResponseEntity getAllBooks() {

    List<Book> books = bookDAO.findAllBooks();
    return new ResponseEntity(books, HttpStatus.OK);
}

```

To test `getAllBook` add the Client code as follows:

```

public class Main_GetAll {

    public static void main(String[] args) {
        RestTemplate template = new RestTemplate();
        ResponseEntity<Book[]> responseEntity=

```

```

        template.getForEntity(
            "http://localhost:8081/Ch09_Spring_Rest_JDBC/book
Book[].class);
Book[] books=responseEntity.getBody();
for(Book book:books)
System.out.println(book.getAuthor()+"\t"+book.getISBN
}
}

```

The response is of type JSON contains array of books which we can get from the response body.

- Let's get the list from POSTMAN by adding the URL as [http://localhost:8081/Ch09\\_Spring\\_Rest\\_JDBC/books](http://localhost:8081/Ch09_Spring_Rest_JDBC/books) and selecting the GET method. We will get the list of books as JSON as shown in the snapshot:

The screenshot shows the Postman application interface. The top navigation bar includes 'Runner', 'Import', 'Builder' (which is selected), 'Team Library', and various status indicators. The main request configuration area shows a 'GET' method being sent to 'http://localhost:8081/Ch09\_Spring\_Rest\_JDBC/books'. The 'Body' tab is active, displaying a JSON response with three items. The response is formatted as follows:

```

1 [ 
2   { 
3     "bookName": "abc",
4     "publication": "my pub",
5     "price": 100,
6     "description": "desc",
7     "author": "abc",
8     "isbn": 12
9   },
10  { 
11    "bookName": "Book Name",
12    "publication": "publication1",
13    "price": 200,
14    "description": "new book",
15    "author": "author1",
16    "isbn": 98564567
17  },
18  { 
19    "bookName": "Book Name",
20    "publication": "Packt Publication",
21    "price": 200,
22    "description": "new book",
23    "author": "author1",

```

The status bar at the bottom indicates a 'Status: 200 OK' and a 'Time: 404 ms'.

In the same way we can write the method to delete the book by ISBN. You can find the code

## Presentation of the data

In the discussed demos we used JSON to present the resource but in practice the consumer may prefer other resource formats as XML, PDF, or HTML. Whatever the representation format the consumer wants to have the controllers are least bother about it. The Spring provides following two ways to deal with the response to transform it to representation state which client will be consumed.

- HTTP based message converters
- Negotiating view based rendering of the view.

### Http-based message converters

The controllers performs their main task of producing the data, this data will be presented in the view part. There are multiple ways to identify the view for representation, but a direct way is available where the object data returned from the controller is implicitly converted to appropriate presentation for the client. The job of implicitly converting is done by HTTP message converters. Following are the message converters provided by Spring which handles common conversion between the message and the java objects

- ByteArrayHttpMessageConverter - it converts byte arrays
- StringHttpMessageConverter - it converts Strings
- ResourceHttpMessageConverter - it converts org.springframework.core.io.Resource for any type of octet stream
- SourceHttpMessageConverter - it converts javax.xml.transform.Source
- FormHttpMessageConverter - it converts form data to/from the value of type MultiValueMap<String, String>.
- Jaxb2RootElementHttpMessageConverter - it converts Java objects to/from XML
- MappingJackson2HttpMessageConverter - it converts JSON
- MappingJacksonHttpMessageConverter - it converts JSON

- AtomFeedHttpMessageConverter - it converts Atom feeds
- RssChannelHttpMessageConverter - it converts RSS feeds
- MarshallingHttpMessageConverter - it converts XMLs

## Negotiating view based rendering of the view

We had already discussed Spring MVC in depth to handle the data and to present the data as well. The ModelAndView helps in setting the view name and the data to be bound in it. The view name then will be used by the front controller to target the actual view from its exact location with the help of ViewResolver. In Spring MVC only resolving the name and then bound data in it was more than sufficient but in RESTful web services we need much more than this. Here matching the view name alone is not sufficient but also choice of suitable view is also important. The view has to be matched to the representation state of the which is required by the client. If user needs JSON the view has to be selected which is able to render the obtained message to JSON.

Spring provides ContentNegotiatingViewResolver to resolve the views according to the content type which is required for the client. Following is the bean configuration which we need to add to select the views as

The configuration has reference to the ContentNegotiationManagerFactoryBean referred by 'cnManager'. We will do the configuration of it while discussing the demo. Here we configured two ViewResolvers one as PDF viewer and other for JSP's.

The very first thing checked from the request path is its extension to determine the media type. If no match found then FileTypeMap is used to get media type using the requested file name. If still the media type is not available then the accept header will be checked. Once the media type is known the whether the supported view resolver is available or not is checked. And if it's available then the request is delegated to the appropriate view resolver. While developing the custom view resolver we need to follow steps as follows:

1. Develop the custom view. This custom view will be child of

AbstractPdfView or AbstractRssFeedView or AbstractExcelView.

- According the view the ViewResolver implementation need to be written.
  - Register the custom view resolvers in the context.
- Let's generate a PDF file using custom ViewResolver and sample data step by step
2. Add handler mapping file boo-servlet.xml which will contain annotation configuration and configuration to discover the controllers. You can copy this from earlier applications.
  3. Add front controller in web.xml as we did in earlier application.
  4. Download and add itextpdf-5.5.6.jar for PDF files.
  5. Create Ch09\_Spring\_Rest\_ViewResolver as dynamic web application and add to it all required jars.
  6. Add MyBookController as a RestController to show list of books in com.packt.ch09.controller package. The method handles 'books/{author}' URL. The method has ModelMap as a parameter to allow the addition of the 'book list' model. Here we are adding a dummy list of the books but you can add the code to fetch the data from the database as well. The code will be as done below:

```
@RestController
public class MyBookController {
    @RequestMapping(value="/books/{author}", method =
        RequestMethod.GET)
    public String getBook(@PathVariable String author,
        ModelMap model)
    {
        List<Book> books=new ArrayList<>();
        books.add(new
            Book("Book1",101,"publication1",100,
                "description","auuthor1"));
        books.add(new Book("Book2",111,"publication1",200,
                "description","auuthor1"));
        books.add(new Book("Book3",121,"publication1",500,
                "description","auuthor1"));

        model.addAttribute("book", books);
        return "book";
    }
}
```

```
}
```

We will add the JSP view later with 'book' as the view name which is returned by the handler method.

7. Let's add PDFView which is a child of `AbstractPdfView` as shown by the code below:

```
public class PdfView extends AbstractPdfView {  
    @Override  
    protected void buildPdfDocument(Map<String, Object> mod  
        Document document, PdfWriter writer,  
        HttpServletRequest request, HttpServletResponse  
        response) throws Exception  
{  
    List<Book> books = (List<Book>) model.get("book");  
    PdfPTable table = new PdfPTable(3);  
    table.getDefaultCell().setHorizontalAlignment  
        (Element.ALIGN_CENTER);  
    table.getDefaultCell().  
        setVerticalAlignment(Element.ALIGN_MIDDLE);  
    table.getDefaultCell().setBackgroundColor(Color.light  
  
    table.addCell("Book Name");  
    table.addCell("Author Name");  
    table.addCell("Price");  
  
    for (Book book : books) {  
        table.addCell(book.getBookName());  
        table.addCell(book.getAuthor());  
        table.addCell("'" + book.getPrice());  
    }  
    document.add(table);  
  
}  
}
```

The `pdfBuildDocument()` method will design the look and feel of the PDF file as a Document with the help of `PdfTable`. The table heading and the data to display will be bounded by the `table.addCell()` methods.

8. Now let's add the `PdfViewResolver` which implements `ViewResolver` as follows:

```

public class PdfViewResolver implements ViewResolver{

    @Override
    public View resolveViewName(String viewName, Locale loc
        throws Exception {
        PdfView view = new PdfView();
        return view;
    }
}

```

9. Now we need to register the ViewResolvers to the context. It can be done by adding the ContentNegotiatingViewResolver bean as done in the configuration.
10. The ContentNegotiatingViewResolver bean refers to the ContentNegotiationManagerFactoryBean so let's add one more bean for it as follows:

```

<bean id="cnManager" class="org.springframework.web.ac
ContentNegotiationManagerFactoryBean">
    <property name="ignoreAcceptHeader" value="true" />
    <property name="defaultContentType" value="text/html" />
</bean>

```

11. We had added the custom view but we will also add the JSP page as its our default view. Let's add book.jsp under /WEB-INF/views. You can check the configuration of the InternalResourceViewResolver to get the exact location of the JSP page. The code is shown using following steps:

```

<html>
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core"%>
<title>Book LIST</title>
</head>
<body>
    <table border="1">
        <tr>
            <td>Book NAME</td>
            <td>Book AUTHOR</td>
            <td>BOOK PRICE</td>
        </tr>
        <tr>
            <td>${book.bookName}</td>
            <td>${book.author}</td>
            <td>${book.price}</td>
        </tr>
    </table>
</body>

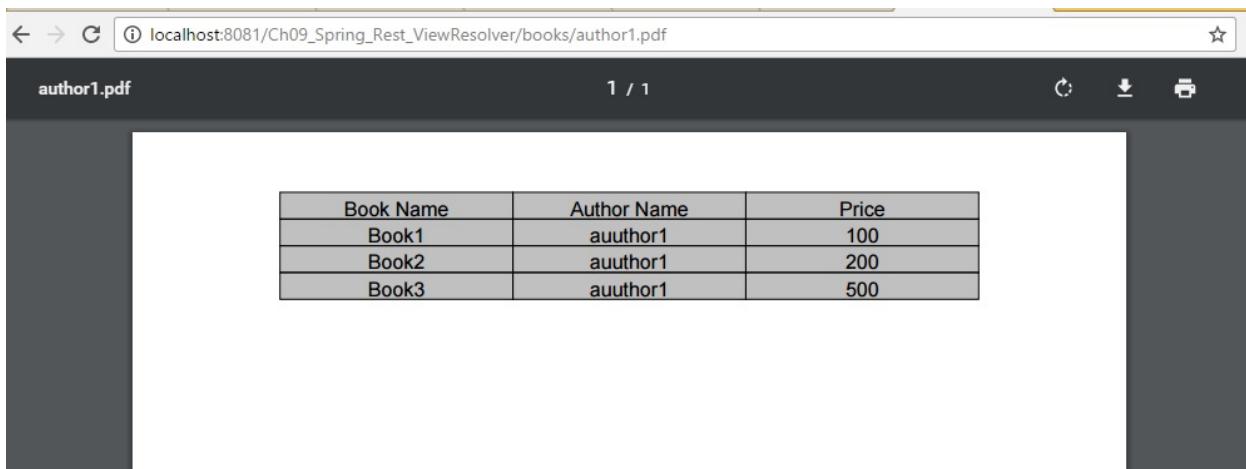
```

```
        </tr>
    </table>
</body>
</html>
```

12. Yes we had done with application now it's time to test the application.  
Run the application on the server and add the URL in the browser as  
[http://localhost:8080/Ch09\\_Spring\\_Rest\\_ViewResolver/books/author1.pdf](http://localhost:8080/Ch09_Spring_Rest_ViewResolver/books/author1.pdf)

The `author1` is the name of the author whose book list we want to fetch, the extension PDF shows what type of view is expected by the consumer.

We will get the following output in the browser:



A screenshot of a PDF viewer window titled "author1.pdf". The URL in the address bar is "localhost:8081/Ch09\_Spring\_Rest\_ViewResolver/books/author1.pdf". The page number "1 / 1" is visible. The content is a table with three columns: Book Name, Author Name, and Price. The data rows are:

Book Name	Author Name	Price
Book1	auuthor1	100
Book2	auuthor1	200
Book3	auuthor1	500

# Summary

---

In the beginning of the chapter we discussed about the web services and importance of the web services. We also discussed about SOAP and RESTful web services. We discuss in depth about how to write the RestController who handles URLs. The rest controller are revolving around the URL, we take a over view of how to design URLs to mapped to the handler methods. We developed a RestController which deals with database whenever the client request arrives for all CRUD methods. The RestTemplate an easy and less complex way to test RESTful web services has been discussed in depth for different types of HTTP methods. Moving one step ahead we also used POSTMAN application to test the developed web services. Developing the web services irrespective what consumer is looking for is one way traffic. We also explored about message converters and content negotiation to serve the consumer by different views.

In next chapter we will explore the most discussable topic and a new entry in Spring who is changing the web experience. We will discuss about WebSocket in next chapter.

# **Chapter 9. Exchange the Message: The Messaging**

Up till now we have discussed a lot about bidirectional web applications, which happens over the traditional HTTP communication. These browser based applications provide two way communication by opening multiple connections. The **websocket protocol** provides a mean for messaging over TCP which doesn't rely on the opening of multiple HTTP connections. In this chapter, we will discuss websocket protocol with the help of following points:

- **Introduction to messaging**
- **Introduction to WebSocket protocol**
- **WebSocket API**
- **Overview of STOMP**

In web applications the bidirectional communication between client and server happens to be synchronous where the client request the resource and server sends the notification as the HTTP call. It addresses the following problems:

- The multiple connections have to open to send the information and to collect the incoming messages
- The track of mapping outgoing connections to the incoming connections so as to track the request and its replies

The better solution will be to maintain a single TCP connection for both sending as well as receiving which has been provided by WebSocket as low level protocol without headers. As headers are not getting added, the amount of data being transmitted over the network decreases, in turn the load reduces. It is done by the process known as pull technology instead of push technology done in long pulling as in **AJAX**. Now a days, developers are using **XMLHttpRequest (XHR)** for the asynchronous HTTP communication. WebSocket uses the HTTP as transport layer to support the existing infrastructure using 80, 443 ports. In such two way communication

of successful connection data transfer happens independently on their will.

The RFC 6455 defines the WebSocket protocol as, one which facilitates the two way communication between the client and the server where the client is running in the controlled environment communicating to the remote host, who had given the permission to accept mail, email or any direct communication from the code. The protocol consists of opening the handshake followed by the basic message framing which is layered over the TCP protocol. The HTTP status code 101 is sent by the server, if it agrees denoting successful handshake. Now the connection will remain open and the message exchange can be done. The following diagram gives an idea how the communication happens:



The WebSocket does the following things on top of TCP:

- It adds the web security model to the browser.
- As one port needs to support multiple host names and multiple services, it adds addressing and naming mechanism to provide such support.
- It forms a layer framing mechanism on top of TCP to facilitate IP packet mechanism.
- A closing handshake mechanism.

The data transfer in WebSocket uses a sequence of the frames. Such data frames can be transmitted by either side at any time after opening the handshake, but before the endpoint has sent the Close frame.

# Spring and Messaging

---

From Spring 4.0 onwards, there is a support for WebSocket introducing spring-websocket module, which is compatible with Java WebSocket API(JSR-356). The HTTPServlet and REST application uses URLs, HTTP methods to exchange the data between the client and the server. But opposite to this the WebSocket application may use single URL for the initial handshakes which is asynchronous, messaging and even driven architecture as JMS or AMQP. Spring 4 includes spring-messaging module to integrate Message, MessageChannel, MessageHandler, a set of annotations for mapping messages to the methods and many more to support the basic messaging architecture. The `@Controller` and `@RestController` which we already used to create Spring MVC web application and RESTful web services which allows handling HTTP request also support handler methods for WebSocket messaging. Also the handler methods from Controllers can broadcasts the message to either all interested or user specific WebSocket client.

## The Use

The WebSocket architecture is suitable in all those web applications, which needs to exchange events frequently but where the time with which the data exchanged to the target matters as:

- The social media is playing a very important role now days and playing a vital role to be in touch with family and friends. The user always enjoys real time updates of the feed done by their circle.
- Now a day's many online multiplayer games are available on web. In such games, each player is always keen to know what his opponent is doing. Nobody wants to discover opponents move when they took their action.
- In development the version control tools like Tortoise SVN, Git helps to keep the track of the files. So that, the exchange of the code becomes

easier without conflict. But here, we won't get information about who is working on which of the file at real time.

- In financial investments one always wants to know the real time price of the company in which he is interested and the not the one before some time.

# Overview of WebSocket API

---

The spring framework facilitates creation of WebSocket by providing the APIs to adopt various WebSocket engines. Today Tomcat7.0.47+, Jetty 9.1+, WebLogic 12.1.3+, GlassFish 4.1+ provides runtime environments for WebSocket.

## Creation of WebSocket handler

The creation of `WebSocketHandler` can be done either by implementing the `WebSocketHandler` interface or extending from `TextWebSocketHandler` or `BinaryWebSocketHandler` as we have done in the following snippet:

```
public class MyWebSocketHandler extends TextWebSocketHandler{  
    @Override  
    public void handleTextMessage(WebSocketSession session,  
        TextMessage message)  
    {  
        // code goes here  
    }  
}
```

The `WebSocketDecorator` class can be used to decorate the `WebSocketHandler`. Spring provides some decorator classes to handle exceptions, logging mechanism, and handling binary. The class `ExceptionWebSocketHandler`, is an Exception handling `WebSocketHandlerDecorator` which helps in handling all instances of the `Throwable`. The `LoggingWebSocketHandlerDecorator`, adds the logging to events which takes place in the life cycle of the `WebSocket`.

## Registering `WebSocketHandler`

The `WebSocket` handlers are mapped to a specific URL to register this mapping .The framework can be done either by Java configuration or XML based configuration.

## Java Based configuration

The `WebSocketConfigurer` is used to map the handler with its specific URL in the `registerWebSocketHandlers()` method as shown in the code below:

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry
        registry)
    {
        registry.addHandler(createHandler(), "/webSocketHandler");
    }
    @Bean
    public WebSocketHandler createMyHandler() {
        return new MyWebSocketHandler();
    }
}
```

Here our `WebSocketHandler` is mapped to `/webSocketHandler` URL.

The customization of the `WebSocketHandler` to customize the handshake can be done as:

```
@Configuration
@EnableWebSocket
public class MyWebSocketConfig implements WebSocketConfigurer {
    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry
        registry)
    {
        registry.addHandler(createHandler(),
            "/webSocketHandler").addInterceptors
            (new HttpSessionHandshakeInterceptor());
    }
    @Bean
    public WebSocketHandler createMyHandler() {
        return new MyWebSocketHandler();
    }
}
```

The `HandshakeInterceptor` exposes `beforeHandshake()` and `afterhandshake()` methods to customize the `WebSocket` handshake. The

`HttpSessionHandshakeInterceptor` facilitates binding information from `HttpSession` to the handshake attributes under the name `HTTP_SESSION_ID_ATTR_NAME`. These attribute can be used as `WebSocketSession.getAttributes()` method.

## XML Based configuration

The registration done in the above Java snippet, can be done in XML as well. We need to register the web-socket namespace in the XML and then configure the handler as shown below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:websocket=
http://www.springframework.org/schema/websocket
       xsi:schemaLocation=
         "http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/websocket
          http://www.springframework.org/schema/websocket/spring-
             websocket.xsd">
<websocket:handlers>
  <websocket:mapping path="/myWebSocketHandler"
    handler="myWebSocketHandler"/>
  </websocket:handlers>
<bean id="myWebSocketHandler"
  class="com.packt.ch10.WebsocketHandlers. MyWebSocketHandler"
  />
</beans>
```

The customised `WebSocketConfigurer` in XML can be written as follows:

```
<websocket:handlers>
  <websocket:mapping path="/myWebSocketHandler"
    handler="myWebSocketHandler"/>
  <websocket:handshake-interceptors>
    <bean class=
      "org.springframework.web.socket.server.support.
       HttpSessionHandshakeInterceptor"/>
  </websocket:handshake-interceptors>
  </websocket:handlers>
  <!-bean for MyWebSocketHandler -à
</beans>
```

## WebSocket Engine Configuration

Tomcat 7.0.47+, Jetty 9.1+, WebLogic 12.1.3+, GlassFish 4.1+ provides runtime environments for WebSocket. The characteristics such as buffer size of the messages, timeout, can be configured for Tomcat runtime environment by adding the bean for `WebSocketConfigurer` as follows:

```
@Bean
public ServletServerContainerFactoryBean
    createWebSocketContainer()
{
    ServletServerContainerFactoryBean webSocketcontainer =
        new ServletServerContainerFactoryBean();
    webSocketcontainer.setMaxTextMessageBufferSize(9000);
    webSocketcontainer.setMaxBinaryMessageBufferSize(9000);
    return webSocketcontainer;
}
```

The equivalent XML configuration can be written as:

```
<bean class= "org.springframework.web.socket.server.standard.
    ServletServerContainerFactoryBean">
    <property name="maxTextMessageBufferSize" value="9000"/>
    <property name="maxBinaryMessageBufferSize" value="9000"/>
</bean>
```

### Configuration of allowed origins

The 'origin' is the scope of the privilege by the agent. The variety of content exists in various format created by numerous authors, something out of which may be harmful. The content created by one origin can freely interact with the content created by other origin. The agents has the facility to set up the rules under which one content interact with other called as '**same-origin policy**'.

Let's take an example of HTML, where we are having form submission. Whenever the user agent enters the data, the entered data is exported to the URI. Here the URI declares the trust on the integrity of the information which the script file has received through the URI.

`http://packt.com/`, `http://packt.com:8080/`, `http://www.packt.com/`,  
`https://packt.com:80/`, `https://packt.com/`, `http://packt.org/` are  
different URIs.

There are three ways to configure origin as:

- Allow same origin
- Allow specified list of origins
- Allow all origins

Let's first discuss in detail about the creation and use of WebSocket for client server communication:

## 1. Creation of WebSocket:

```
WebSocket socket= new WebSocket( URL, protocols);
```

- Where URL contains:
  - **Schema:** the URL must contain either `ws` denoting insecure connection or `wss` denoting secure connections
  - **Host:** it's a name or IP of the server
  - **Port:** the remote port where you want to get connected ws connection by default uses port '80' and wss uses 443
  - **Resource name:** path URL of the resource to fetch
- We can write the URL for WebSocket as:
  - `scheme://host_name:port_no/resource_path`
  - `ws://host_name:port_no/resource_path`
  - `wss://host_name:port_no/resource_path`

## 2. Closing the WebSocket:

To close the connection we use `close()` method as `close(code, reason)`.

## Note

`code:` It's a numeric status sent to the server. 1000 indicates normal closing of connections.

### 3. States of the WebSocket:

Following are the connection states of the WebSocket, giving information in which state it is:

- **Connecting:** The WebSocket is constructed and it is attempting to connect to the specified URL. This state is considered as the connecting state having ready State as 0.
- **Open:** Once the WebSocket is successfully connected to the URL it will enter to the open state. The data can be sent to and from the network only when the WebSocket is in open state. The ready state value of open state is "1".
- **Closing:** The WebSocket won't directly close, it must communicate to the server to inform it is disconnecting. This state is considered as closing state. The ready state value of open state is "2".
- **Closed:** After the successful disconnection from the server the WebSocket enters in the closed state. The WebSocket in the closed state has a "readyState" value of 3.

### 4. Event Handling in WebSocket:

The WebSocket works on the principle of event handling, where the call back methods get invoked to complete the process. Following are the events which occurs in the life cycle of the WebSocket:

- **onopen:** When the WebSocket transit to open state the "onopen" event handler gets called.
- **onmessage:** When the WebSocket receives data from the server the "onmessage" event handler gets called. The received data gets stored in the "data" field of the "message" event.

The data field has the parameters as:

- **onclose:** When the WebSocket is closed the "onclose" event handler gets called. The event object will get passed to "onclose". It has three fields named:
- **code:** a numeric status value provided by the server
- **reason:** its a string describing the close event.

- **wasClean**: has a boolean value indicating whether the connection closed without any problem. Under normal circumstances, "wasClean" is true.
- **onerror**: When a WebSocket encounters any problem the "onerror" event handler gets called. The event passed to the handler will be a standard error object which includes "name" and "message" fields.

## 5. Sending the data:

The data transmission happens via `send()` method which deals with UTF-8 text data, data of type `ArrayBuffer` and a data of type `blob`. The 'bufferedAmount' property with value as zero ensures data sent successfully.

Let us develop a demo for WebSocket with the help of following steps to find capital of the country:

1. Create Ch10\_Spring\_Message\_Handler as dynamic web application.
2. Add jars for Spring core, Spring web , spring-websocket, spring-messaging modules. Also add jars for Jackson.
3. Let's add MyMessageHandler as a child of `TextWebSocketHandler` in the `compackt.ch10.config` package. Override the methods for handling message, WebSocket connection, connection closing as shown below:

```
public class MyMessageHandler extends TextWebSocketHandle
{
    List<WebSocketSession> sessions = new CopyOnWriteArrayList();

    @Override
    public void handleTextMessage(WebSocketSession session,
        TextMessage message) throws IOException {
        String country = message.getPayload();
        String reply="No data available";
        if(country.equals("India")) {
            reply="DELHI";
        }
        else if(country.equals("USA")) {
            reply="Washington, D.C";
        }
        System.out.println("hanlding message");

        for(WebSocketSession webSsession:sessions){
            session.sendMessage(new TextMessage(reply));
        }
    }
}
```

```

    }
    @Override
    public void afterConnectionEstablished(WebSocketSession
        session) throws IOException {
        // Handle new connection here
        System.out.println("connection established:hello");
        sessions.add(session);
        session.sendMessage(new TextMessage("connection
            established:hello"));
    }
    @Override
    public void afterConnectionClosed(WebSocketSession sess
        CloseStatus status) throws IOException {
        // Handle closing connection here
        System.out.println("connection closed : BYE");
    }
    @Override
    public void handleTransportError(WebSocketSession sessi
        Throwable exception) throws IOException {
        session.sendMessage(new TextMessage("Error!!!!!!"));
    }
}

```

This MessageHandler needs to register with WebSocketConfigurer for URL '/myHandler' for all origins as shown below:

```

@Configuration
@EnableWebSocket
public class MyWebSocketConfigurer extends
WebMvcConfigurerAdapter implements WebSocketConfigurer
{
    @Override
    public void
    registerWebSocketHandlers(WebSocketHandlerRegistry
        registry) {
        registry.addHandler(myHandler(),
            "/myHandler").setAllowedOrigins("*");
    }
    @Bean
    public WebSocketHandler myHandler() {
        return new MyMessageHandler();
    }
    // Allow the HTML files through the default Servlet
    @Override
    public void configureDefaultServletHandling
        (DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }
}

```

```
    }
}
```

4. Add Front controller mapping in web.xml as we did in earlier applications with servlet name as 'books'.
5. Add books-servlet.xml to add bean for 'viewResolver'. You can decide adding it as a bean depending upon the application requirement.
6. Also add configuration to enable spring web MVC as:

```
<mvc:annotation-driven />
```

7. Add country.jsp as a JSP page having list of countries where user can select the country from drop down to get name of its capital:

```
<div>
    <select id="country">
        <option value="India">INDIA</option>
        <option value="USA">U.S.A</option>
    </select><br>
    <br> <br>
    <button id="show" onclick="connect() ;">Connect</button>
    <br /> <br />
</div>
<div id="messageDiv">
    <p>CAPITAL WILL BE DISPLAYED HERE</p>
    <p id="msgResponse"></p>
</div>
</div>
```

8. Add the SockJS support by adding sockjs-0.3.4.js in your resources or adding the following code:

```
<script type="text/javascript"
src="http://cdn.sockjs.org/sockjs-0.3.4.js"></script>
```

9. On form submission a method of JavaScript gets invoked, where we handle the WebSocket events onopen, onmessage etc as discussed earlier:

```
<script type="text/javascript">
    var stompClient = null;
    function setConnected(connected) {
        document.getElementById('show').disabled = connected;
```

```

        }
        function connect() {
            if (window.WebSocket) {
                message = "supported";
                console.log("BROWSER SUPPORTED");
            } else {
                console.log("BROWSER NOT SUPPORTED");
            }
            var country = document.getElementById('country').value;
            var socket = new WebSocket(
                "ws://localhost:8081/Ch10_Spring_Message_Handler
                /webS/myHandler");
                socket.onmessage=function(data) {
                    showResult("Message Arrived"+data.data)
                };
                setConnected(true);
                socket.onopen = function(e) {
                    console.log("Connection established!");
                    socket.send(country);
                    console.log("sending data");
                };
            }
        function disconnect() {
            if (socket != null) {
                socket.close();
            }
            setConnected(false);
            console.log("Disconnected");
        }
        function showResult(message) {
            var response = document.getElementById('messageDiv');
            var p = document.createElement('p');
            p.style.wordWrap = 'break-word';
            p.appendChild(document.createTextNode(message));
            response.appendChild(p);
        }
    </script>

```

We already discussed about how to write the WebSocket URLs and event handling mechanism.

Deploy the application and access the page. Select the country from dropdown and click on show capital button. The message will appear displaying the name of the capital.

The following diagram shows the flow of the application:

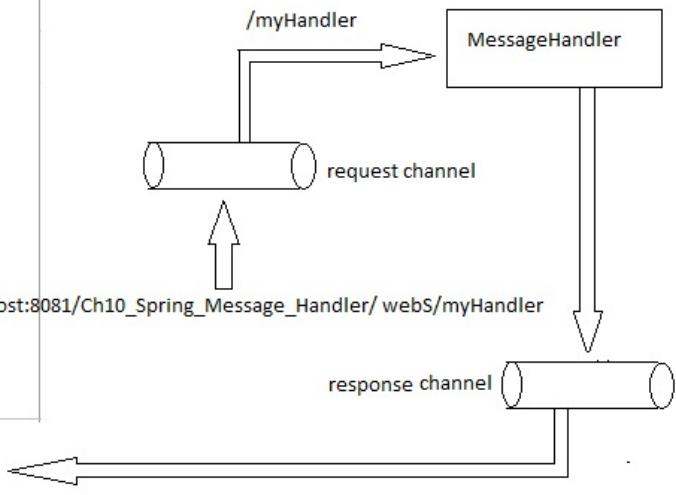
① [localhost:8081/Ch10\\_Spring\\_Message\\_Handler/country.jsp](http://localhost:8081/Ch10_Spring_Message_Handler/country.jsp)

## WebSocket

### Select Country to Know it's CAPITAL

ws://localhost:8081/Ch10\_Spring\_Message\_Handler/webS/myHandler

CAPITAL WILL BE HERE



We had added the Console logs as well as the Alert messages to know the progress and to and fro of the messages. As per requirement you can customize it or can completely omit as well.

In the earlier example, we have used WebSocket for communication but still its support is limited. The SockJS is a JavaScript library which provides the objects like WebSocket.

# SockJS

---

The SockJS library provides cross browser, JavaScript API to enable low latency, cross domain communication between the browser and server. It aims to support the following goals:

- Instead of using WebSocket instance, the SockJS instance is used
- The API which are close to WebSocket API both for server as well as client side APIs
- Faster communication support
- JavaScript for client side
- It comes with some chosen protocols which supports cross domain communication

The following code shows how to enable in the SockJS support for `WebSocketConfigurere` as:

```
@Override  
public void registerWebSocketHandlers(WebSocketHandlerRegistry  
    registry)  
{  
    registry.addHandler(myHandler(),  
        "/myHandler_sockjs").setAllowedOrigins("*").withSockJS();  
}
```

Or even we can configure in XML as:

```
<websocket:handlers>  
    <websocket:mapping path="/myHandler"  
        handler="myHandler_sockjs"/>  
    <websocket:sockjs/>  
</websocket:handlers>
```

We can update the Capital demo developed earlier to support SockJS as follows:

1. Add `country_sockjs.jsp` in `WebContent` to use with SockJS as follows:

```
var socket = new SockJS(  
    "http://localhost:8080/Ch10_Spring_Message_Handler  
/webS/myHandler_sockjs");
```

2. Add MyWebSocketConfigurer\_sockjs in com.packt.ch10.config package to configure the WebSocket as we did earlier. To enable the SockJS support we have to modify the registerWebSocketHandlers() method as shown in the configuration above using withSockJS().
3. Run the application and request for country\_sockjs.jsp to use the SockJS. You can observe the console logs as well.

In the above example, we had used WebSocket to get the connection and handle the events. The new WebSocket protocol also has been introduced for the communication which we used here. It uses less bandwidth. It has no headers like HTTP gives simpler, efficient communication. We can also use STOMP for the communication.

# STOMP

---

**Simple (or Streaming) Text Oriented Message Protocol (STOMP)** over WebSocket provides a straightforward mapping from a STOMP frame to a JavaScript object. WebSocket is fastest protocol but, still it is not supported by all browsers. The browsers have problems to support proxies and protocol handling. It will take a while to get wide support by all the browsers, meanwhile we need to find some substitute or real time solution. The SockJS supports STOMP protocol for communicating with any message broker from the scripting languages and is an alternative to AMQP. STOMP is lightweight and easy to implement both on client as well as server side. It comes with reliable sending single message and then disconnect or consume all messages from the destination. It defines following different frames that are mapped to WebSocket frames:

- **CONNECT**: It connects the client to the server.
- **SUBSCRIBE**: It is used to register which can listen to the given destination.
- **UNSUBSCRIBE**: It is used to remove existing subscription.
- **SEND (messages sent to the server)**: The frame sends a message to the destination.
- **MESSAGE (for messages send from the server)**: It conveys the messages from the subscriptions to the client.
- **BEGIN**: It starts the transaction.
- **COMMIT**: It commits the ongoing transaction.
- **ABORT**: It rollbacks the ongoing transaction.
- **DISCONNECT**: It disconnects the client from the server.

It also supports the following standard headers:

- **content-length**: The SEND, MESSAGE and ERROR frames contain content-length header having its value as content length of the message body.
- **content-type**: The SEND, MESSAGE and ERROR frames contain

content-type. It is similar to MIME type in web technology.

- **receipt:** The CONNECT frame may contain receipt as header attribute to acknowledge the server of the RECEIPT frame.
- **heart-beat:** It got added by the CONNECT and CONNECTED frames. It contains two positive integer values separated by the comma.
- 1st value represents outgoing heart beats. '0' specifies it cannot send heart beats.
- 2nd value denotes incoming heart beats. '0' denotes unwillingness to receive the heart beats.

## Spring STOMP support

The Spring WebSocket application works as a STOMP broker to all the clients. Each message will be routed through the Spring controllers. These controllers are capable of handling HTTP request and response by `@RequestMapping` annotation. Similarly they are capable of handling WebSocket Messages in all those methods who are annotated by `@Messaging`. Spring also facilitates integration of RabbitMQ, ActiveMQ as the STOMP brokers for the message broad casting.

Let's us develop an application to use STOMP step by step:

1. Create Ch10\_Spring\_Messaging\_STOMP as a dynamic web application and add the jars which we added earlier.
2. Add mapping for DispatcherServlet in web.xml having books as name and 'webS' as URL pattern.
3. Add books-servlet.xml to register bean for 'viewResolver'. Registration to discover the controllers and to consider all MVC annotations.
4. Add WebSocketConfig\_custom as a class in com.packt.ch10.config package to add the '/book' as endpoint enabled for SockJS. '/topic' as SimpleBroker for '/bookApp' as prefix. The code is as shown below:

```
@Configuration  
@EnableWebSocketMessageBroker  
public class WebSocketConfig_custom extends  
AbstractWebSocketMessageBrokerConfigurer {  
    @Override  
    public void configureMessageBroker(  
        ...)
```

```

        MessageBrokerRegistry config) {
    config.enableSimpleBroker("/topic");
    config.setApplicationDestinationPrefixes("/bookApp");
}
@Override
public void registerStompEndpoints(
    StompEndpointRegistry registry) {
    registry.addEndpoint("/book").withSockJS();
}
}

```

The `@EnableWebSocketMessageBroker` enables the class to act as a message broker.

5. Add POJO MyBook with bookName as data member in `com.packt.ch10.model` package.
6. Similarly add Result having result as data member as POJO having `getOffer` method as:

```

public void getOffer(String bookName) {
    if (bookName.equals("Spring 5.0")) {
        result = bookName + " is having offer of having 20% off";
    } else if (bookName.equals("Core JAVA")) {
        result = bookName + " Buy two books and get 10% off";
    } else if (bookName.equals("Spring 4.0")) {
        result = bookName + " is having for 1000 till month end";
    }
    else
        result = bookName + " is not available on the list";
}

```

7. Add `index.html` to have the link for the '`bookPage`' from the controller as follows:

```

<body>
    <a href="webS/bookPage">CLICK to get BOOK Page</a>
</body>

```

8. Add `WebSocketController` class in `com.packt.ch10.controller` package and annotate it by `@Controller("webs")`.
9. Add `bookPage()` method annotated by `@RequestMapping` to send `bookPage.jsp` to the client as shown below:

```

@Controller("/webS")
public class WebSocketController {
    @RequestMapping("/bookPage")
    public String bookPage() {
        System.out.println("hello");
        return "book";
    }
}

```

- Add bookPage.jsp in the jsps folder. The page will display book names to get offers associated with them. The code will be as follows:

```

<body>
<div>
    <div>
        <button id="connect"
            onclick="connect();">Connect</button>
        <button id="disconnect" disabled="disabled"
            onclick="disconnect();">Disconnect</button><br/>
    </div>
    <div id="bookDiv">
        <label>SELECT BOOK NAME</label>
        <select id="bookName" name="bookName">
            <option> Core JAVA </option>
            <option> Spring 5.0 </option>
            <option> Spring 4.0 </option>
        </select>
        <button id="sendBook" onclick="sendBook();">Send
        <p id="bookResponse"></p>
    </div>
</div>
</body>

```

- We will be handling the call back methods, once the client click the button. Add the scripts for sockjs and STOMP as:

```

<script type="text/javascript"
    src="http://cdn.sockjs.org/sockjs-0.3.4.js"></script>
    src="https://cdnjs.cloudflare.com/ajax/libs/stomp.js/2.3
stomp.js"/>

```

- Now we will add connect, disconnect, send, subscribe one by one. Let us first add connect method to get the STOMP connection as follows:

```

<script type="text/javascript">
    var stompClient = null;

```

```

function connect() {
    alert("connection");
    if (window.WebSocket) {
        message="supported";
        console.log("BROWSER SUPPORTED");
    } else {
        console.log("BROWSER NOT SUPPORTED");
    }
    alert(message);
    var socket = new SockJS('book');
    stompClient = Stomp.over(socket);
    stompClient.connect({}, function(frame) {
        alert("in client");
        setConnected(true);
        console.log('Connected: ' + frame);
        stompClient.subscribe('/topic/showOffer',
            function(bookResult) {
                alert("subscribing");
                showResult(JSON.parse(bookResult.body).result); }));
    });
}

```

The connect method creates an object of SockJS and using `Stomp.over()` adds support for STOMP protocol. The connection adds the `subscribe()` to subscribe the messages from '`/topic/showOffer`' handler. We had added '`/topic`' as the SimpleBroker in the `WebSocketConfig_custom` class. We are handling, sending and receiving the JSON objects. The offers received by the Result JSON object will be in form of result: `value_of_offer`.

13. Add disconnect method as:

```

function disconnect() {
    stompClient.disconnect();
    setConnected(false);
    console.log("Disconnected");
}

```

14. Add `sendBook` to sent the request to get the offer as:

```

function sendBook()
{
    var bookName =
    document.getElementById('bookName').value;
    stompClient.send("/bookApp/book", {}, 
        JSON.stringify({ 'bookName': bookName }));
}

```

```
}
```

The `send()` gives request to the handler `/bookApp/book`, which will accept JSON object having `bookName` data member. We registered destination prefix as '`bookApp`' which we are using while sending the request.

15. Add method to display the offer as:

```
function showResult(message) {  
    //similar to country.jsp  
}
```

16. Now let us add the handler method in controller for '`/book`'. This method will be annotated by `@SendTo("/topic/showOffer")` as below:

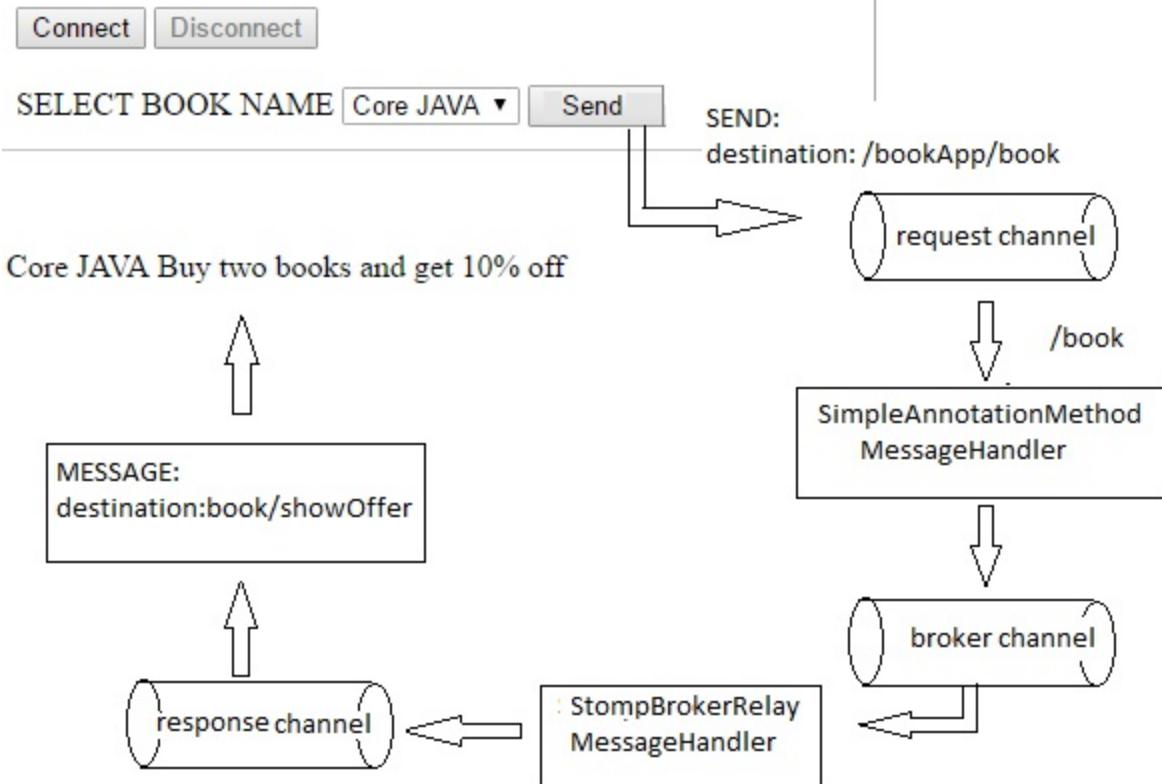
```
@MessageMapping("/book")  
@SendTo("/topic/showOffer")  
public Result showOffer(MyBook myBook) throws Exception  
    Result result = new Result();  
    result.getOffer(myBook.getBookName());  
    return result;  
}
```

17. Deploy the application. And click on the link to get offer page.
18. Click on connect to get server connection. Select the book to know offer and click on send. The offers associated with the book will get displayed.

The following diagram explains the application flow:

① localhost:8081/Ch10\_Spring\_Messaging\_STOMP/webS/bookPage

## WebSocket Book Offers



On console the log will be displayed as below showing different frames of STOMP:

Elements Console Sources Network Timeline Profiles Application

```
Web Socket Opened...
>>> CONNECT
accept-version:1.1,1.0
heart-beat:10000,10000

<<< CONNECTED
version:1.1
heart-beat:0,0

connected to server undefined
Connected: CONNECTED
heart-beat:0,0
version:1.1

>>> SUBSCRIBE
id:sub-0
destination:/topic/showOffer

>>> SEND
destination:/bookApp/book
content-length:25

{"bookName":"Spring 5.0"}

<<< MESSAGE
destination:/topic/showOffer
content-type:application/json;charset=UTF-8
subscription:sub-0
message-id:_xr9_jiz-0
content-length:57

{"result":"Spring 5.0 is having offer of having 20% off"}
```

# Summary

---

In this chapter we discussed in depth about the messaging using WebSocket. We took overview of why WebSocket is important and how it differs from the traditional web applications as well as the XMLHttpRequest based AJAX applications. We discussed the areas in which WebSocket can play a vital role. Spring provides API to work with WebSocket. We had seen `WebSocketHandler`, `WebSocketConfigurer` and its registration both using Java classes as well as XML based configurations using Capital of Country application. The SockJS library provides cross browser, JavaScript API to enable low latency, cross domain communication between the browser and server. We enabled the SockJS both in XML and Java configuration. We had also seen in depth about STOMP to be used in WebSocket over SockJS and to enable it and its event handling methods.

In the next chapter, we will discover the reactive web programming.



If you have any feedback on this eBook or are struggling with something we haven't covered, let us know at survey [link](#).

If you have any concerns you can also get in touch with us at

[customercare@packtpub.com](mailto:customercare@packtpub.com)

We will send you the next chapters when they are ready.....!

Hope you like the content presented.