config)

- Spring & Struts Integration (XML)
- Spring & Struts Integration (Java config)
- 14 Tips for Writing Spring MVC Controller

Understanding Spring MVC

Written by Nam Ha Minh
Last Updated on 24 June 2019 | Print Email

Spring framework makes the development of web applications very easy by providing the *Spring MVC* module. Spring MVC module is based on two most popular design patterns - *Front controller* and *MVC*.

In this article, firstly we learn about the *Front controller* and *MVC* design pattern and then explore the details of Spring MVC module in detail, its architecture, and various components and finally we build a simple web application using Eclipse IDE.

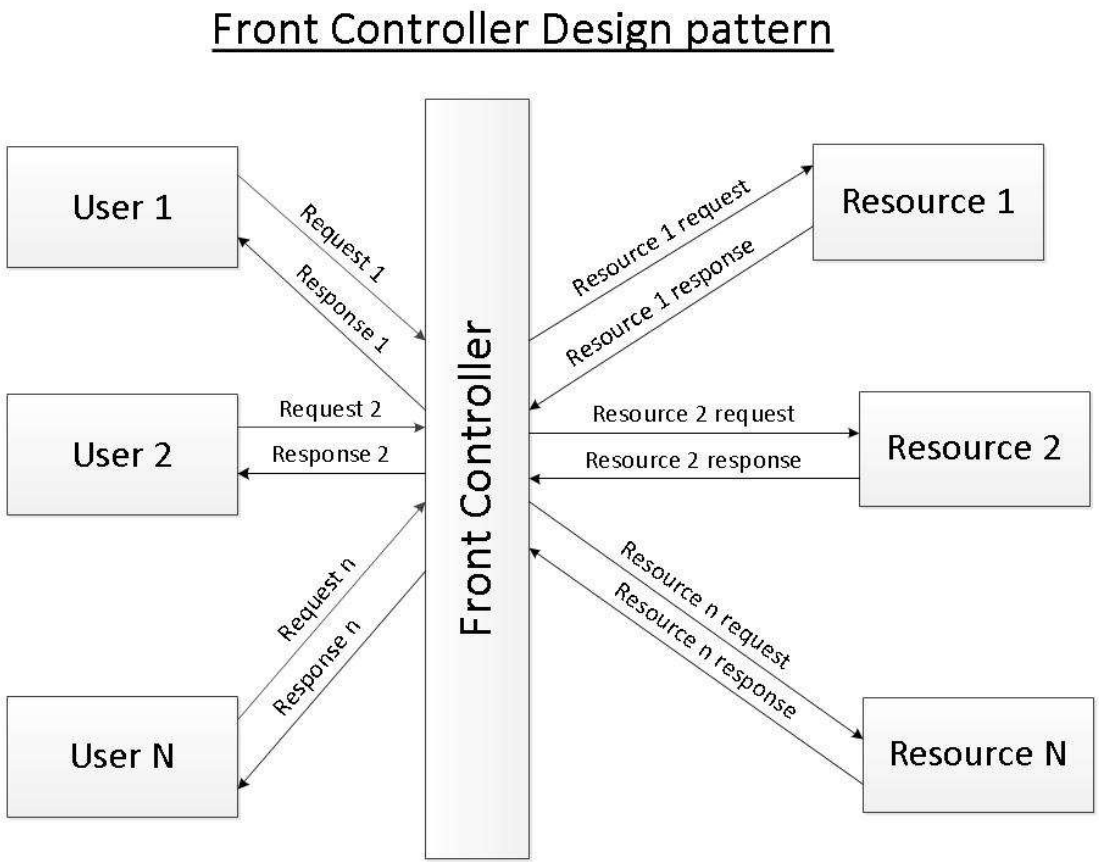
Table of Content

1. Spring MVC Architecture

Before going into details of Spring MVC architecture, let us first look at the two popular design patterns used for web development.

Front Controller design pattern

This design pattern enforces a single point of entry for all the incoming requests. All the requests are handled by a single piece of code which can then further delegate the responsibility of processing the request to further application objects.



Front Controller Design Pattern

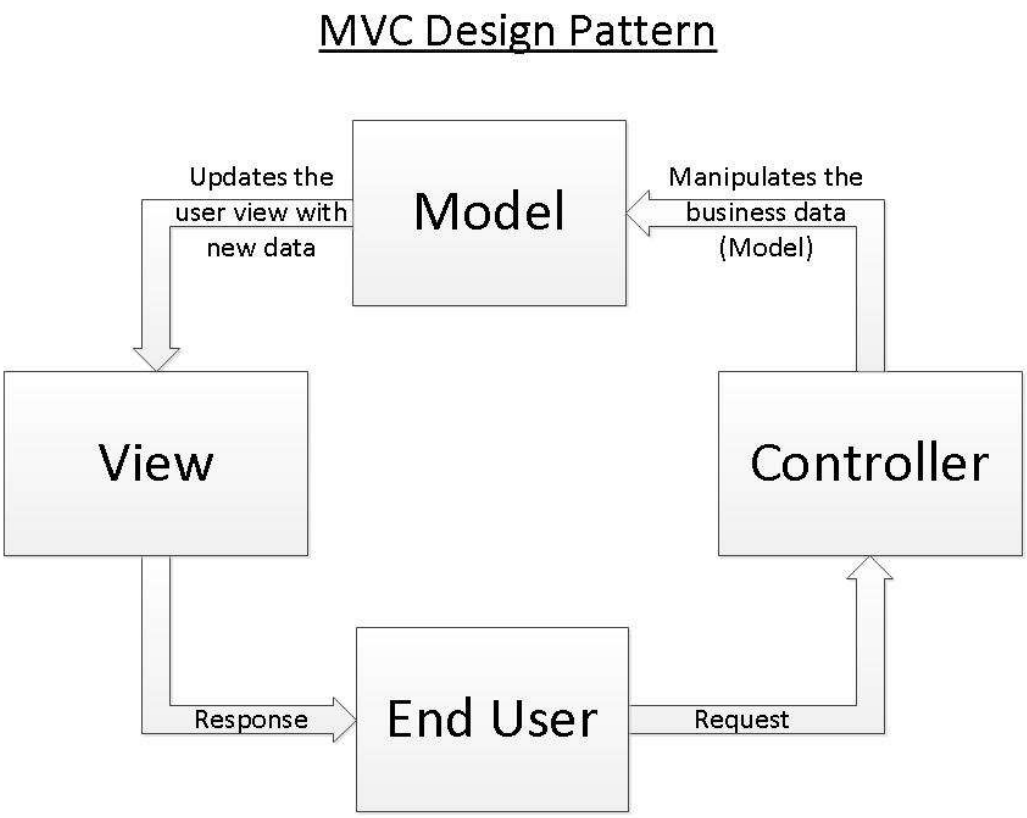
MVC design pattern

This design pattern helps us develop loosely coupled application by segregating various concerns into different layers. MVC design pattern enforces the application to be divided into three layers, *Model*, *View* and *Controller*.

Model: This represents the application data.

View: This represents the application's user interface. View takes model as the input and renders it appropriately to the end user.

Controller: The controller is responsible for handling the request and generating the model and selecting the appropriate view for the request.

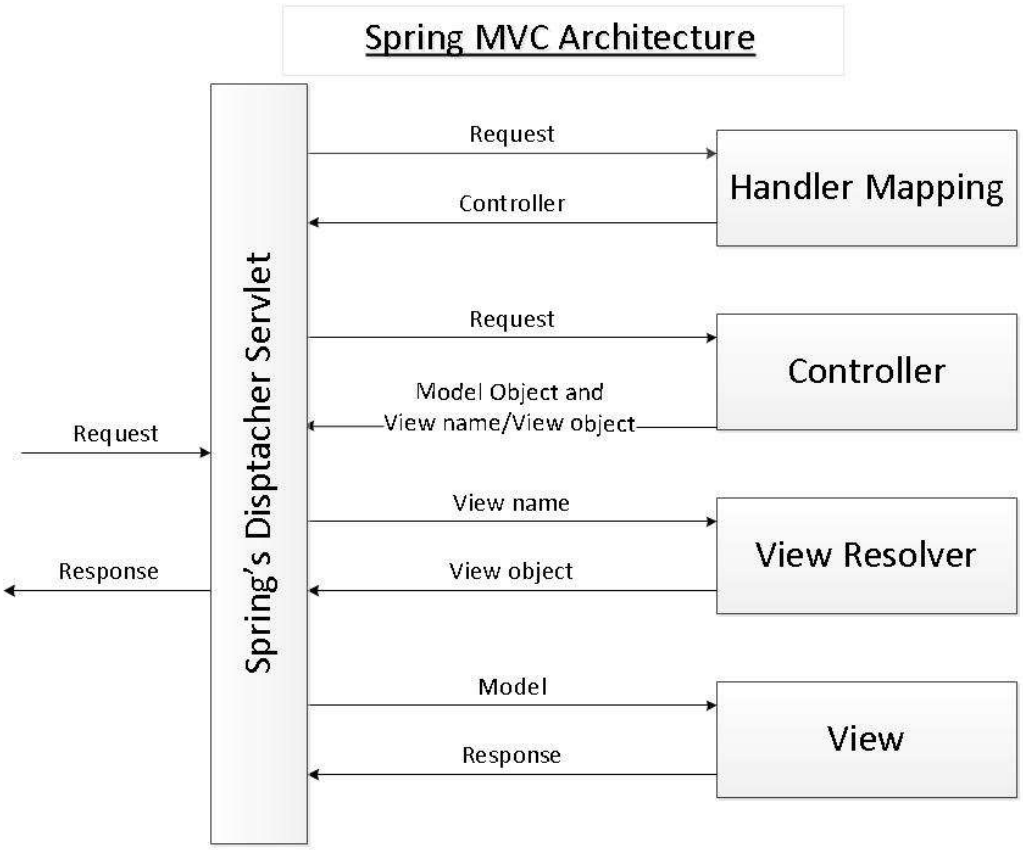


MVC Design Pattern

Spring's MVC module

Spring's MVC module is based on front controller design pattern followed by MVC design pattern. All the incoming requests are handled by the single servlet named `DispatcherServlet` which acts as the *front controller* in Spring's MVC module. The `DispatcherServlet` then refers to the `HandlerMapping` to find a controller object which can handle the request. `DispatcherServlet` then dispatches the request to the controller object so that it can actually perform the business logic to fulfil the user request. (Controller may delegate the responsibility to further application objects known as service objects). The controller returns an encapsulated object containing the model object and the view object (or a logical name of the view). In Spring's MVC, this encapsulated object is represented by class `ModelAndView`. In case `ModelAndView` contains the logical name of the view, the

DispatcherServlet refers the ViewResolver to find the actual View object based on the logical name. DispatcherServlet then passes the model object to the view object which is then rendered to the end user.



Spring MVC Architecture

2. Dispatcher Servlet

DispatcherServlet acts as the front controller in the Spring's MVC module. All the user requests are handled by this servlet. Since this is like any other servlet, it must be configured in the application's web deployment descriptor file i.e. web.xml.

```
<web-app xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
id="WebApp_ID"
version="2.5"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <display-name>Library</display-name>
   <servlet>
      <servlet-name>myLibraryAppFrontController</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servl
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>myLibraryAppFrontController</servlet-name>
      <url-pattern>*.htm</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>welcome.htm</welcome-file>
   </welcome-file-list>
</web-app>
```

We have named the servlet as "`myLibraryAppFrontController`". The URI pattern in the servlet mapping section is "`*.htm`". Thus all the requests matching the URI pattern will be handled by `myLibraryAppFrontController`.

3. Spring Application Context

- Default Application context file
- User defined application context file
- Multiple application context files

Default Application context file

By default the dispatcher servlet loads the Spring application context from XML file with name *[servlet name]-servlet.xml*. Thus when our servlet `myLibraryAppFrontController` is loaded by the container, it will load the Spring application context from XML file "*/WEB-INF/myLibraryAppFrontController-servlet.xml*".

User defined application context file

We can override the name and location of the default XML file by providing the initialization parameters to the dispatcher servlet. The name of the initialization parameter is `contextConfigLocation`. The parameter value specifies the name and location of the application context which needs to be loaded by the container.

```
<servlet>
   <servlet-name>myLibraryAppFrontController</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:libraryAppContext.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
```

In the above configuration of `myLibraryAppFrontController`, when the container initializes the dispatcher servlet, it will load the Spring application context from XML file "`classpath:libraryAppContext.xml`" instead of "*/WEB-INF/myLibraryAppFrontController-servlet.xml*".

Multiple application context files

It is a good practice to split the application into multiple logical units and have multiple application context file. Thus on servlet initialization we need to load all these application context files. It is possible to load the Spring application context from multiple XML file as shown below:

```
<servlet>
   <servlet-name>myLibraryAppFrontController</servlet-name>
   <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>classpath:libraryAppContext.xml
                   classpath:books.xml
                   classpath:chapters.xml
                   classpath:titles.xml</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
```

In the above servlet configuration, we have provided multiple XML files as initialization parameter value. All these XML files will be loaded by the container on initialization of the servlet `myLibraryAppFrontController`.

4. Spring Handler mappings

- BeanNameUrlHandlerMapping
- SimpleUrlHandlerMapping

As the name specifies, the handler mapping maps the request with the corresponding request handler (in fact handler execution chain). When a request comes to Spring's dispatcher servlet, it hands over the request to the handler mapping. Handler mapping then inspects the request and identifies the appropriate handler execution chain and delivers it to dispatcher servlet. The handler execution chain contains handler that matches the incoming request and optionally contains the list of interceptors that are applied for the request. Dispatcher servlet then executes the handlers and any associated handler interceptor.

There are number of implementation of hander mapping provided by Spring's MVC module. Some of these are described below. All the handler mappings classes implement the interface `org.springframework.web.servlet.HandlerMapping`.

BeanNameUrlHandlerMapping

This implementation of handler mapping matches the URL of the incoming request with the name of the controller beans. The matching bean is then used as the controller for the request. This is the default handler mapping used by the Spring's MVC module i.e. in case

dispatcher servlet does not find any handler mapping bean defined in Spring's application context then the dispatcher servlet uses BeanNameUrlHandlerMapping.

Let us assume that we have three web pages in our application. The URL of the pages are:

1. http://servername:portnumber/ApplicationContext/welcome.htm
2. http://servername:portnumber/ApplicationContext/listBooks.htm
3. http://servername:portnumber/ApplicationContext/displayBookContent.htm

The controllers which will perform the business logic to fulfil the request made to the above pages are:

1. net.codejava.frameorks.spring.mvc.controller.WelcomeController
2. net.codejava.frameorks.spring.mvc.controller.ListBooksController
3. net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCController

Thus we need to define the controllers in Spring's application context file such that the name of the controller matches the URL of the request. The controller beans in XML configuration file will look as below.

```
<bean
    name="/welcome.htm"
    class="net.codejava.frameorks.spring.mvc.controller.WelcomeController" />
<bean
    name="/listBooks.htm"
    class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>
<bean
    name="/displayBookTOC.htm"
    class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCControll
```

Note that we need not define the BeanNameUrlHandlerMapping in Spring's application context file because this is the default one being used.

SimpleUrlHandlerMapping

The `BeanNameUrlHandlerMapping` puts a restriction on the name of the controller beans that they should match the URL of the incoming request. `SimpleUrlHandlerMapping` removes this restriction and maps the controller beans to request URL using a property "*mappings*".

```
<bean
 id="myHandlerMapping"
 class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
   <property name="mappings">
      <props>
         <prop key="/welcome.htm">welcomeController</prop>
         <prop key="/listBooks.htm">listBooksController</prop>
         <prop key="/displayBookTOC.htm">displayBookTOCController</prop>
      </props>
   </property>
</bean>
<bean name="welcomeController"
 class="net.codejava.frameorks.spring.mvc.controller.WelcomeController"/>
<bean name="listBooksController"
 class="net.codejava.frameorks.spring.mvc.controller.ListBooksController"/>
<bean name="displayBookTOCController"
 class="net.codejava.frameorks.spring.mvc.controller.DisplayBookTOCControlle
```

The key of the *<prop>* element is the URL pattern of the incoming request. The value of the *<prop>* element is the name of the controller bean which will perform the business logic to fulfil the request. `SimpleUrlHandlerMapping` is one of the most commonly used handler mapping.

5. Spring Controllers

- MultiActionController

Controller is the actual piece of code which performs the business logic to fulfil the incoming request. Controllers may delegate this responsibility to further service objects as well. All the user defined controllers must either implement the interface Controller or extend the abstract class `AbstractController`. The user defined controllers need to override the method `handleRequestInternal`. The method `handleRequestInternal` takes `HttpServletRequest` and `HTTPServletResponse` as the input and returns an object of `ModelAndView`.

In the Spring's application context file, we have defined a user defined custom controller named `welcomeController`. As per the `SimpleUrlHandlerMApping`, all the requests matching URL pattern /welcome.htm will be handled by this controller. The `WelcomeController` must extend `AbstractController` and provide the definition of method `handleRequestInternal`. Thus `WelcomeController` looks as below:

```
public class WelcomeController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest arg0,
            HttpServletResponse arg1) throws Exception {

        return new ModelAndView("welcome");
    }
}
```

MultiActionController

In any medium to large size enterprise web application, there are quite a number of web pages. To fulfil the request for those web pages we need to define multiple controllers, one each for a web page. And sometimes the business logic is executed to fulfil those requests is similar. This creates redundancy of business logic in multiple controllers and makes the maintenance difficult.

Spring's MVC module provides a way to deal with this scenario by providing a single controller fulfilling the request for multiple web pages. Such a controller is known as *Multi Action Controller*. A user defined multi action controller should extend the class `org.springframework.web.servlet.mvc.multiaction.MultiActionController`. Each method in user defined multi action controller contains the logic to fulfil the request for a particular web page.

By default, the URL of the incoming request (excluding the extension part) will be matched against the name of the method in multi action controller and the matching method will perform the business logic for the incoming request. So for the incoming request with URL `/welcome.htm`, the method name containing the business logic will be welcome.

Let us assume that the multi action controller in our application is `MyMultiActionController` which fulfils the request for the three web pages with URL `/welcome.htm`, `/listBooks.htm` and `/displayBookTOC.htm`. Thus the class should extend the `MultiActionController` and have three methods with name `welcome`, `listBooks` and `displayBookTOC`. The controller will look as below:

```
1  public class MyMultiActionController extends MultiActionController {
2      // This method will server all the request matching URL pattern /welcome
3      public ModelAndView welcome(HttpServletRequest request,
4              HttpServletResponse response) {
5          // Business logic goes here
6          // Return an object of ModelAndView to DispatcherServlet
7          return new ModelAndView("Welcome");
8      }
9      // This method will server all the request matching URL pattern
10     // /listBooks.htm
11     public ModelAndView listBooks(HttpServletRequest request,
12             HttpServletResponse response) {
13         // Business logic goes here
14         // Return an object of ModelAndView to DispatcherServlet
15         return new ModelAndView("listBooks");
16     }
17     // This method will server all the request matching URL pattern
18     // /displayBookTOC.htm
19     public ModelAndView displayBookTOC(HttpServletRequest request,
20             HttpServletResponse response) {
21         // Business logic goes here
22         // Return an object of ModelAndView to DispatcherServlet
23         return new ModelAndView("displayBookTOC");
24     }
25 }
```

MethodNameResolver

Spring MVC provides a number of other method name resolvers that helps to resolve the multi action controller method name based on the request. Some of these are:

meterMethodNameResolver

A particular parameter in the request contains the method name. The name of the parameter is defined in the Spring's application context file while defining `ParameterMethodNameResolver`. In the example below, the parameter `controllerMethod` in the request will determine the multi action controller method which will be executed to fulfil the request.

```
<bean name="parameterMethodNameResolver"
      class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNa
   <property name="paramName">
      <value>controllerMethod</value>
   </property>
</bean>
```

Notes: The request for a particular web page should now contain an additional parameter with name "`controllerMethod`" and value as the multi action controller method name to be executed. The request URL will be as follow:

1. http://servername:portnumber/ProjectWebContext/welcome.htm?controllerMethod= handleWelcomePage
2. http://servername:portnumber/ProjectWebContext/listBooks.htm?controllerMethod= handleListBooksPage
3. http://servername:portnumber/ProjectWebContext/displayBookTOC.htm? controllerMethod= handleDisplayBookTOCPage

In the above configuration, the request for URL `/welcome.htm` will be fulfilled by method `handleWelcomePage` of multi action controller. Request for URL `/listBooks.htm` will be fulfilled by method `handleListBooksPage` and request for URL `/displayBookTOC.htm` will be fulfilled by method `handleDisplayBookTOCPage`.

PropertiesMethodNameResolver

The name of method is determined from the list of pre-defined properties supplied to the method name resolver in Spring's application context file. The `PropertiesMethodNameResolver` in Spring's application context file will look as below.

```
<bean name="propertiesMethodNameResolver"
      class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethod
   <property name="mappings">
      <props>
         <prop key="/welcome.htm">handleWelcomePage</prop>
         <prop key="/listBooks.htm">handleListBooksPage</prop>
         <prop key="/displayBookTOC.htm">handleDisplayBookTOCPage</prop>
      </props>
   </property>
</bean>
```

Again, in the above configuration, the request for URL `/welcome.htm` will be fulfilled by method `handleWelcomePage` of multi action controller. Request for URL `/listBooks.htm` will be fulfilled by method `handleListBooksPage` and request for URL `/displayBookTOC.htm` will be fulfilled by method `handleDisplayBookTOC`.

need to tell the multi action controller to use a particular method name resolver by ing its property `methodNameResolver`. Thus the configuration of multi action controller will look as below:

```
<bean name="myMultiActionController"
      class="net.codejava.frameworks.spring.mvc.controller.MyMultiActionContr
   <property name="methodNameResolver">
      <ref bean="propertiesMethodNameResolver"/>
   </property>
</bean>
```

6. ModelAndView and ViewResolver

ModelAndView

Spring's MVC module encapsulates the model object and the view object in a single entity which is represented by the object of class `ModelAndView`. This object contains the model object and view object or the logical name of the view. The model object is the application data and the view is the object that renders the output to the user. The controller returns an object of `ModelAndView` to the dispatcher servlet for further processing.

ViewResolver

In case `ModelAndView` object contains the logical name of the view then the `DispatcherServlet` needs resolving the view object based on its logical name. To resolve the view object, `DispatcherServlet` take the help of `ViewResolver`. There are number of implementation of view resolver provided by Spring. All the view resolvers implement the interface `org.springframework.web.servlet.ViewResolver`.

InternalResourceViewResolver

It resolves the logical name of the view to an internal resource by prefixing the logical view name with the resource path and suffixing it with the extension.

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver
      <property name="prefix"value="/WEB-INF/jsp/" />
      <property name="suffix"value=".jsp" />
</bean>
```

If the logical name of the view returned by the controller in `ModelAndView` object is `Welcome` then the view which is shown to the user is `/WEB-INF/jsp/Welcome.jsp`

BeanNameViewResolver

olves the logical name of the view to the bean name which will render the output to the The bean should be defined in the Spring app context file. So if the logical name returned by the controller in `ModelAndView` object is `Welcome` then the bean with name `Welcome` defined in the application context will be responsible to render the model to the user.

XMLFileViewResolver

This view resolver is the same as `BeanNameViewResolver` with only difference is that instead of looking for the beans in Spring's application context file it looks for beans defined in a separate XML file `(/WEB-INF/views.xml` by default). The location and file name can be overridden by providing a `location` property while defining the `XMLFileViewResolver`.

```
<bean name="propertiesMethodNameResolver"
      class="org.springframework.web.servlet.view.XMLFileViewResolver">
      <propertyname="location">
            <value>classpath:myViews.xml</value>
      </property>
</bean>
```

ResourceBundleViewResolver

It resolves the logical name of the view to the actual view defined in the resource bundle. This view resolver takes `basename` as the input which is the name of the property file where views can be located.

```
<bean name="propertiesMethodNameResolver"
      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
   <property name="basename">
      <value>myViews</value>
   </property>
</bean>
```

So if the logical name returned by the controller in `ModelAndView` object is `Welcome` then the view resolver will look for the property `Welcome.class` in properties file `myViews.properties` (or `myViews_en_US.properties` depending upon the user language and locale).

7. Sample Application

Application Overview

Mr. XYZ has written a book on his favourite topic "*Spring - Core*". He has written a number of books and now is planning to develop a web site **–** *an online book store*, where he can publish all his books. The web site is very simple, a welcome page, a page listing all the books in his online book store and a page listing the table to content of a particular book selected by the end user.

Mr. XYZ has chosen to develop his online book store using his favourite framework Spring MVC. He has named the front controller of his application as myLibraryAppFrontController which loads the application context from multiple XML files - libraryAppContext.xml, books.xml, chapters.xml and titles.xml. He has written three JSP pages corresponding to his three pages in the application and has placed them in folder WebContent/WEB-INF/jsp.

The Handler Mapping used in the online book store application is the Spring MVC module's default handler mapping **– BeanNamURLHandlerMApping.**

The requests for the three pages are being served by three different controllers.

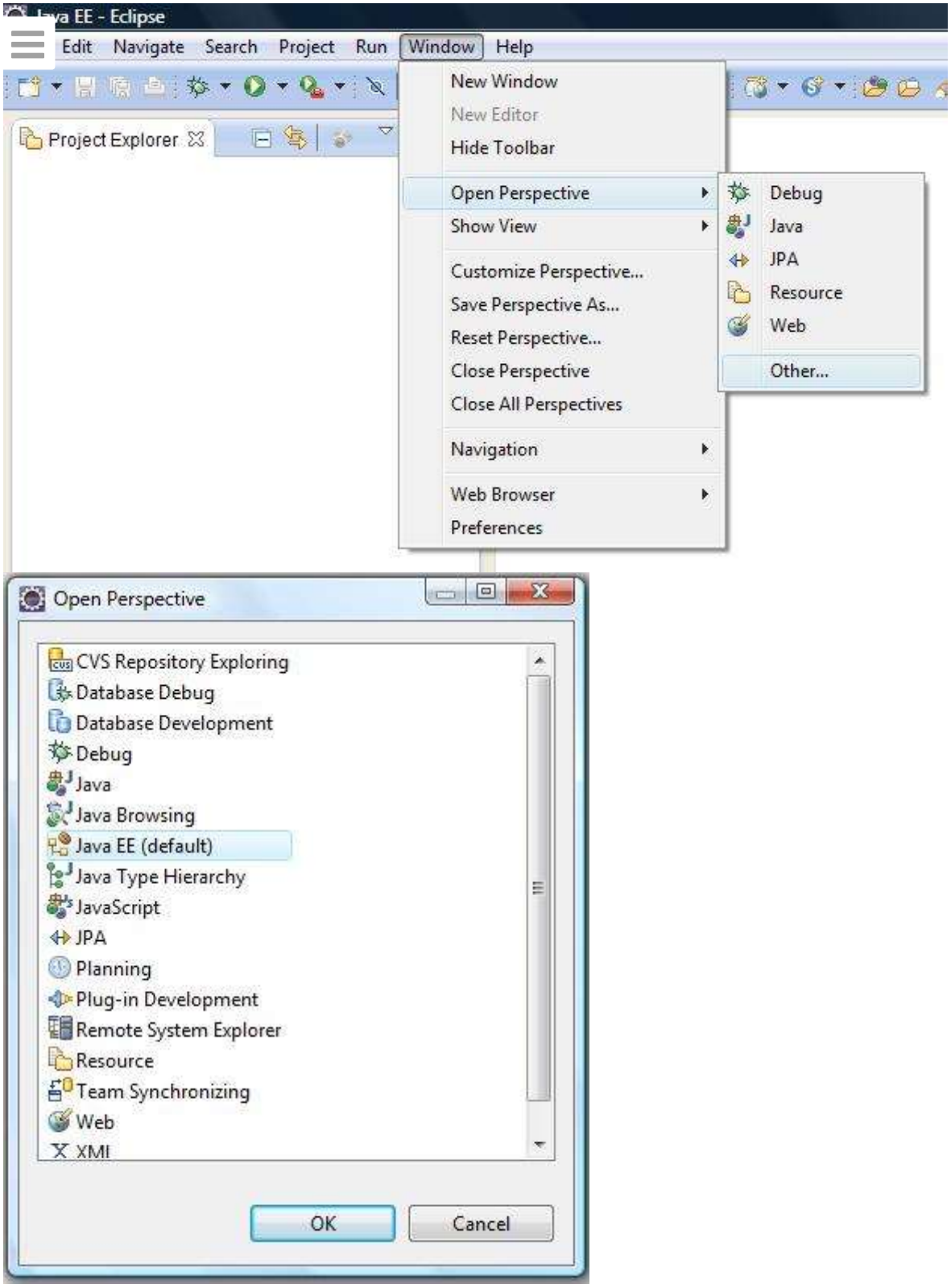
1. Welcome page (/welcome.htm)**–** WelcomeController
2. Page displaying list of books (/listBooks.htm) **–** ListBooksController
3. Page displaying table of content of the book (/displayBookTOC.htm) **–** DisplayBookTOCController

Thus there are three beans (three controllers) defined in XML file libraryAppContext.xml with bean name as page URL. The view resolver used in this application is Spring's InternalResourceViewResolver which appends the prefix as /Web-INF/jsp and suffix .jsp to the logical view name returned by the controller to resolve the actual view (in this case the jsps).

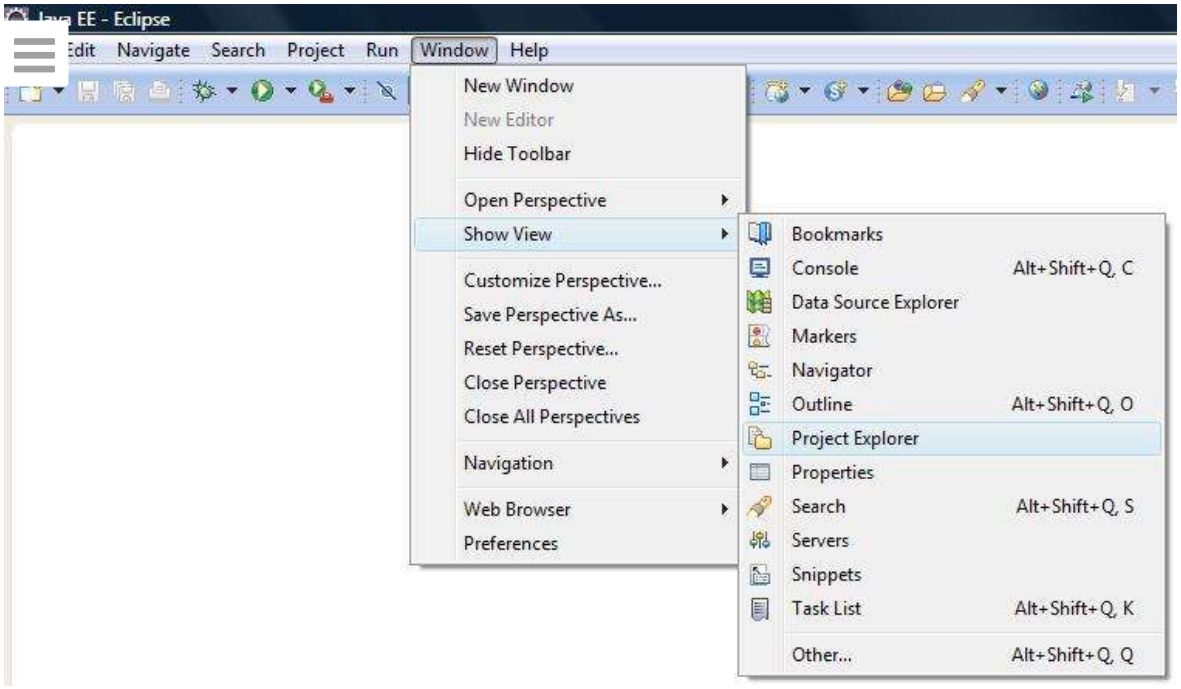
Setting up workspace in Eclipse IDE

We will use **Eclipse IDE for Java EE developers version 4.2 (Juno)** to build the sample application using **Spring version 3.2.0.M2**. We will run our sample application on **Apache Tomcat server v 7.0** thus please ensure that the Apache Tomcat v7.0 is already installed on the system and is integrated with the Eclipse IDE.

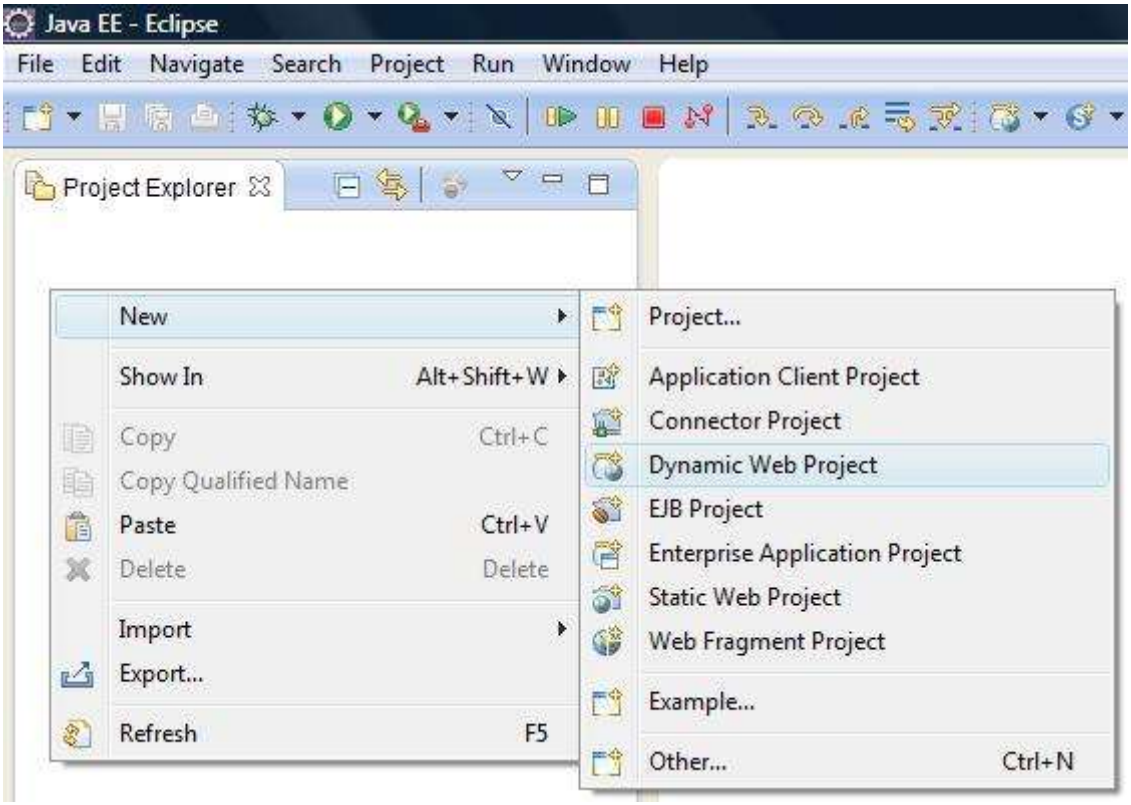
Open Eclipse IDE. From the menu bar select, **Window > Open Perspective > Others... > Java EE**.



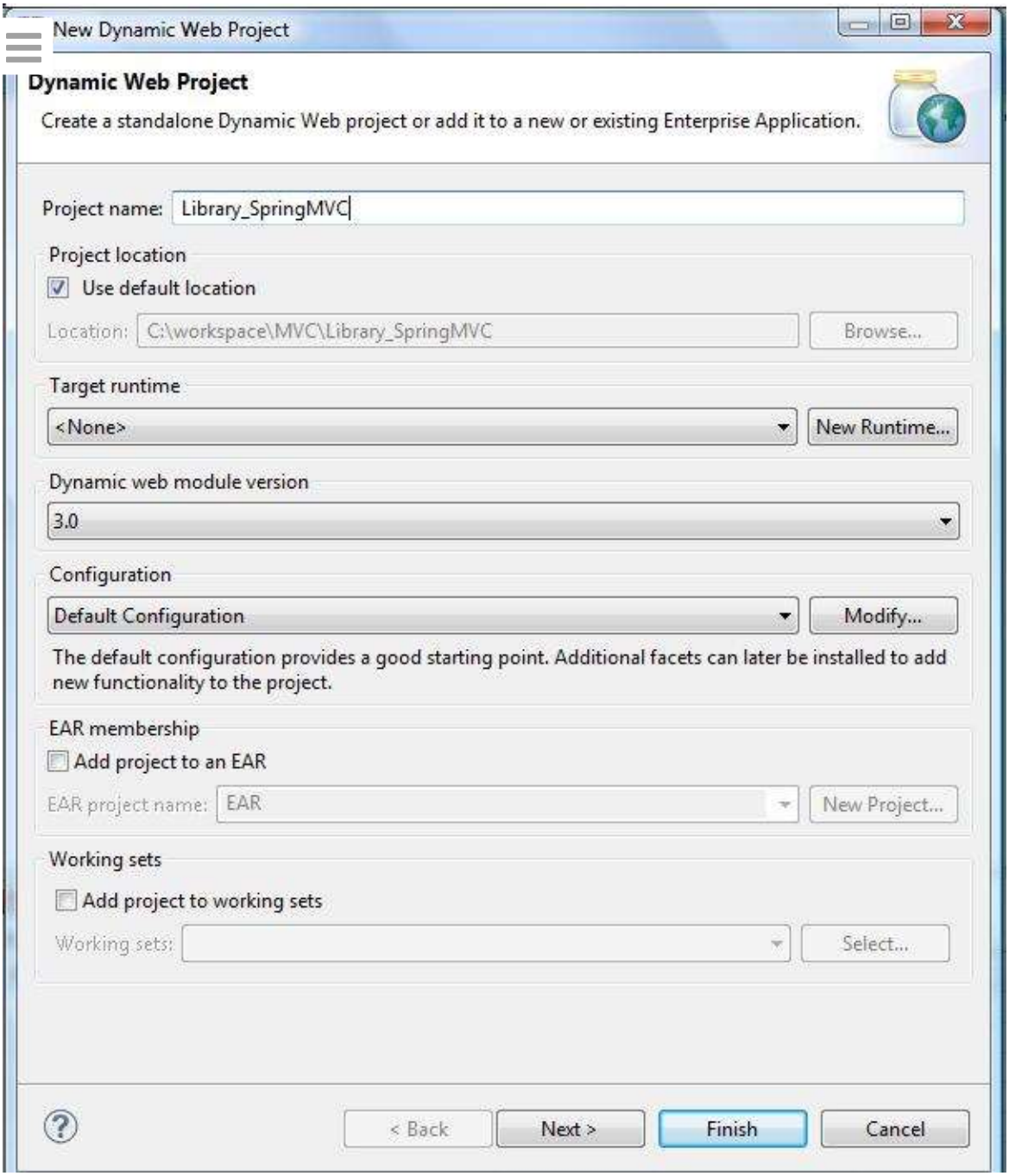
Open Project Explorer view. From the menu bar select, **Window > Show View > Project Explorer**.



Right click anywhere in project explorer view. Select **New > Dynamic Web Project**.



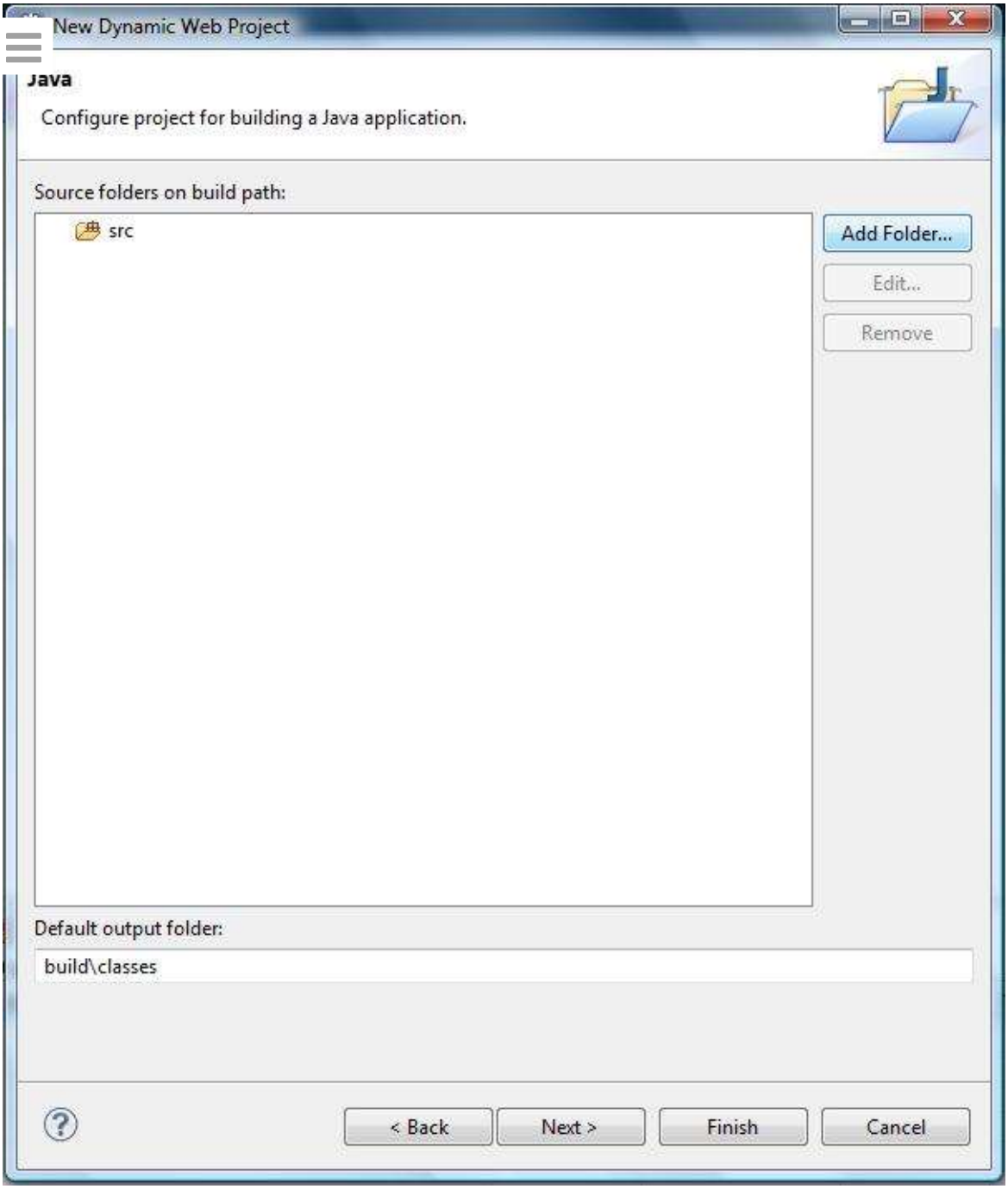
A pop up window will open up, we will enter the details for creating the dynamic web project. Enter *Project name* : **Library_SpringMVC**. Select *Target Runtime* as **Apache Tomcat v7.0**. Also ensure that *configuration* is set as **Default Configuration for Apache Tomcat v7.0***.* Please ensure that the Apache Tomcat v7.0 is already installed on the system and is integrated with the Eclipse IDE.



Click the **Next** button on the *Dynamic Web Project* dialogue. The next dialogue is for *Java* settings for the project. Add the following two source folders using **Add Folder...** button on the right hand side of the *Java* dialogue.

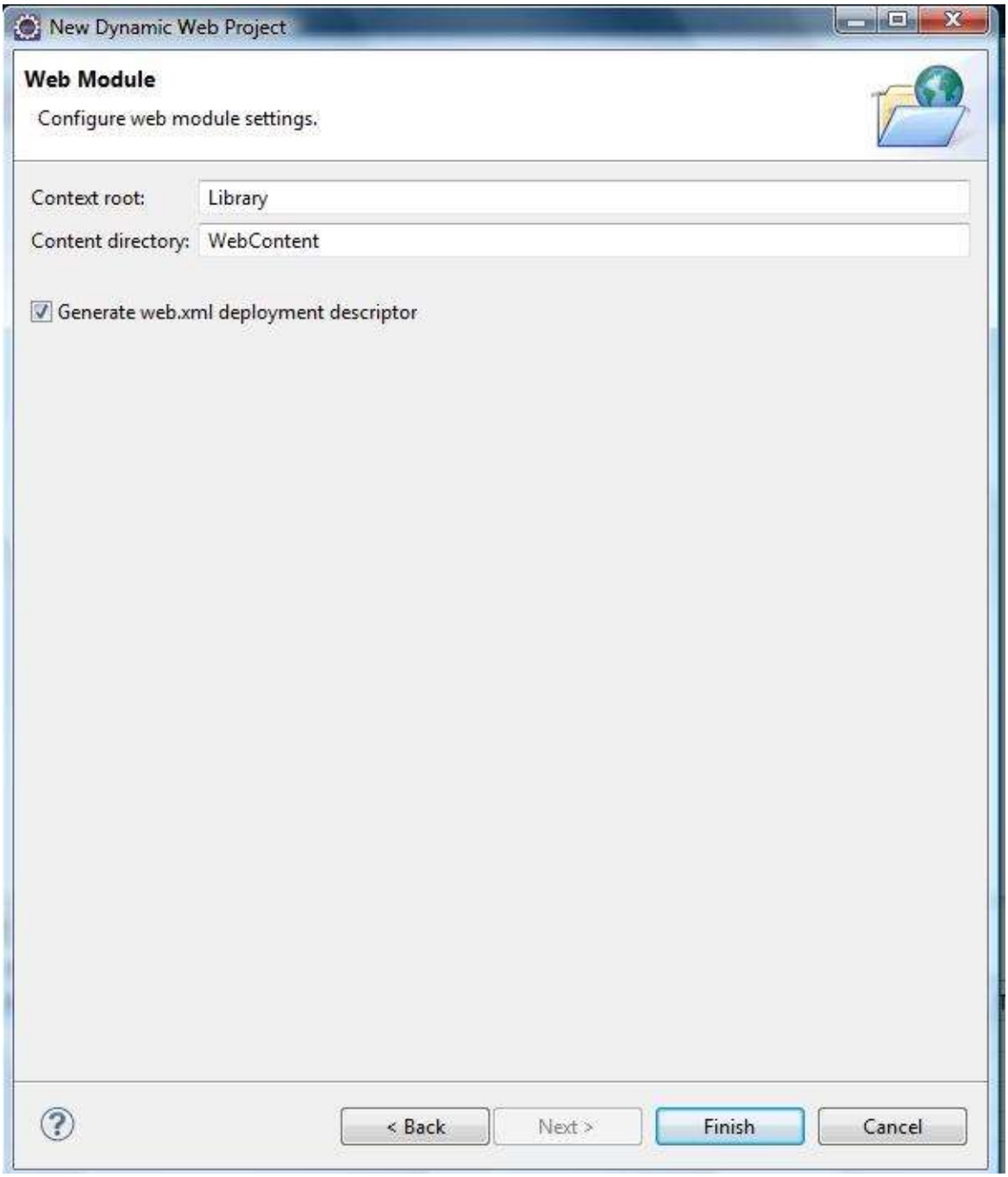
1. SourceCode/JavaSource - All the java code of our sample application will reside here.

2. SourceCode/Config - All the XML and properties files of our sample application will reside here.


New Dynamic Web Project
Java
Configure project for building a Java application.
Source folders on build path:
src
Add Folder...
Edit...
Remove
Default output folder:
build\classes
< Back
Next >
Finish
Cancel

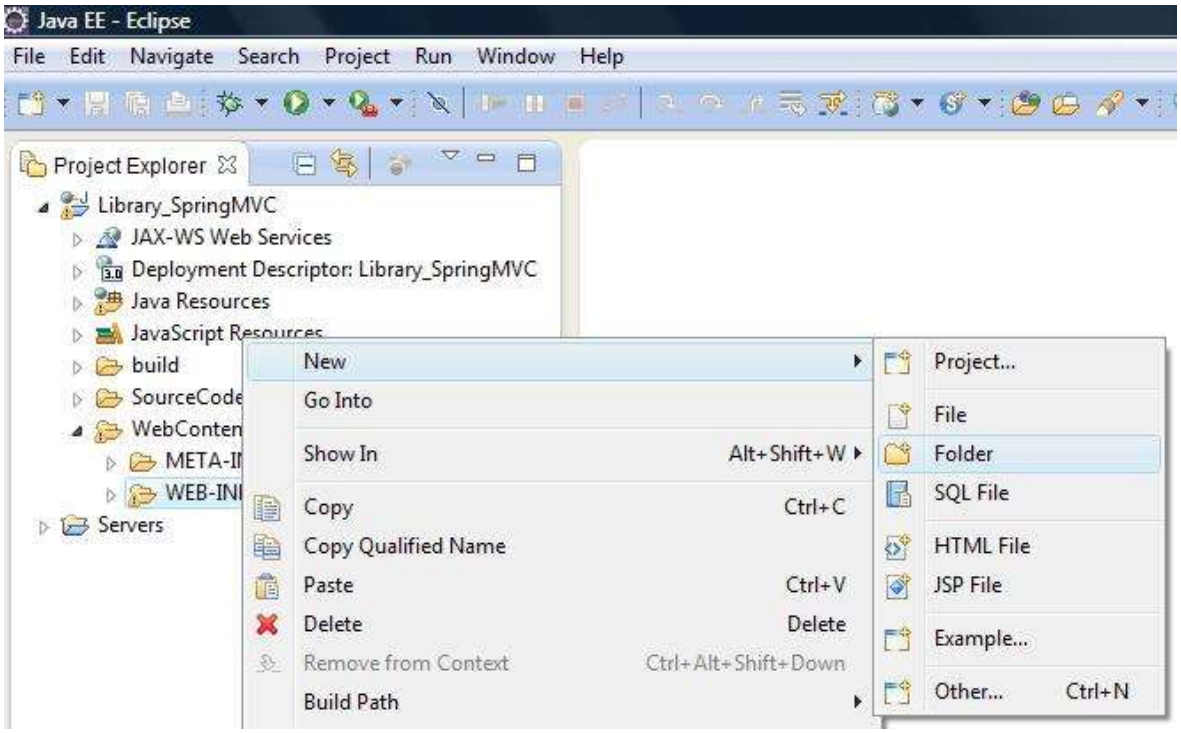

Add Source Folder
Specify the name of the new source folder.
SourceCode/JavaSource
OK
Cancel

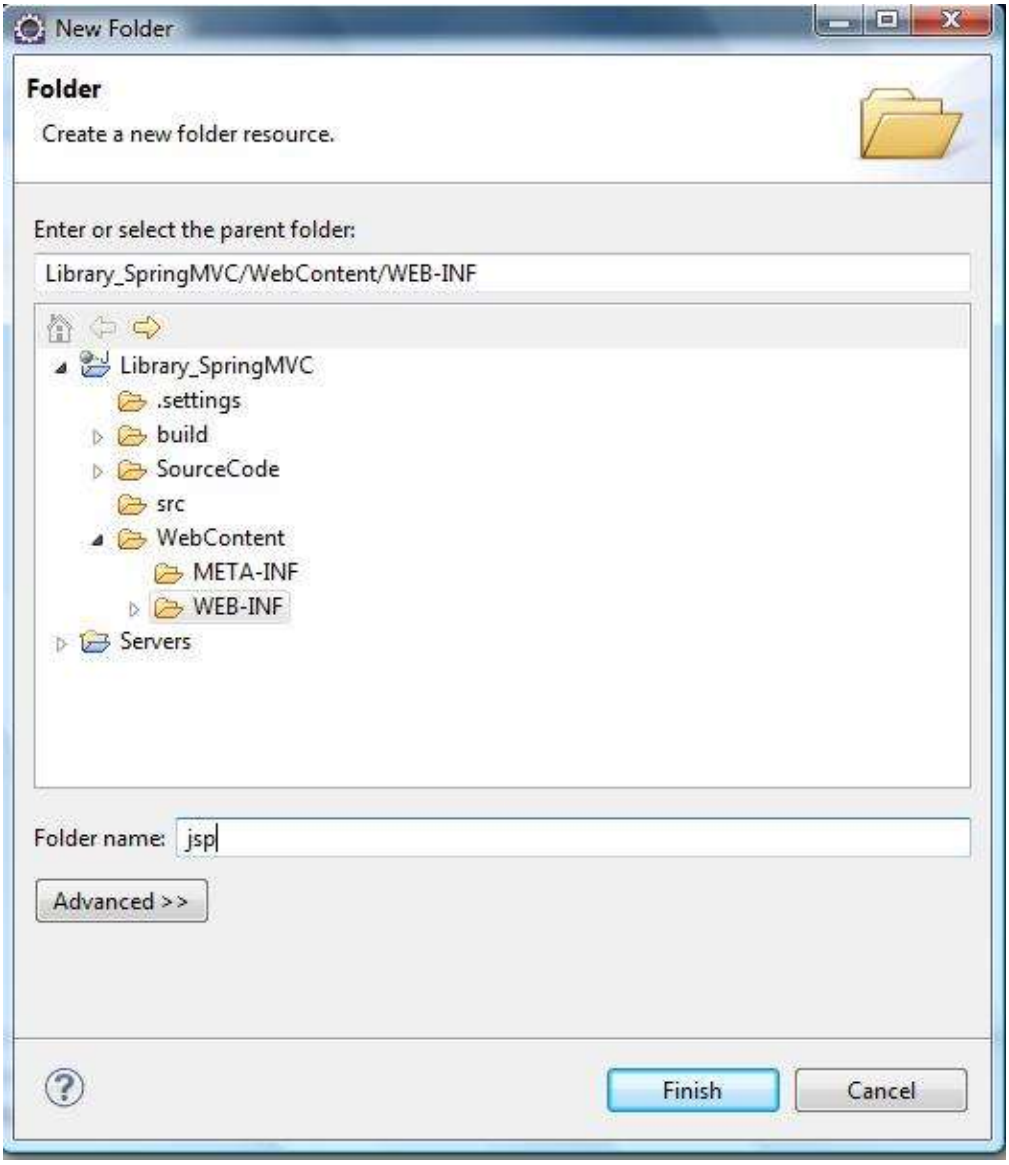
Click on the **Next** button on the *Java* dialogue. The next dialogue is for *Web Module*. In this dialgoue, ensure that the *Context root* is set as **Library**, *Content directory* is set as **WebContent** and ensure that check box corresponding to **Generate web.xml deployment descriptor** is checked.



n you click the **Finish** button, the dynamic web project with name **Library_SpringMVC** e created in your workspace.

Create folder for JSP pages: We will place all the JSP pages in folder *WebContent/WEB-INF/jsp*. Thus we need to create the folder with name *jsp* within *WEB-INF* folder. Right click the WEB-INF folder and select **New>folder**. Enter the name as *jsp*.





The various jar files which are required to build and run our sample application based on Spring MVC are as below:

- spring-beans-3.2.0.M2.jar
- spring-context-3.2.0.M2.jar
- spring-core-3.2.0.M2.jar
- spring-expression-3.2.0.M2.jar
- spring-web-3.2.0.M2.jar
- spring-webmvc-3.2.0.M2.jar

The above jars can be downloaded from following location:

https://repo.springsource.org/libs-milestone-local/org/springframework/spring/3.2.0.M2/spring-3.2.0.M2-dist.zip

Download the zip file from above location. Unzip it. Copy the relevant jars (as mentioned above) from *spring-3.2.0.M2\libs* to the *lib* folder of the sample java application.

We also need following jars:

- commons-logging-1.1.1.jar - This jar is required for logging. Spring MVC jars (mentioned above) require this jar file.

n the jar from the following location and copy it in the lib folder present in the project.

http://apache.techartifact.com/mirror/commons/logging/binaries/commons-logging-1.1.1-bin.zip

Following two jar files are required for using Java Tag Library in JSP pages.

- javax.servlet.jsp.jstl-1.2.2.jar

Down the jar from the following location and copy it in the lib folder present in the project.

http://search.maven.org/remotecontent?filepath=org/glassfish/web/javax.servlet.jsp.jstl/1.2.2/javax.servlet.jsp.jstl-1.2.2.jar

- javax.servlet.jsp.jstl-api-1.2.1.jar

Down the jar from the following location and copy it in the lib folder present in the project.

http://search.maven.org/remotecontent?filepath=javax/servlet/jsp/jstl/javax.servlet.jsp.jstl-api/1.2.1/javax.servlet.jsp.jstl-api-1.2.1.jar

Coding the model

The various application objects used in our sample applications are:

1. `Book`
2. `Chapter`
3. `Title`

Book object consist of:

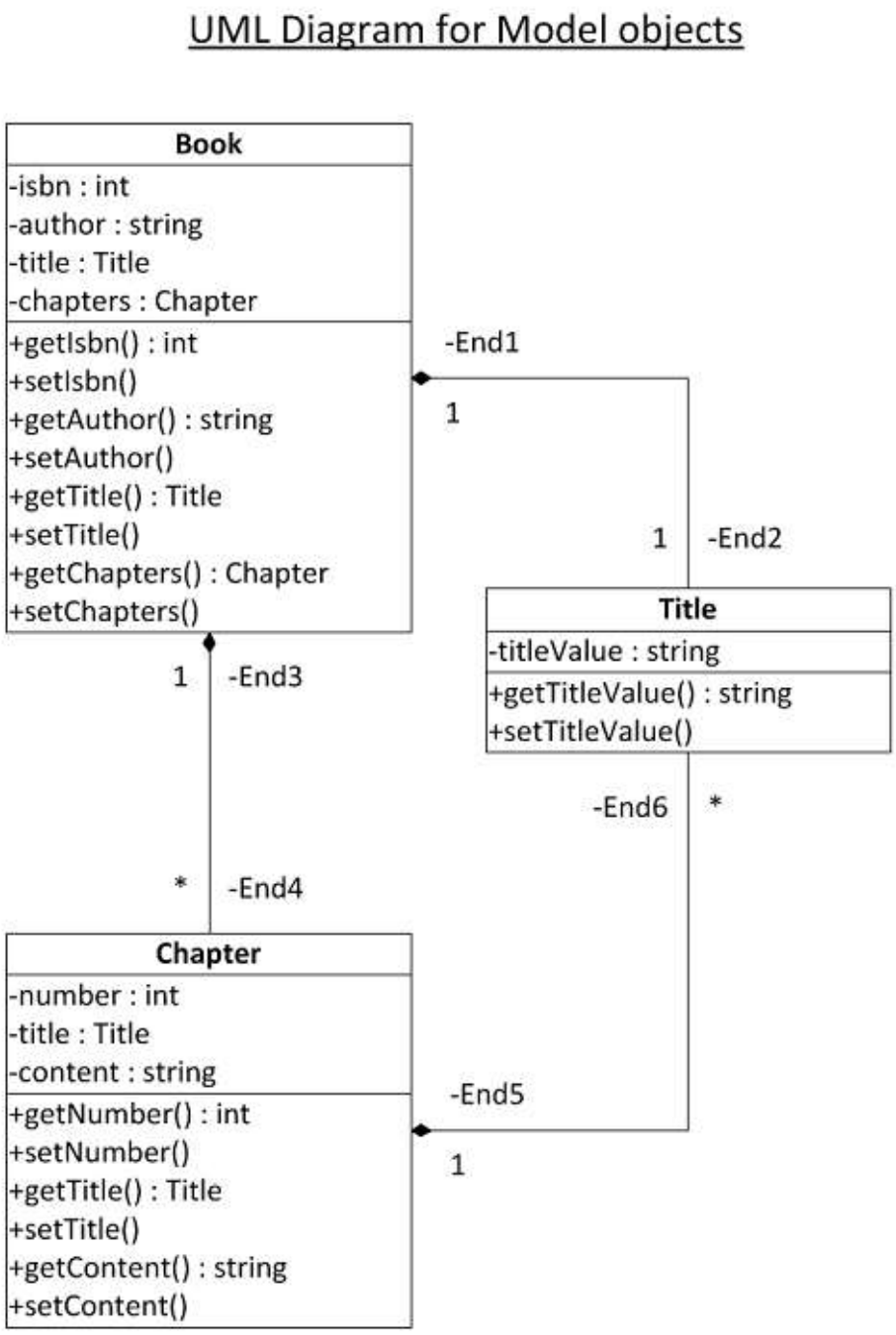
1. `Title` – Represented by `Title` object
2. `Author` – Represented by String
3. `ISBN` – Represented by integer
4. List of Chapters – Represented by List`<Chapter>`

Chapter object consist of:

1. Number – Chapter number represented by integer
2. `Title` – Represented by `Title` object
3. Content – Content of the chapter represented by String

Title object consist of:

1. `titleValue` – Represented by String



UML Diagram for our sample application Model objects

The java classes for the application objects are placed within source folder *SourceCode/JavaSource*. The java code of our application objects looks as below:

Title.java

```java
package net.codejava.frameworks.spring.bo;

public class Title {
    private String titleValue;
    public Title(){
    }

    public Title(String titleValue){
        this.titleValue = titleValue;
    }

    public String getTitleValue() {
        return titleValue;
    }
    public void setTitleValue(String titleValue) {
        this.titleValue = titleValue;
    }
}
```

Chapter.java

```java
package net.codejava.frameworks.spring.bo;

public class Chapter {
    private int number;
    private Title title;
    private String content;

    public Chapter(){
    }

    public Chapter(int number, Title title, String content){
        this.number = number;
        this.title = title;
        this.content = content;
    }

    public int getNumber() {
        return number;
    }
    public void setNumber(int number) {
        this.number = number;
    }
    public Title getTitle() {
        return title;
    }
    public void setTitle(Title title) {
        this.title = title;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }

}
```

Book.java

```java
package net.codejava.frameworks.spring.bo;

import java.util.List;

public class Book {
    private int isbn;
    private String author;
    private Title title;
    private List<Chapter> chapters;

    public Book(){
    }

    public Book(int isbn, String author, Title title, List<Chapter> chapters)
        this.isbn = isbn;
        this.author = author;
        this.title = title;
        this.chapters = chapters;
    }

    public int getIsbn() {
        return isbn;
    }
    public void setIsbn(int isbn) {
        this.isbn = isbn;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public Title getTitle() {
        return title;
    }
    public void setTitle(Title title) {
        this.title = title;
    }
    public List<Chapter> getChapters() {
        return chapters;
    }
    public void setChapters(List<Chapter> chapters) {
        this.chapters = chapters;
    }
}
```

All the book beans are defined in XML file `books.xml`, all the chapter beans are defined in `chapters.xml` and all the title beans are defined in `titles.xml` file. All these XML files are placed within *SourceCode/Config* folder.

The various Spring's application context files look as below:

titles.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholder
          <property name="location">
                <value>classpath:beans.properties</value>
          </property>
    </bean>
    <bean id="bookTitle" class="net.codejava.frameworks.spring.bo.Title">
        <property name="titleValue">
            <value>${myFirstSpringBook.title}</value>
        </property>
    </bean>
    <bean id="chapter1Title" class="net.codejava.frameworks.spring.bo.Title">
        <constructor-arg>
            <value>${myFirstSpringBook.chapter1.title}</value>
        </constructor-arg>
    </bean>

    <bean id="chapter2Title" class="net.codejava.frameworks.spring.bo.Title">
        <constructor-arg>
            <value>${myFirstSpringBook.chapter2.title}</value>
        </constructor-arg>
    </bean>

    <bean id="chapter3Title" class="net.codejava.frameworks.spring.bo.Title">
        <property name="titleValue">
            <value>${myFirstSpringBook.chapter3.title}</value>
        </property>
    </bean>
</beans>
```

chapters.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholder
          <property name="location">
                <value>classpath:beans.properties</value>
          </property>
    </bean>
    <bean id="chapter1" class="net.codejava.frameworks.spring.bo.Chapter">
        <property name="number">
            <value>1</value>
        </property>
        <property name="content">
            <value>${myFirstSpringBook.chapter1.content}</value>
        </property>
        <property name="title">
            <ref bean="chapter1Title"/>
        </property>
    </bean>

    <!-- injecting the dependencies of chapter 2 using constructor by index
    <bean id="chapter2" class="net.codejava.frameworks.spring.bo.Chapter">
          <constructor-arg index="0">
                <value>2</value>
          </constructor-arg>
          <constructor-arg index="1">
                <ref bean="chapter2Title"/>
          </constructor-arg>
          <constructor-arg index="2">
                <value>${myFirstSpringBook.chapter2.content}</value>
          </constructor-arg>
    </bean>

    <!-- injecting the dependencies of chapter 3 using constructor by type --
    <bean id="chapter3" class="net.codejava.frameworks.spring.bo.Chapter">
          <constructor-arg type="int">
                <value>3</value>
          </constructor-arg>
          <constructor-arg type="net.codejava.frameworks.spring.bo.Title">
                <ref bean="chapter3Title"/>
          </constructor-arg>
          <constructor-arg type="String">
                <value>${myFirstSpringBook.chapter3.content}</value>
          </constructor-arg>
    </bean>
</beans>
```

books.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholder
          <property name="location">
                <value>classpath:beans.properties</value>
          </property>
    </bean>
    <bean id="myFirstSpringBook" class="net.codejava.frameworks.spring.bo.Boo
        <property name="isbn">
            <value>1</value>
        </property>
        <property name="author">
            <value>${myFirstSpringBook.author}</value>
        </property>
        <property name="title">
            <ref bean="bookTitle"/>
        </property>
        <property name="chapters">
            <list>
                <ref bean="chapter1"/>
                <ref bean="chapter2"/>
                <ref bean="chapter3"/>
            </list>
        </property>
    </bean>

</beans>
```

Note that we have used the beans.properties file in the above XML files. The *beans.properties* file looks as below:

```
myFirstSpringBook.title=My First Spring Book
myFirstSpringBook.chapter1.title=Spring framework - Chapter 1
myFirstSpringBook.chapter2.title=Spring framework - Chapter 2
myFirstSpringBook.chapter3.title=Spring framework - Chapter 3
myFirstSpringBook.chapter1.content=The content of chapter 1 goes here.
myFirstSpringBook.chapter2.content=The content of chapter 2 goes here.
myFirstSpringBook.chapter3.content=The content of chapter 3 goes here.
myFirstSpringBook.author=Mr. XYZ
```

Configure the Spring's front controller (DispatcherServlet)

Open `Web.xml` file present in the Library project at location `WebContent/WEB-INF`. Configure the dispatcher Servlet and servlet mapping as shown below.

```
<web-app
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.cor
    id="WebApp_ID"
    version="2.5"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
   <display-name>Library</display-name>
   <servlet>
      <servlet-name>myLibraryAppFrontController</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servl
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>classpath:libraryAppContext.xml
                      classpath:books.xml
                      classpath:chapters.xml
                      classpath:titles.xml</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>myLibraryAppFrontController</servlet-name>
      <url-pattern>*.htm</url-pattern>
   </servlet-mapping>
   <welcome-file-list>
      <welcome-file>welcome.htm</welcome-file>
   </welcome-file-list>
</web-app>
```

Here servlet name is `myLibraryAppFrontController`. URI pattern in servlet mapping is *.htm.

There is one initialization parameter given to the servlet. The initialization parameter name is `contextConfigLocation and its values are`

```
classpath:libraryAppContext.xml
classpath:books.xml
classpath:chapters.xml
classpath:titles.xml
```

This ensures that the container looks for the mentioned XML files to load the Spring application context instead of looking for default XML file “WEB-INB/`myLibraryAppFrontController-servelt.xml`”

Create the XML file with name `libraryAppContext.xml` in folder SourceCode/Config. This is the Spring’s application context file which will contain the Spring MVC specific beans. The various beans defined in this file are:

1. `HandlerMapping` maps the request with the controller bean. We have used `SimpleUrlHandlerMapping`. Web page `/welcome.htm` will be served by controller with bean name `welcomeController`, `/listBooks.htm` will be served by controller with bean name `listBooksController` and `/displayBookTOC.htm` will be served by controller with bean name `displayBookTOCController`.
2. Three controller beans serving the request for three web pages.
3. `ViewResolver` bean resolves the view logical name to the JSP file. We have used `InternalResourceViewResolver` with prefix as `/WEB-INF/jsp` and suffix as `.jsp`. This is because we have placed our view objects in this folder.

aryAppContext.xml file will look as below

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean
        class="org.springframework.beans.factory.config.PropertyPlaceholderC
        <property name="location">
            <value>classpath:beans.properties</value>
        </property>
    </bean>
    <bean id="myHandlerMapping" class="org.springframework.web.servlet.handle
        <property name="mappings">
            <props>
                <prop key="/welcome.htm">welcomeController</prop>
                <prop key="/listBooks.htm">listBooksController</prop>
                <prop key="/displayBookTOC.htm">displayBookTOCController</pr
            </props>
        </property>
    </bean>
    <bean name="welcomeController" class="net.codejava.frameworks.spring.mvc

    <bean name="listBooksController" class="net.codejava.frameworks.spring.m
        <property name="books">
            <list>
                <ref bean="myFirstSpringBook"/>
            </list>
        </property>
    </bean>
    <bean name="displayBookTOCController" class="net.codejava.frameworks.spr
        <property name="books">
            <list>
                <ref bean="myFirstSpringBook"/>
            </list>
        </property>
    </bean>
    <bean id="viewResolver" class="org.springframework.web.servlet.view.Inte
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

Coding the controller

Let us have a quick look on the three controllers. All the three controllers extends `AbstractController` and override method `handleRequestInternal`. This method returns an object of `ModelAndView` which contains the logical name of the view and the model object.

In case of `welcomeBookController` the logical name of the view is `Welcome` and there is no model object being returned.

```
package net.codejava.frameworks.spring.mvc.controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class WelcomeController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        return new ModelAndView("welcome");
    }
}
```

In case of `listBooksController`, the logical name of the view is `listBooks` and the model object being returned is the list of books objects.

```
package net.codejava.frameworks.spring.mvc.controller;

import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import net.codejava.frameworks.spring.bo.Book;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class ListBooksController extends AbstractController {

    private List<Book> books;
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        return new ModelAndView("listBooks","books",books);
    }
    public List<Book> getBooks() {
        return books;
    }
    public void setBooks(List<Book> books) {
        this.books = books;
    }
}
```

In case of `displayBookTOCController`, the logical name of the view is `displayBookTOC` and the model object being returned is the object of the book whose ISBN is being passed in the request.

```
package net.codejava.frameworks.spring.mvc.controller;

import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import net.codejava.frameworks.spring.bo.Book;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class DisplayBookTOCController extends AbstractController {
    private List<Book> books;

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        Book myBook = null;
        if(books != null && !books.isEmpty()){
            for(Book book : books){
                if(book.getIsbn() == Integer.parseInt(request.getParameter("
                    myBook = book;
                    break;
                }
            }
        }
        return new ModelAndView("displayBookTOC","book",myBook);
    }
    public List<Book> getBooks() {
        return books;
    }
    public void setBooks(List<Book> books) {
        this.books = books;
    }
}
```

Coding the View

The various JSPs being displayed to the user are placed within `WebContent/WEB-INF/jsp` folder. The JSPs are as below:

Welcome.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEnc
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"
      <title>Welcome page</title>
   </head>
   <body>
      <h1>Welcome to the world of books !</h1>
      <p>The page content goes here . . .</p>
      <a href="/listBooks.htm">List all books.</a>
   </body>
</html>
```

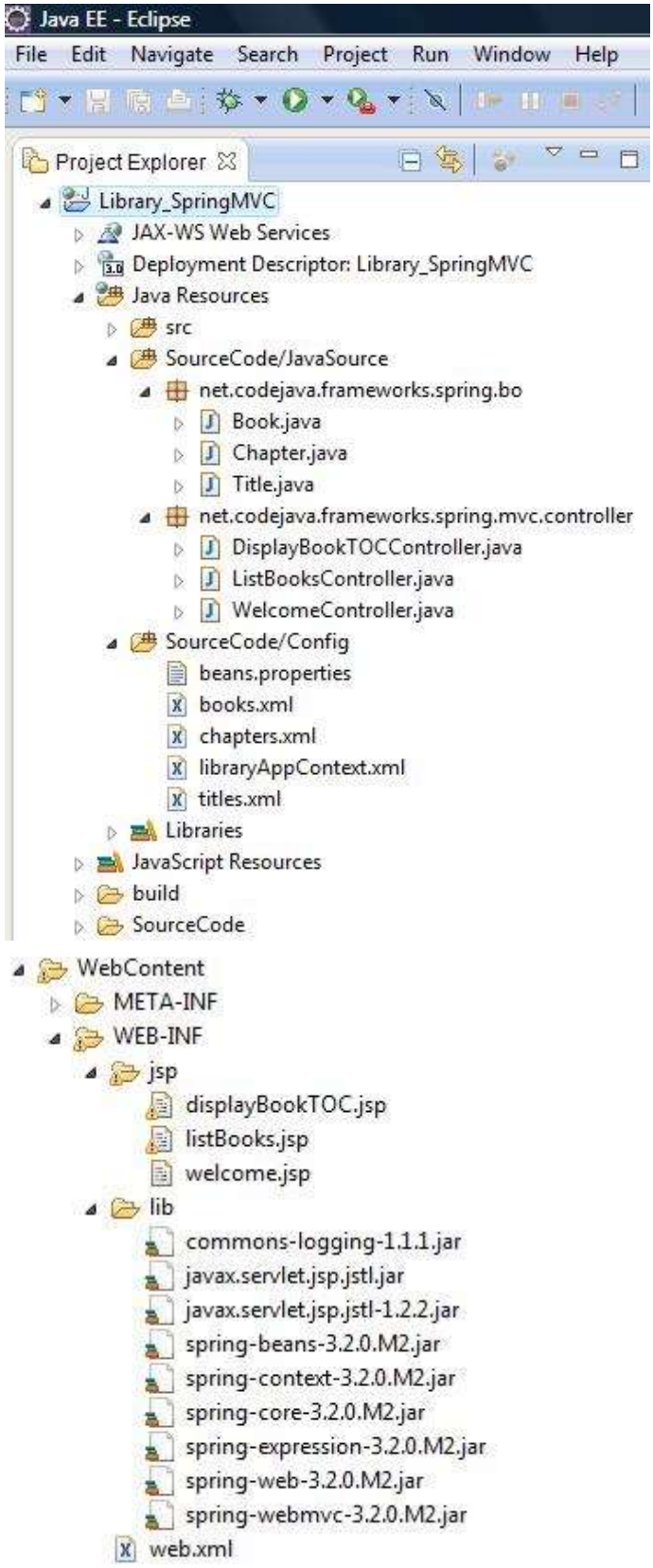
listBooks.jsp

```
<%@ page language="java"
     contentType="text/html; charset=ISO-8859-1"
     pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1"/>
<title>List of books</title>
</head>
<body>
<h1 align="center">List of books in my library :</h1>
<table width="50%" align="center">
      <tr>
            <th width="20%" align="center">S. No.</th>
            <th width="50%" align="center">Title</th>
            <th width="30%" align="center">Author</th>
      </tr>
      <%int i = 1; %>
      <c:forEach items="${books}" var="book">
            <tr>
                  <td width="20%" align="center"><%= i++ %></td>
                  <td width="50%" align="center">
            <a href="/displayBookTOC.htm?isbn=${book.isbn}">${book.title.titl
          </td>
                  <td width="30%" align="center">${book.author}</td>
            </tr>
      </c:forEach>
</table>
</body>
</html>
```

displayBookTOC.jsp

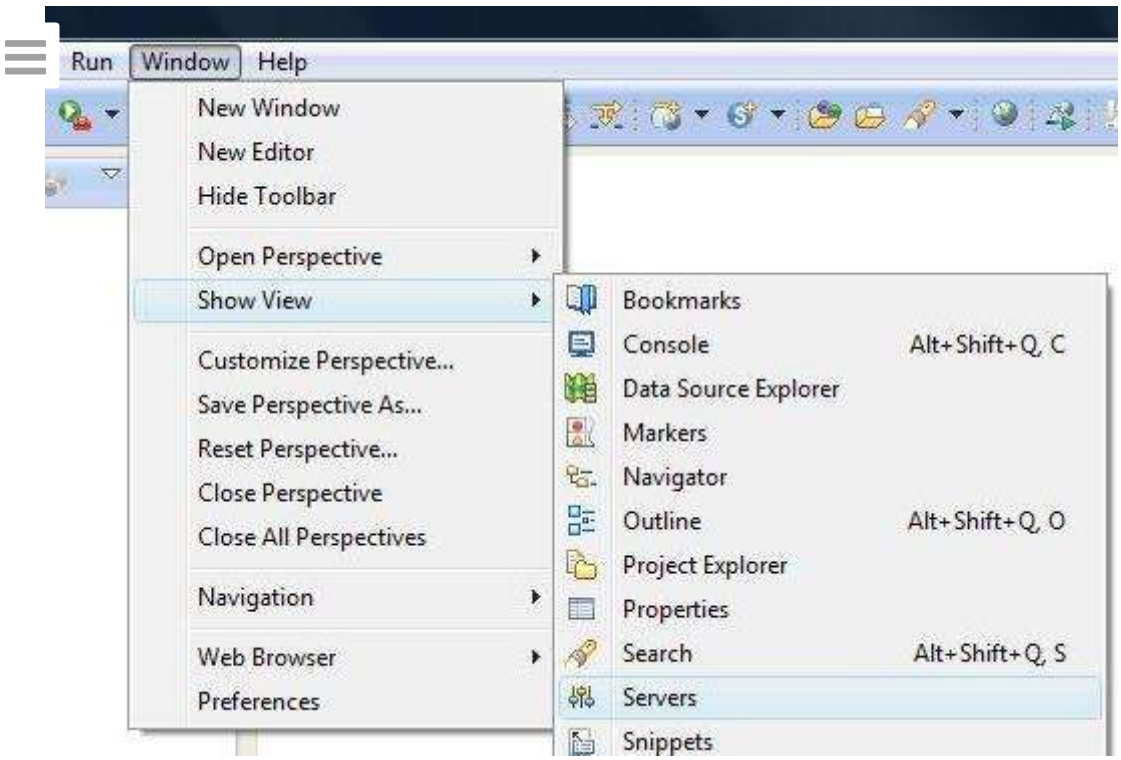
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEnc
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Book Table of Content</title>
</head>
<body>
<h1 align="center">Book : ${book.title.titleValue}, written by ${book.author
<h2>Table of content : </h2>
<%int i = 1; %>
<table width="100%">
      <c:forEach items="${book.chapters}" var="chapter">
            <tr width="100%">
                  <td width="5%" align="center"><%= i++ %></td>
                  <td width="95%" align="left">${chapter.title.titleValue}</t
            </tr>
      </c:forEach>
</table>
</body>
</html>
```

Finally our **Library_SpringMVC** project in **Eclipse IDE** will look as below.

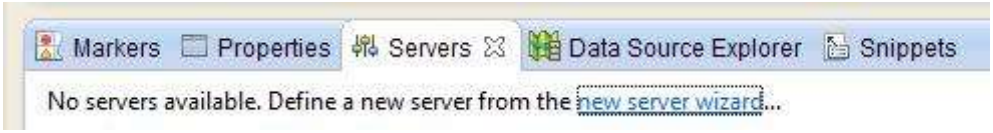


Create Tomcat Server in Eclipse IDE

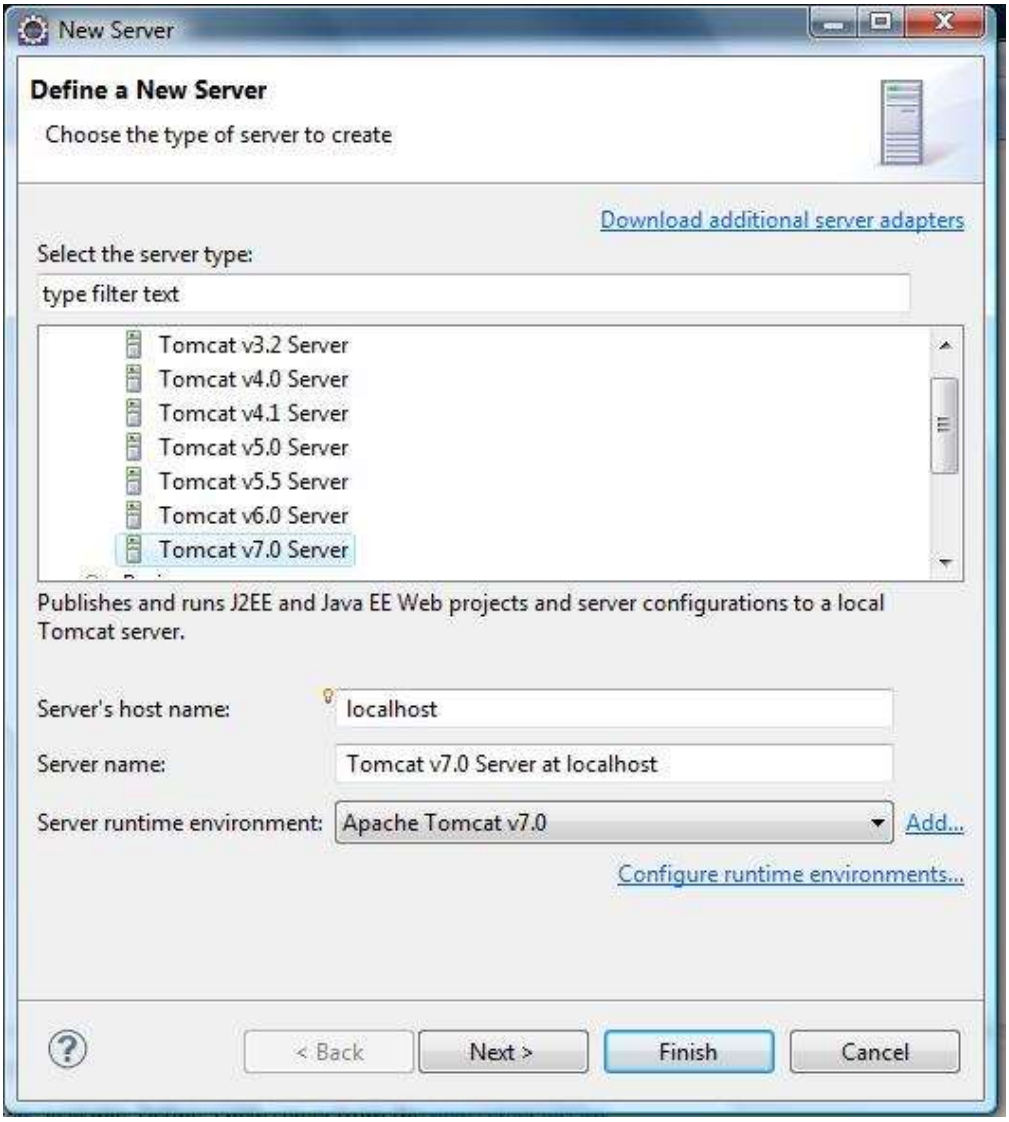
From the menu bar, select **Windows > Show View > Servers**. The server view gets opened up.



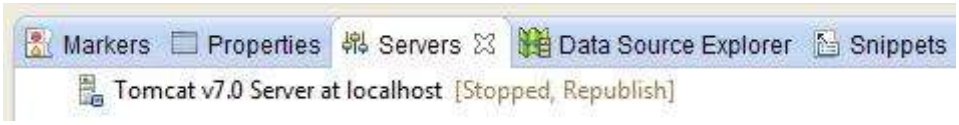
Click the link *new server wizard* on the server view.



In the pop up window which gets open up, Enter Server's host name as **localhost**, Server Type as **Apache > Tomcat v7.0 Server** and Server runtime as **Apache Tomcat v7.0**. Click the finish button.

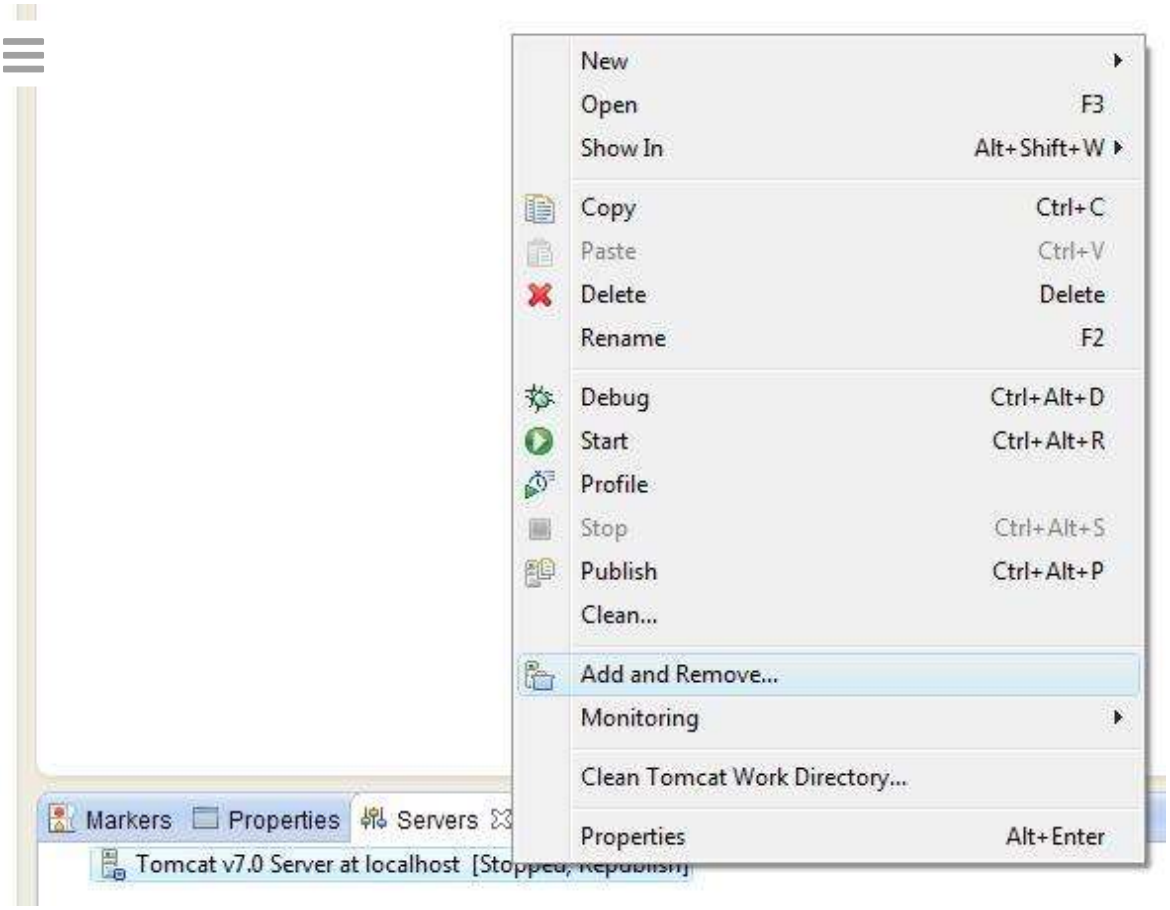


Click the **Finish** button and the Apache Tomcat v7.0 Server will start appearing in the server view.

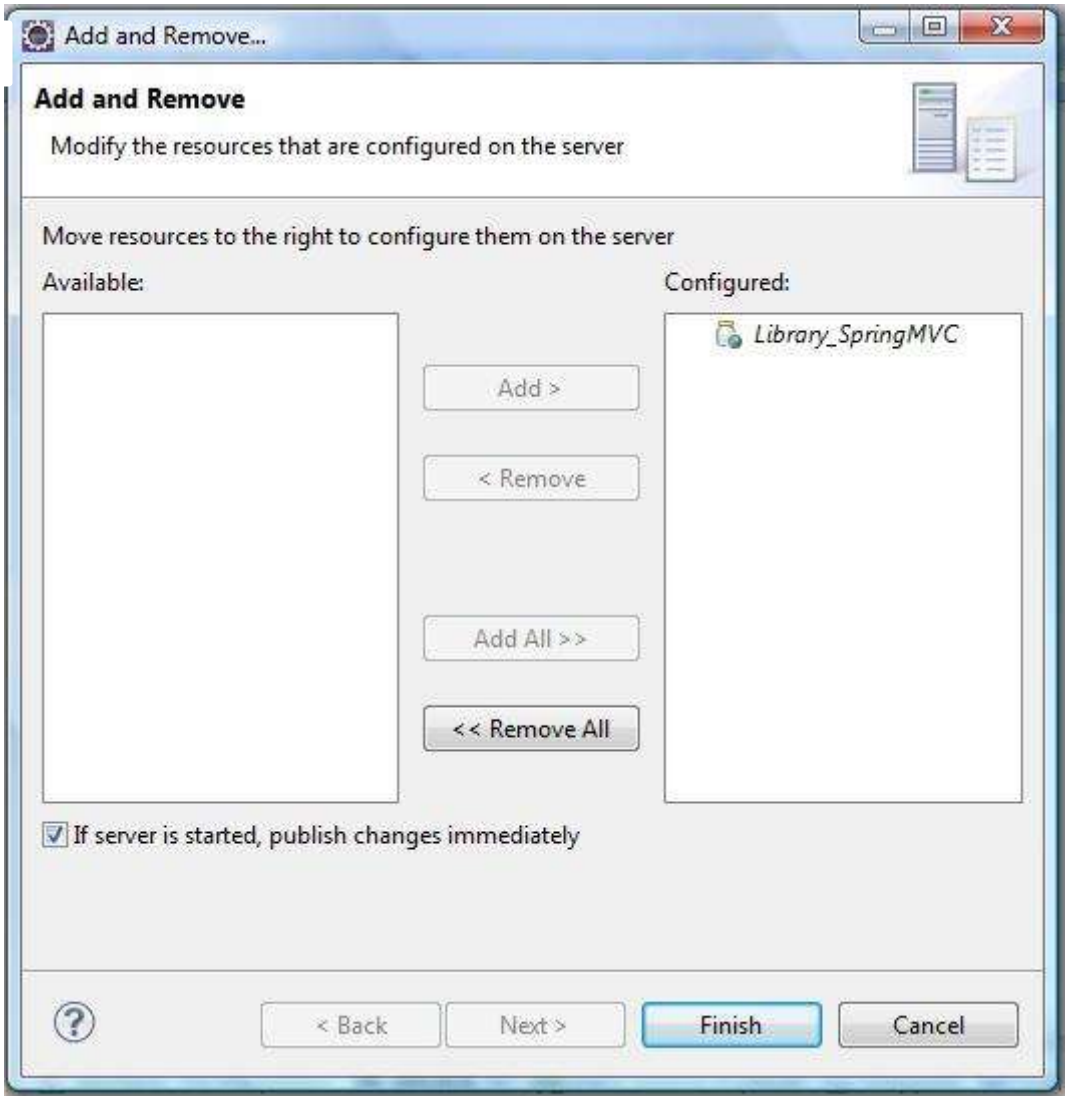


Deploying the application

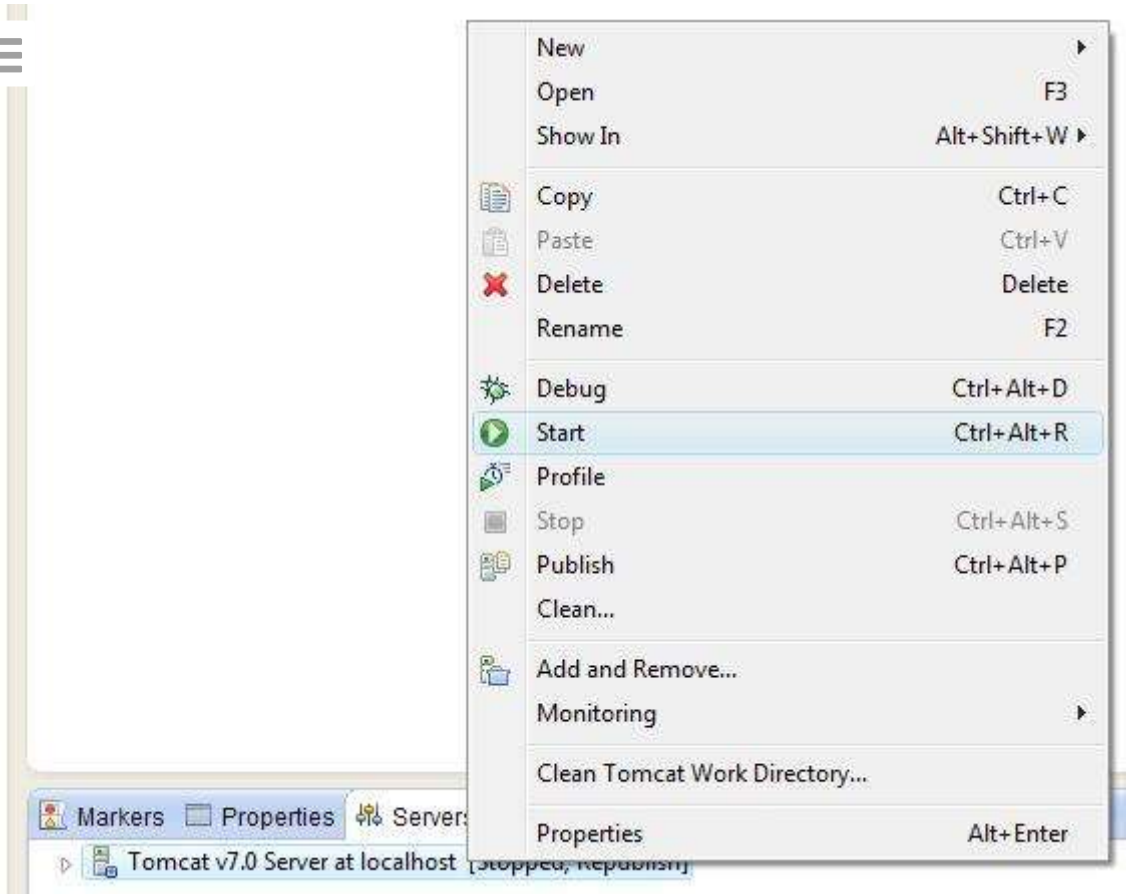
Right click on the server created above. Select **Add and Remove Projects**.



In the pop up window which gets open up, **Library_SpringMVC** project will appear on the list of **Available projects**. Select the **Library_SpringMVC** project and move it to **Configured projects**. Click the Finish button.



Click the **Finish** button in the above dialogue box. Now right click the server and select Start.

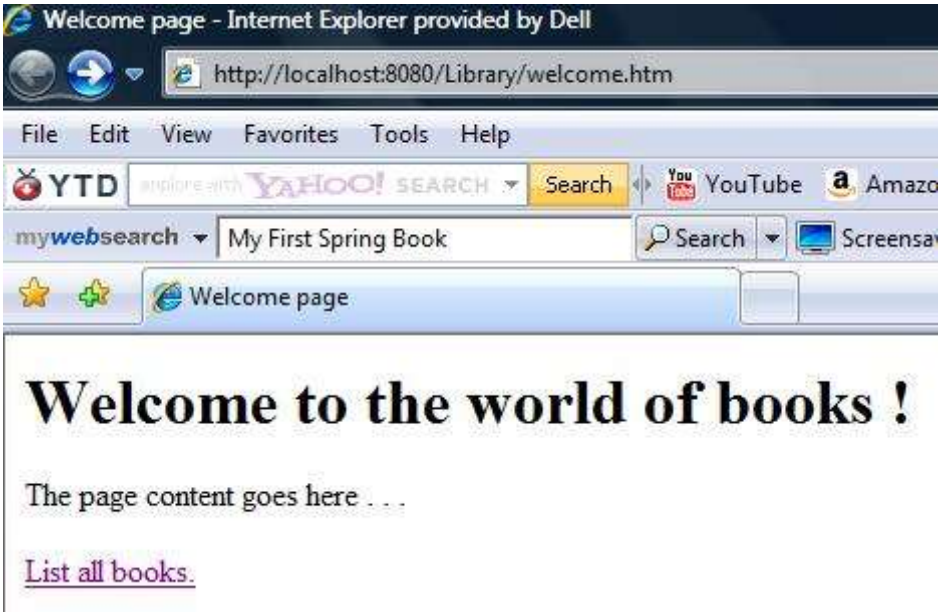


Our library application is now deployed on the Apache Tomcat server.

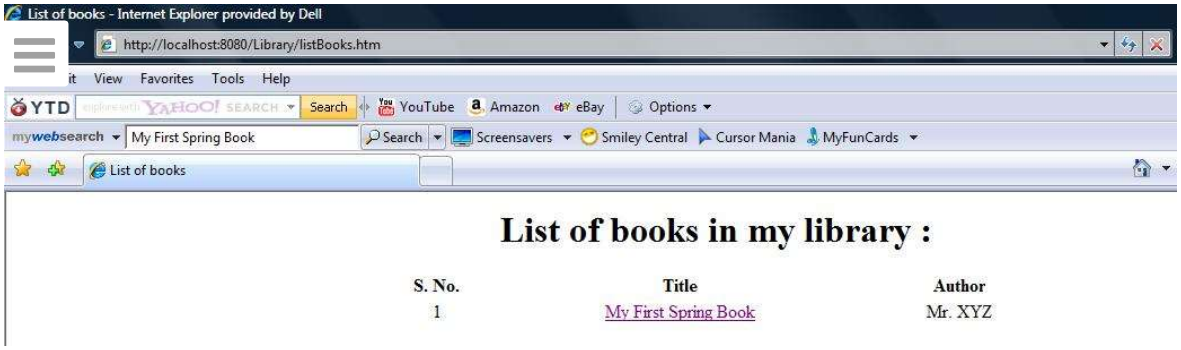
Testing the application

Open the web browser. Enter the following URL (Note that the Tomcat server is running on port 8080). http://localhost:8080/Library/welcome.htm

The **welcome** page of **Library** application will be displayed on the browser.



Click the hyperlink *List all books* (http://localhost:8080/Library/listBooks.htm) on the welcome page. This will show the *listBooks.htm* page on the browser with the list of books (Model).



Click the hyperlink *My First Spring Book* (http://localhost:8080/Library/displayBookTOC.htm?isbn=1). The browser will render the *displayBookTOC.htm* page of our library application with the details of the book with ISBN = 1 (Model).

8. Conclusion

In this article we have learned how the Spring framework makes the development of web application very simple and easy. We have learned the Spring's MVC module in detail, its architecture and various components. We have also learned the design patterns on which Spring's MVC module is based upon. Finally we have built a simple application using Eclipse IDE.

Other Spring Tutorials:

- Understand the core of Spring framework
- Understand Spring AOP
- Spring Dependency Injection Example with XML Configuration
- Spring MVC beginner tutorial with Spring Tool Suite IDE
- Spring MVC Form Handling Tutorial
- Spring MVC Form Validation Tutorial
- 14 Tips for Writing Spring MVC Controller
- Spring Web MVC Security Basic Example (XML Configuration)
- Understand Spring Data JPA with Simple Example

About the Author: