

2018

ACE Inspiration

Professional Development Center

Spring MVC And Basic Tutorial

TUTORIAL GUIDE
TUTORIALS POINT

About the Tutorial

Spring MVC Framework is an open source Java platform that provides comprehensive infrastructure support for developing robust Java based Web applications very easily and very rapidly.

Spring Framework was initially written by Rod Johnson and was first released under the Apache 2.0 license in June 2003. This tutorial is written based on the Spring Framework Version 4.1.6 released in March 2015.

Audience

This tutorial is designed for Java programmers with a need to understand the Spring MVC Framework in detail along with its architecture and actual usage. This tutorial is intended to make you comfortable in getting started with the Spring MVC Framework and its various functions.

Prerequisites

This tutorial is designed for Java programmers with a need to understand the Spring MVC Framework in detail along with its architecture and actual usage. This tutorial will bring you at the intermediate level of expertise from where you can take yourself to a higher level of expertise.

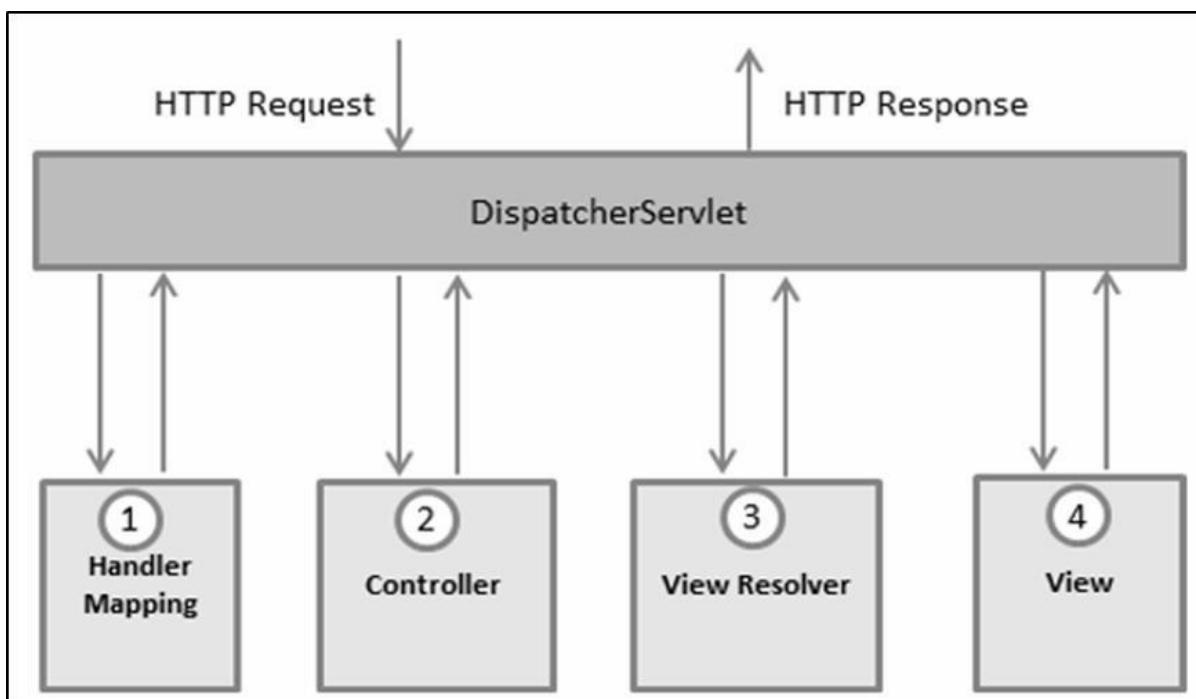
1. SpringMVC–Overview

The Spring Web MVC framework provides a model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general, they will consist of **POJO**.
- The **View** is responsible for rendering the model data and in general, it generates **HTML Output** that the client's browser can interpret.
- The **Controller** is responsible for processing **User Requests** and **Building Appropriate Model** and passes it to the view for rendering.

The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is shown in the following illustration.



Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet:

- After receiving an HTTP request, DispatcherServlet consults the **HandlerMapping** to call the appropriate Controller.
- The Controller takes the request and calls the appropriate service methods based on used **GET** or **POST method**. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
- The DispatcherServlet will take help from **ViewResolver** to pick up the defined view for the request.
- Once view is finalized, The DispatcherServlet passes the model data to the view, which is finally rendered, on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller and ViewResolver are parts of **WebApplicationContext**, which is an extension of the plain **ApplicationContext** with some extra features necessary for web applications.

Required Configuration

We need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the **web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** DispatcherServlet:

```
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Spring MVC Application</display-name>

    <servlet>
        <servlet-name>HelloWeb</servlet-name>
        <servlet-class> org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloWeb</servlet-name>
        <url-pattern>*.jsp</url-pattern>
    </servlet-mapping>

</web-app>
```

The **web.xml** file will be kept in the **WebContent/WEB-INF** directory of your web application. Upon initialization of the **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INF directory. In this case, our file will be **HelloWeb-servlet.xml**.

Next, the **<servlet-mapping>** tag indicates which URLs will be handled by which DispatcherServlet. Here, all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with the default filename as **[servlet-name]-servlet.xml** and default location as WebContent/WEB-INF, you can customize this file name and location by adding the servlet listener **ContextLoaderListener** in your web.xml file as follows:

```
<web-app...>

<!----- DispatcherServlet definition goes here ---- >
.....
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
</context-param>

<listener>
    <listener-class> org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
</web-app>
```

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's WebContent/WEB-INF directory.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.ai" />
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" /> </bean>
    </beans>

```

Following are some important points about **HelloWeb-servlet.xml** file:

- The **[servlet-name]-servlet.xml** file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
- The **<context:component-scan...>** tag will be used to activate the Spring MVC annotation scanning capability, which allows to make use of annotations like **@Controller** and **@RequestMapping**, etc.
- The **InternalResourceViewResolver** will have rules defined to resolve the view names. As per the above-defined rule, a logical view named **hello** is delegated to a view implementation located at **/WEB-INF/jsp/hello.jsp**.

Let us now understand how to create the actual components i.e., Controller, Model and View.

Defining a Controller

The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller** annotation indicates that a particular class serves the role of a controller. The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```

@Controller
@RequestMapping("/hello") public
class HelloController{

    @RequestMapping(method = RequestMethod.GET) public String
    printHello(ModelMap model) { model.addAttribute("message",
    "Hello Spring MVC Framework!"); return "hello";
} }

```

The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path.

The next annotation **@RequestMapping (method = RequestMethod.GET)** is used to declare the **printHello()** method as the controller's default service method to handle HTTP GET request. We can define another method to handle any POST request at the same URL.

We can also write the above controller in another form, where we can add additional attributes in the **@RequestMapping** as follows:

```

@Controller
public class HelloController{

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
}

```

```

public String printHello(ModelMap model) {
    model.addAttribute("message", "Hello Spring MVC Framework!");
    return "hello";
}
}

```

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle the HTTP GET request.

Following are some important points to be noted regarding the controller defined above:

- You will define the required business logic inside a service method. You can call another method inside this method as per the requirement.
- Based on the business logic defined, you will create a **model** within this method. You can set different model attributes and these attributes will be accessed by the view to present the result. This example creates a model with its attribute "message".

- A defined service method can return a String, which contains the name of the **view** to be used to render the model. This example returns "hello" as the logical view name.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - **JSPs**, **HTML**, **PDF**, **Excel Worksheets**, **XML**, **Velocity Templates**, **XSLT**, **JSON**, **Atom** and **RSS** feeds, **JasperReports**, etc. However, the most common ones are the JSP templates written with JSTL. So, let us write a simple **hello** view in /WEB-INF/hello/hello.jsp:

```
<html>
  <head>
    <title>Hello Spring MVC</title>
  </head>
  <body>

    <h2>${message}</h2>
  </body>
</html>
```

Here **\${message}** is the attribute, which we have setup inside the Controller. You can have multiple attributes to be displayed inside your view.

2. SpringMVC– Environment Setup

This chapter will guide us on how to prepare a development environment to start your work with the Spring Framework. This chapter will also teach us how to setup **JDK**, **Tomcat** and **Eclipse** on your machine before you setup the Spring Framework:

Step 1 - Setup Java Development Kit (JDK)

You can download the latest version from Oracle's Java site: [Java SE Downloads](#). You will find instructions for installing JDK in downloaded files, follow the given instructions to install and configure the setup. Once done with the setup, set PATH and JAVA_HOME environment variables to refer to the directory that contains **java** and **javac**, typically **java_install_dir/bin** and **java_install_dir** respectively.

If you are running Windows and installed the JDK in **C:\jdk1.6.0_15**, you would have to put the following line in your **C:\autoexec.bat file**.

```
set PATH=C:\jdk1.6.0_15\bin;%PATH% set  
JAVA_HOME=C:\jdk1.6.0_15
```

Alternatively, on Windows NT/2000/XP, you could also right-click on My Computer → select Properties → Advanced → Environment Variables. Then, you would update the PATH value and click on the OK button.

On UNIX (Solaris, Linux, etc.), if the SDK is installed in **/usr/local/jdk1.6.0_15** and you use the C shell, then you should key-in the following command into your **.cshrc** file.

```
setenv PATH /usr/local/jdk1.6.0_15/bin:$PATH setenv  
JAVA_HOME /usr/local/jdk1.6.0_15
```

Alternatively, if you use an Integrated Development Environment (IDE) like **Borland JBuilder**, **Eclipse**, **IntelliJ IDEA** or **Sun ONE Studio**, then compile and run a simple program to confirm that the IDE knows where Java is installed, otherwise do proper setup as given in the documents of IDE.

Step 2 - Install Apache Common Logging API

You can download the latest version of Apache Commons Logging API from – <http://commons.apache.org/logging/>. Once you have downloaded the installation, unpack the binary distribution into a convenient location.

For example – C:\commons-logging-1.1.1 on windows, or /usr/local/commons-logging-1.1.1 on Linux/Unix. This directory will have the following jar files and other supporting documents, etc.

Name	Date modified	Type	Size
site	11/22/2007 12:28 ...	File folder	
commons-logging-1.1.1	11/22/2007 12:28 ...	WinRAR archive	60 KB
commons-logging-1.1.1-javadoc	11/22/2007 12:28 ...	WinRAR archive	139 KB
commons-logging-1.1.1-sources	11/22/2007 12:28 ...	WinRAR archive	74 KB
commons-logging-adapters-1.1.1	11/22/2007 12:28 ...	WinRAR archive	26 KB
commons-logging-api-1.1.1	11/22/2007 12:28 ...	WinRAR archive	52 KB
commons-logging-tests	11/22/2007 12:28 ...	WinRAR archive	109 KB
LICENSE	11/22/2007 12:27 ...	Text Document	12 KB
NOTICE	11/22/2007 12:27 ...	Text Document	1 KB
RELEASE-NOTES	11/22/2007 12:27 ...	Text Document	8 KB

Make sure you set your CLASSPATH variable on this directory properly, otherwise you will face problem while running your application.

Step 3 - Setup Eclipse IDE

All the examples in this tutorial have been written using the Eclipse IDE. Therefore, it is recommended that we should have the latest version of Eclipse installed on the machine.

To install Eclipse IDE, download the latest Eclipse binaries from the following link – <http://www.eclipse.org/downloads/>. Once the installation is downloaded, unpack the binary distribution into a convenient location.

For example in – C:\eclipse on windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

Eclipse can be started by executing the following commands on a windows machine, or we can simply double click on the eclipse.exe.

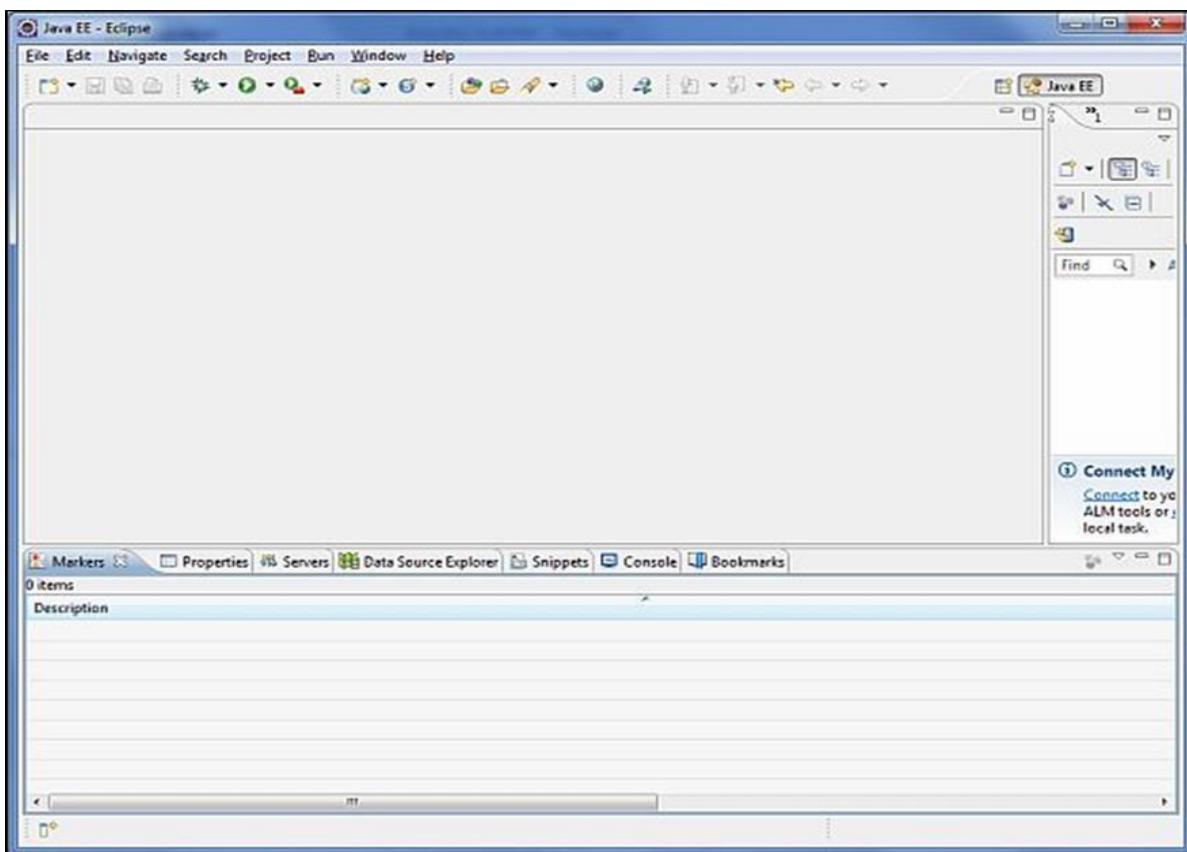
```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on a UNIX (Solaris, Linux, etc.) machine:

```
$/usr/local/eclipse/eclipse
```

Spring MVC

After a successful startup, if everything is fine, then it should display the following screen.



Step 4 - Setup Spring Framework Libraries

Now if everything is fine, then we can proceed to setup the Spring Framework. Following are the steps to download and install the framework on the machine.

- Make a choice whether you want to install Spring on Windows or UNIX and then proceed to the next step to download **.zip** file for windows and **.tz** file for Unix.
- Download the latest version of Spring framework binaries from – <http://repo.spring.io/release/org/springframework/spring>.
- We have downloaded the **spring-framework-4.3.1.RELEASE-dist.zip** on the Windows Machine and when we unzip the downloaded file, it will give out the directory structure inside – E:\spring as follows.

Name	Date modified	Type	Size
docs	4/22/2015 2:44 PM	File folder	
libs	4/22/2015 2:45 PM	File folder	
schema	4/22/2015 2:45 PM	File folder	
license	4/22/2015 2:42 PM	Text Document	15 KB
notice	4/22/2015 2:42 PM	Text Document	1 KB
readme	4/22/2015 2:42 PM	Text Document	1 KB

You will find all the Spring libraries in the directory **E:\spring\libs**. Make sure you set your CLASSPATH variable on this directory properly; otherwise, we will face a problem while running the application. If we use Eclipse, then it is not required to set the CLASSPATH because all the setting will be done through Eclipse.

Once you are done with this last step, you are ready to proceed for your first Spring Example, which you will see in the next chapter.

3. SpringMVC– HelloWorld

The following example shows how to write a simple web based **Hello World** application using the Spring MVC Framework. To start with, let us have a working Eclipse IDE in place and follow the subsequent steps to develop a Dynamic Web Application using the Spring Web Framework.

Step	Description
1	Create a Dynamic Web Project with a name HelloWeb and create a package com.ai under the src folder in the created project.
2	Drag and drop the following Spring and other libraries into the folder WebContent/WEB-INF/lib .
3	Create a Java class HelloController under the com.ai package.
4	Create Spring configuration files web.xml and HelloWeb-servlet.xml under the WebContent/WEB-INF folder.
5	Create a sub-folder with a name jsp under the WebContent/WEB-INF folder. Create a view file hello.jsp under this sub-folder.
6	The final step is to create the content of the source and configuration files and export the application as explained below.

HelloController.java

```
package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.ui.ModelMap;

@Controller
@RequestMapping("/hello") public
class HelloController{

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

```
public String printHello(ModelMap model) {
    model.addAttribute("message", "Hello Spring MVC Framework!");

    return "hello";
}
```

web.xml

```

<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Spring MVC Application</display-name>

  <servlet>
    <servlet-name>HelloWeb</servlet-name>
    <servlet-class> org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloWeb</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>

```

HelloWeb-servlet.xml

```

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

```

```

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

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

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

```

hello.jsp

```

<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
<title>Hello World</title>
</head> <body>
    <h2>${message}</h2>
</body> </html>

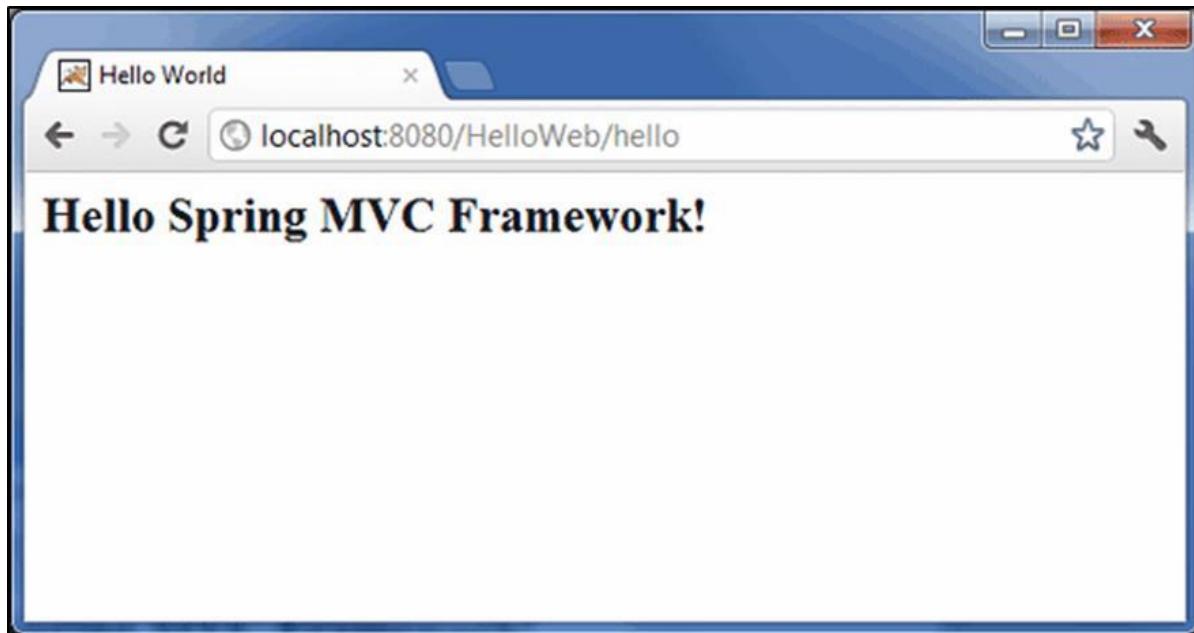
```

Following is the list of Spring and other libraries to be included in the web application. We can just drag these files and drop them in – **WebContent/WEB-INF/lib** folder.

- servlet-api-x.y.z.jar
- commons-logging-x.y.z.jar
- spring-aop-x.y.z.jar
- spring-beans-x.y.z.jar
- spring-context-x.y.z.jar
- spring-core-x.y.z.jar
- spring-expression-x.y.z.jar
- spring-webmvc-x.y.z.jar
- spring-web-x.y.z.jar

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save your **HelloWeb.war** file in Tomcat's **webapps** folder.

Now start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Now, try to access the URL – **http://localhost:8080/HelloWeb/hello**. If everything is fine with the Spring Web Application, we will see the following screen.



You should note that in the given URL, **HelloWeb** is the application name and **hello** is the virtual subfolder, which we have mentioned in our controller using `@RequestMapping("/hello")`. You can use direct root while mapping your URL using `@RequestMapping("/")`, in this case you can access the same page using short URL **http://localhost:8080/HelloWeb/**, but it is advised to have different functionalities under different folders.

Spring MVC – Form Handling

4. SpringMVC – Form Handling

The following example explains how to write a simple web based application, which makes use of HTML forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and follow the subsequent steps to develop a Dynamic Form based Web Application using Spring Web Framework:

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes Student, StudentController under the com.ai package.
3	Create view files student.jsp, result.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

Student.java

```
package com.ai;

public class Student { private
    Integer age; private String
    name; private Integer id;

    public void setAge(Integer age) { this.age
        = age;
    }
    public Integer getAge() { return
        age;
    }
    public void setName(String name) { this.name
        = name;
    }
    public String getName() { return
        name;
    }
```

```
    public void setId(Integer id) { this.id
        = id;
    }
    public Integer getId() { return
        id;
    }
}
```

StudentController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class StudentController {

    @RequestMapping(value = "/student", method = RequestMethod.GET)
    public ModelAndView student() { return new
    ModelAndView("student", "command", new Student());
    }

    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute("SpringWeb")Student student,
    ModelMap model) { model.addAttribute("name",
    student.getName()); model.addAttribute("age",
    student.getAge()); model.addAttribute("id",
    student.getId());

    return "result"; }
}

```

Here, the first service method **student()**, we have passed a blank Student object in the ModelAndView object with name "command". This is done because the spring framework expects an object with name "command", if we use <form:form> tags in the JSP file. So, when the student() method is called, it returns student.jsp view.

The second service method **addStudent()** will be called against a POST method on the HelloWeb/addStudent URL. You will prepare your model object based on the submitted information. Finally, a "result" view will be returned from the service method, which will result in rendering result.jsp.

student.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Student Information</h2>
<form:form method="POST" action="/HelloWeb/addStudent">
    <table> <tr>
        <td><form:label path="name">Name</form:label></td>
        <td><form:input path="name" /></td>
    </tr> <tr>
        <td><form:label path="age">Age</form:label></td>
        <td><form:input path="age" /></td>
    </tr> <tr>
        <td><form:label path="id">id</form:label></td>
        <td><form:input path="id" /></td>
    </tr> <tr>
        <td colspan="2">
            <input type="submit" value="Submit"/> </td>
        </tr>
    </table>
</form:form>
</body>
</html>
```

result.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted Student Information</h2>
<table> <tr>
    <td>Name</td>
    <td>${name}</td>
</tr> <tr>
    <td>Age</td>
    <td>${age}</td>
</tr> <tr>
    <td>ID</td>
    <td>${id}</td>
</tr>
</table>
</body> </html>

```

Once we are done with creating source and configuration files, export your application. Right click on your application, use Export → WAR File option and save the **SpringWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Now, try a URL—
<http://localhost:8080/SpringWeb/student> and you should see the following screen if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Exception Handing". The address bar displays "localhost:8080/HelloWeb/student". The main content area is titled "Student Information". It contains three input fields: "Name" with value "ZARA", "Age" with value "10", and "id" with value "12234". Below these fields is a "Submit" button.

After submitting the required information, click on the submit button to submit the form. You should see the following screen, if everything is fine with your Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handling". The address bar displays "localhost:8080/HelloWeb/addStudent". The main content area is titled "Submitted Student Information". It displays the submitted student information: Name ZARA, Age 10, and ID 12234.

5. SpringMVC- PageRedirection

The following example shows how to write a simple web based application, which makes use of redirect to transfer an http request to another page. To start with, let us have a working Eclipse IDE in place and consider the following steps to develop a Dynamic Form based Web Application using Spring Web Framework:

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create a Java class WebController under the com.ai package.
3	Create view files index.jsp, final.jsp under jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

WebController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class WebController {

    @RequestMapping(value = "/index", method = RequestMethod.GET)
    public String index() { return "index";
    }

    @RequestMapping(value = "/redirect", method = RequestMethod.GET)

    public String redirect() { return
        "redirect:finalPage";
    }
}

```

```

@RequestMapping(value = "/finalPage", method = RequestMethod.GET)
public String finalPage() {

    return "final";
}
}

```

Following is the content of Spring view file **index.jsp**. This will be a landing page, this page will send a request to the access-redirect service method, which will redirect this request to another service method and finally a **final.jsp** page will be displayed.

index.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring Page Redirection</title>
</head>
<body>
    <h2>Spring Page Redirection</h2>
    <p>Click below button to redirect the result to new page</p>
    <form:form method="GET" action="/HelloWeb/redirect"> <table>
        <tr>
            <td>
                <input type="submit" value="Redirect Page"/>
            </td>
        </tr>
    </table>
    </form:form>
</body>
</html>
```

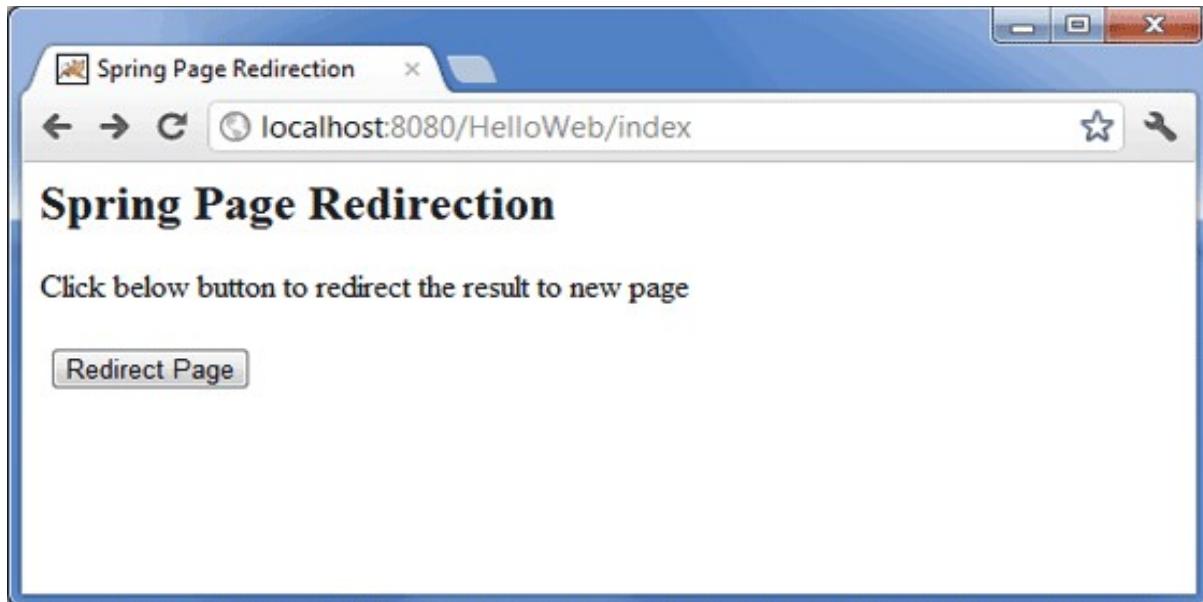
final.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring Page Redirection</title>
</head>
<body>
    <h2>Redirected Page</h2>
```

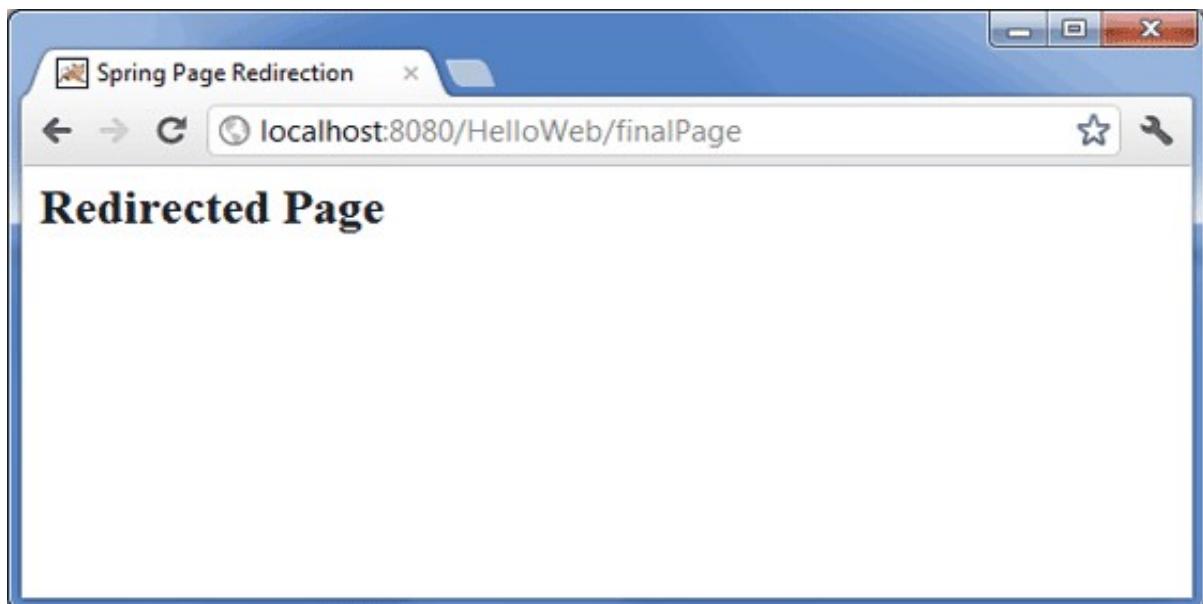
```
</body>
</html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use Export → WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL –<http://localhost:8080>HelloWeb/index> and you should see the following screen if everything is fine with the Spring Web Application.



Now click on the "Redirect Page" button to submit the form and to get to the final redirected page. We should see the following screen, if everything is fine with our Spring Web Application:



6. SpringMVC– StaticPages

The following example shows how to write a simple web based application using Spring MVC Framework, which can access static pages along with dynamic pages with the help of a **<mvc:resources>** tag.

To begin with, let us have a working Eclipse IDE in place and adhere to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create a Java class WebController under the com.ai package.
3	Create a static file final.htm under jsp sub-folder.
4	Update the Spring configuration file HelloWeb-servlet.xml under the WebContent/WEB-INF folder as shown below.
4	The final step is to create the content of the source and configuration files and export the application, which is explained below.

WebController.java

```
package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class WebController {

    @RequestMapping(value = "/index", method = RequestMethod.GET)
    public String index() { return "index"; }
}
```

```

}

@RequestMapping(value = "/staticPage", method = RequestMethod.GET)
public String redirect() {

    return "redirect:/pages/final.htm"; }

HelloWeb-servlet.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"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

    <mvc:resources mapping="/pages/**" location="/WEB-INF/pages/" />
    <mvc:annotation-driven/>

</beans>
```

Here, the **<mvc:resources..../>** tag is being used to map static pages. The mapping attribute must be an **Ant pattern** that specifies the URL pattern of an http requests. The location attribute must specify one or more valid resource directory locations having static pages including images, stylesheets, JavaScript, and other static content. Multiple resource locations may be specified using a comma-separated list of values.

Following is the content of Spring view file **WEB-INF/jsp/index.jsp**. This will be a landing page; this page will send a request to access the **staticPage service method**, which will redirect this request to a static page available in WEB-INF/pages folder.

index.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring Landing Page</title>
</head>
<body>
    <h2>Spring Landing Page</h2>
    <p>Click below button to get a simple HTML page</p>
    <form:form method="GET" action="/HelloWeb/staticPage"> <table>
        <tr>
            <td>
                <input type="submit" value="Get HTML Page"/>
            </td>
        </tr>
    </table>
    </form:form>
</body>
</html>
```

final.htm

```
<html>
<head>
    <title>Spring Static Page</title>
</head>
<body>

    <h2>A simple HTML page</h2>

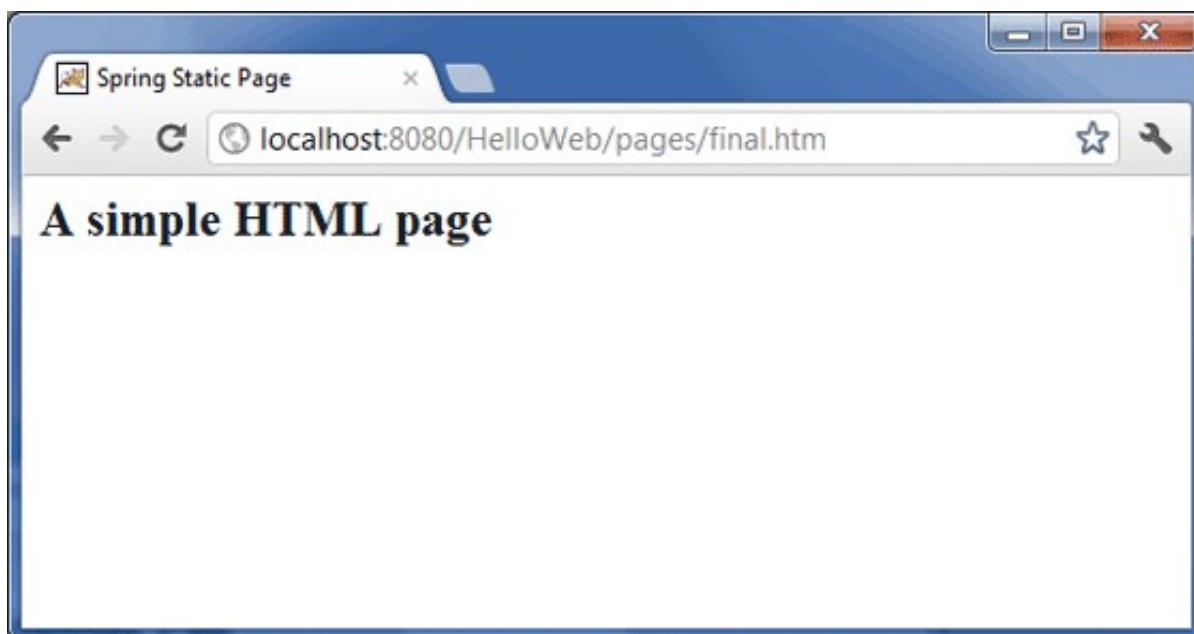
</body> </html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use Export → WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Now try to access the URL – <http://localhost:8080>HelloWeb/index>. If everything is fine with the Spring Web Application, we will see the following screen.



Click on "Get HTML Page" button to access a static page mentioned in the staticPage service method. If everything is fine with your Spring Web Application, we will see the following screen.



Spring MVC – Form Tag Library

7. SpringMVC– TextBox

The following example shows how to use Text boxes in forms using the Spring Web MVC framework. To begin with, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework:

Step	Description
1	Create a project with a name HelloWeb under a package com.aias explained in the Spring MVC - Hello World Example chapter.
2	Create a Java classes Student, StudentController under the com.ai package.
3	Create a view files student.jsp, result.jsp under jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

Student.java

```
package com.ai;

public class Student { private
    Integer age; private String
    name; private Integer id;

    public void setAge(Integer age) { this.age
        = age;
    }
    public Integer getAge() { return
        age;
    }

    public void setName(String name) { this.name
        = name;
    }
    public String getName() { return
        name;
    }
```

```
}
```

```
    public void setId(Integer id) { this.id
        = id;
    }
    public Integer getId() { return
        id;
    }
}
```

StudentController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class StudentController {
    @RequestMapping(value = "/student", method = RequestMethod.GET)
    public ModelAndView student() { return new
        ModelAndView("student", "command", new Student()); }

    @RequestMapping(value = "/addStudent", method = RequestMethod.POST) public
    String addStudent(@ModelAttribute("SpringWeb")Student student,
    ModelMap model) { model.addAttribute("name",
        student.getName()); model.addAttribute("age",
        student.getAge()); model.addAttribute("id",
        student.getId()); return "result";
    }
}

```

Here, the first service method **student()**, we have passed a blank Studentobject in the ModelAndView object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when the **student()** method is called it returns **student.jsp** view.

The second service method **addStudent()** will be called against a POST method on the **HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally, a "result" view will be returned from the service method, which will result in rendering result.jsp

student.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Student Information</h2>
<form:form method="POST" action="/HelloWeb/addStudent">
    <table> <tr>
        <td><form:label path="name">Name</form:label></td>
        <td><form:input path="name" /></td>
    </tr> <tr>
        <td><form:label path="age">Age</form:label></td>
        <td><form:input path="age" /></td>
    </tr> <tr>
        <td><form:label path="id">id</form:label></td>
        <td><form:input path="id" /></td>
    </tr> <tr>
        <td colspan="2">
            <input type="submit" value="Submit"/> </td>
        </tr>
    </table>
</form:form>
</body>
</html>

```

Here, we are using **<form:input />** tag to render an HTML text box. For example –

<form:input path="name" />

It will render the following HTML content.

<input id="name" name="name" type="text" value="" />

result.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted Student Information</h2>
<table> <tr>
    <td>Name</td>
    <td>${name}</td>
</tr> <tr>
    <td>Age</td>
    <td>${age}</td>
</tr> <tr>
    <td>ID</td>
    <td>${id}</td>
</tr>
</table>
</body> </html>

```

Once we are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **HelloWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL – **http://localhost:8080/HelloWeb/student** and we will see the following screen if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Exception Han". The address bar displays "localhost:8080/HelloWeb/student". The main content area is titled "Student Information". It contains three input fields: "Name" with value "ZARA", "Age" with value "10", and "id" with value "12234". Below these fields is a "Submit" button.

After submitting the required information, click on the submit button to submit the form. We should see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handling". The address bar displays "localhost:8080/HelloWeb/addStudent". The main content area is titled "Submitted Student Information". It displays the submitted data:
Name ZARA
Age 10
ID 12234

8. SpringMVC–Password

The following example describes how to use Password in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and adhere to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai; public  
class User { private String  
username; private String  
password;  
  
    public String getUsername() { return  
        username;  
    }  
    public void setUsername(String username) {  
        this.username = username;  
    }  
    public String getPassword() { return  
        password;  
    }  
    public void setPassword(String password) {  
        this.password = password;  
    }  
}
```

UserController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public ModelAndView user() { return new ModelAndView("user",
    "command", new User()); }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("SpringWeb")User user,
    ModelMap model) {
        model.addAttribute("username", user.getUsername());
        model.addAttribute("password", user.getPassword());

        return "users";
    }
}

```

Here, the first service method **user()**, we have passed a blank User object in the ModelAndView object with name "command", because the spring framework expects an object with name "command", if you are using <form:form> tags in your JSP file. So, when the user() method is called it returns user.jsp view.

The second service method **addUser()** will be called against a POST method on the HelloWeb/addUser URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>

```

```
<title>Spring MVC Form Handling</title>
</head>
<body>

<h2>User Information</h2>
<form:form method="POST" action="/HelloWeb/addUser"> <table>
    <tr>
        <td><form:label path="username">User Name</form:label></td>
        <td><form:input path="username" /></td>
    </tr> <tr>
        <td><form:label path="password">Age</form:label></td>
        <td><form:password path="password" /></td>
    </tr> <tr>
        <td colspan="2">
            <input type="submit" value="Submit"/> </td>
        </tr>
    </table>
</form:form>
</body>
</html>
```

Here, we are using the `<form:password />` tag to render an HTML password box. For example –

```
<form:password path="password" />
```

It will render the following HTML content.

```
<input id="password" name="password" type="password" value="" />
```

users.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>
```

```
<h2>Submitted User Information</h2> <table>
<tr>
    <td>Username</td>
    <td>${username}</td>
</tr> <tr>
    <td>Password</td>
    <td>${password}</td>
</tr>
</table>
</body>
</html>
```

Once we are done with creating source and configuration files, export the application. Right click on your application, use Export → WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL –<http://localhost:8080>HelloWeb/user> and we will see the following screen if everything is fine with the Spring Web Application.

A screenshot of a web browser window titled "Spring MVC Form Handlin". The address bar shows "localhost:8081/HelloWeb/user". The main content area displays a form titled "User Information". The form has two text input fields: "User Name" containing "Mahesh" and "Password" which is empty. Below the inputs is a "Submit" button.

After submitting the required information, click on the submit button to submit the form. We will see the following screen, if everything is fine with the Spring Web Application.

A screenshot of a web browser window titled "Spring MVC Form Handlin". The address bar shows "localhost:8081/HelloWeb/addUser". The main content area displays a message "Submitted User Information" followed by the submitted data: "Username Mahesh" and "Password Password".

9. SpringMVC–TextArea

The following example explains how to use TextArea in forms using the Spring Web MVC framework. To begin with, let us have a working Eclipse IDE in place and follow the subsequent steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai; public  
class User { private String  
username; private String  
password; private String  
address;  
  
    public String getUsername() { return  
        username;  
    }  
    public void setUsername(String username) {  
        this.username = username;  
    }  
    public String getPassword() { return  
        password;  
    }  
    public void setPassword(String password) {  
        this.password = password;  
    }  
    public String getAddress() {
```

```
        return address;  
    }  
    public void setAddress(String address) {  
        this.address = address;  
    }  
}
```

UserController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public ModelAndView user() { return new ModelAndView("user",
    "command", new User()); }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("SpringWeb")User user,
    ModelMap model) {
        model.addAttribute("username", user.getUsername());
        model.addAttribute("password", user.getPassword());
        model.addAttribute("address", user.getAddress());

        return "users";
    }
}

```

Here, for the first service method user(), we have passed a blank User object in the ModelAndView object with name "command", because the spring framework expects an object with name "command", if you are using <form:form> tags in your JSP file. So, when the user() method is called, it returns the user.jsp view.

The second service method addUser() will be called against a POST method on the HelloWeb/addUser URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

    <h2>User Information</h2>
    <form:form method="POST" action="/HelloWeb/addUser"> <table>
        <tr>
            <td><form:label path="username">User Name</form:label></td>
            <td><form:input path="username" /></td>
        </tr> <tr>
            <td><form:label path="password">Age</form:label></td>
            <td><form:password path="password" /></td>
        </tr>
        <tr>
            <td><form:label path="address">Address</form:label></td>
            <td><form:textarea path="address" rows="5" cols="30" /></td>
        </tr> <tr>
            <td colspan="2">
                <input type="submit" value="Submit"/> </td>
            </tr>
        </table>
    </form:form>
</body>
</html>

```

Here, we are using **<form:textarea />** tag to render a HTML textarea box. For example –

```
<form:textarea path="address" rows="5" cols="30" />
```

It will render the following HTML content.

```
<textarea id="address" name="address" rows="5" cols="30"></textarea>
```

users.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted User Information</h2> <table>
    <tr>
        <td>Username</td>
        <td>${username}</td>
    </tr> <tr>
        <td>Password</td>
        <td>${password}</td>
    </tr> <tr>
        <td>Address</td>
        <td>${address}</td>
    </tr>
</table>
</body>
</html>

```

Once you are done with creating source and configuration files, export your application. Right click on your application, use Export → WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL –<http://localhost:8080>HelloWeb/user> and we will see the following screen if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handlin". The address bar displays "localhost:8081/HelloWeb/user". The main content area is titled "User Information". It contains three input fields: "User Name" with value "Mahesh", "Password" with value "*****", and "Address" with value "Sample Address". Below these fields is a "Submit" button.

After submitting the required information, click on the submit button to submit the form. We will see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handlin". The address bar displays "localhost:8081/HelloWeb/addUser". The main content area is titled "Submitted User Information". It displays the submitted user information: "Username Mahesh", "Password Password", and "Address Sample Address".

10. SpringMVC–Checkbox

The following example describes how to use a Single Checkbox in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and consider the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.aias explained in the Spring MVC - Hello World Example chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create a view files user.jsp, users.jsp under jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai;

public class User {

    private String username; private String password; private String
    address; private boolean receivePaper;

    public String getUsername() { return username;
    }
    public void setUsername(String username) { this.username = username;
    }

    public String getPassword() {

        return password;
    }
    public void setPassword(String password) { this.password =
        password;
    }
    public String getAddress() { return
        address;
    }
    public void setAddress(String address) { this.address =
        address;
    }
    public boolean isReceivePaper() { return
        receivePaper;
    }
    public void setReceivePaper(boolean receivePaper) {
        this.receivePaper = receivePaper; }
}
```

UserController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestMethod; import
org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET) public
    ModelAndView user() { return new ModelAndView("user", "command", new User());
    }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST) public String
    addUser(@ModelAttribute("SpringWeb")User user,
    ModelMap model) {
        model.addAttribute("username", user.getUsername());
        model.addAttribute("password", user.getPassword());
        model.addAttribute("address", user.getAddress());
        model.addAttribute("receivePaper", user.isReceivePaper()); return
        "users";
    }
}

```

Here, for the first service method user(), we have passed a blank User object in the ModelAndView object with name "command", because the spring framework expects an object with name "command", if you are using <form:form> tags in your JSP file. So, when the user() method is called it returns the user.jsp view.

The second service method addUser() will be called against a POST method on the HelloWeb/addUser URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

    <h2>User Information</h2>
    <form:form method="POST" action="/HelloWeb/addUser">
        <table>
            <tr>
                <td><form:label path="username">User Name</form:label></td>
                <td><form:input path="username" /></td>
            </tr>
            <tr>
                <td><form:label path="password">Age</form:label></td>
                <td><form:password path="password" /></td>
            </tr>
            <tr>
                <td><form:label path="address">Address</form:label></td>
                <td><form:textarea path="address" rows="5" cols="30" /></td>
            </tr>
        <tr>
            <td><form:label path="receivePaper">Subscribe Newsletter</form:label></td>
            <td><form:checkbox path="receivePaper" /></td>
        </tr> <tr>
            <td colspan="2">
                <input type="submit" value="Submit"/> </td>
            </tr>
        </table>
    </form:form>
</body> </html>

```

Here, we are using the **<form:checkbox />** tag to render an HTML checkbox box.

For example –

```
<form:checkbox path="receivePaper" />
```

It will render the following HTML content.

```
<input id="receivePaper1" name="receivePaper" type="checkbox" value="true"/> <input  
type="hidden" name="_receivePaper" value="on"/>
```

users.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>  
<html>  
<head>  
    <title>Spring MVC Form Handling</title>  
</head>  
<body>  
  
<h2>Submitted User Information</h2>  
<table>  
    <tr>  
        <td>Username</td>  
        <td>${username}</td>  
    </tr>  
    <tr>  
        <td>Password</td>  
        <td>${password}</td>
```

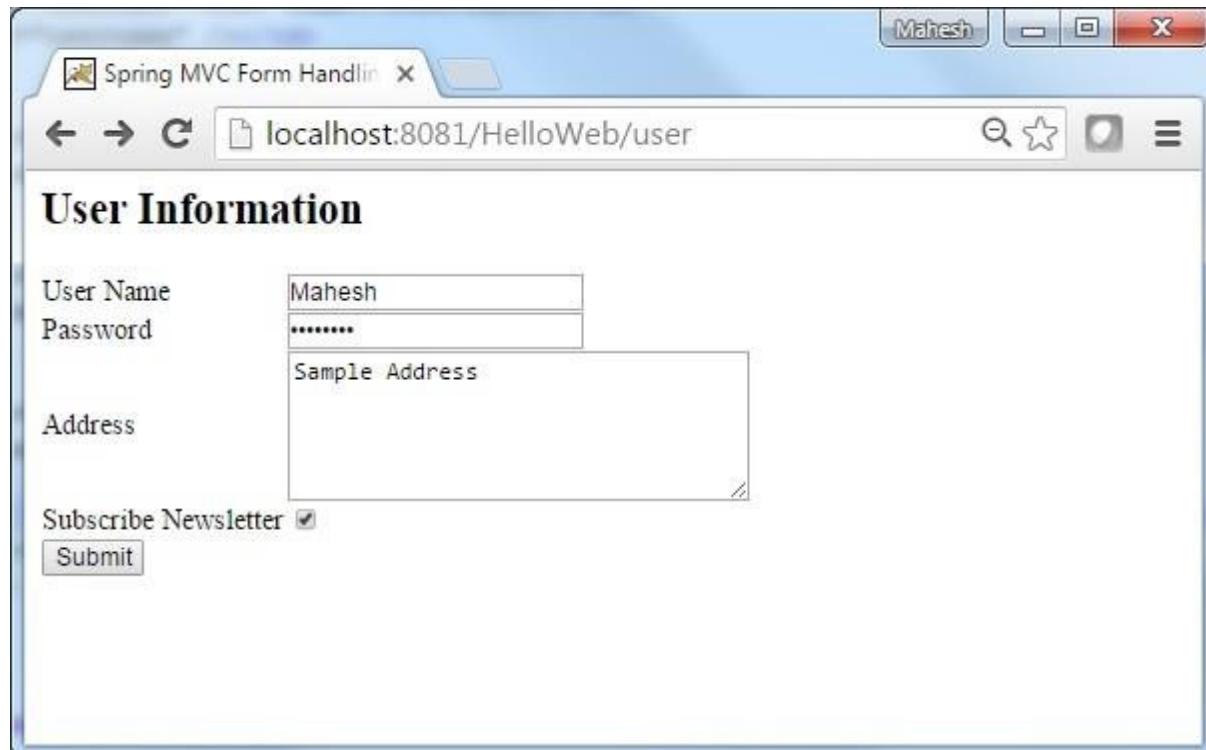
```

</tr> <tr>
    <td>Address</td>
    <td>${address}</td>
</tr> <tr>
    <td>Subscribed to Newsletter</td>
    <td>${receivePaper}</td>
</tr>
</table>
</body>
</html>

```

Once you are done with creating source and configuration files, export your application. Right click on your application, use Export → WAR File option and save your HelloWeb.war file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL – <http://localhost:8080>HelloWeb/user> and we will see the following screen if everything is fine with the Spring Web Application.



After submitting the required information, click on the submit button to submit the form. We will see the following screen if everything is fine with the Spring Web Application:



11. SpringMVC–Checkboxes

The following example explains how to use Multiple Checkboxes in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai;

public class User {

    private String username;
    private String password;
    private String address; private
    boolean receivePaper;
    private String [] favoriteFrameworks;

    public String getUsername() { return
        username;
    }

    public void setUsername(String username) {
        this.username = username;
```

```

    }

    public String getPassword() { return
        password;
    }

    public void setPassword(String password) { this.password
        = password;
    }

    public String getAddress() { return
        address;
    }

    public void setAddress(String address) { this.address
        = address;
    }

    public boolean isReceivePaper() { return
        receivePaper;
    }

    public void setReceivePaper(boolean receivePaper) {
        this.receivePaper = receivePaper;
    }

    public String[] getFavoriteFrameworks() {
        return favoriteFrameworks;
    }

    public void setFavoriteFrameworks(String[] favoriteFrameworks)
    { this.favoriteFrameworks = favoriteFrameworks; }

}

```

UserController.java

```

package com.ai;

import java.util.ArrayList; import
java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod; import
org.springframework.web.servlet.ModelAndView;

```

```

import org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET) public
    ModelAndView user() {
        User user = new User(); user.setFavoriteFrameworks((new String
        []){"Spring MVC","Struts 2"})); ModelAndView modelAndView = new
        ModelAndView("user", "command", user); return modelAndView;
    }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("SpringWeb")User user,
    ModelMap model) {
        model.addAttribute("username", user.getUsername());
        model.addAttribute("password", user.getPassword());
        model.addAttribute("address", user.getAddress());
        model.addAttribute("receivePaper", user.isReceivePaper());
        model.addAttribute("favoriteFrameworks", user.getFavoriteFrameworks());
        return "users";
    }

    @ModelAttribute("webFrameworkList") public
    List<String> getWebFrameworkList()
    {
        List<String> webFrameworkList = new
        ArrayList<String>(); webFrameworkList.add("Spring
        MVC"); webFrameworkList.add("Struts 1");
        webFrameworkList.add("Struts 2");
        webFrameworkList.add("Apache Wicket"); return
        webFrameworkList;
    }
}

```

Here, for the first service method **user()**, we have passed a blank **User** object in the **ModelAndView** object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when the **user()** method is called, it returns the **user.jsp** view.

The second service method **addUser()** will be called against a POST method on the **HelloWeb/addUser** URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>User Information</h2>
<form:form method="POST" action="/HelloWeb/addUser"> <table>
    <tr>
        <td><form:label path="username">User Name</form:label></td>
        <td><form:input path="username" /></td>
    </tr> <tr>
        <td><form:label path="password">Age</form:label></td>
        <td><form:password path="password" /></td>
    </tr>
    <tr>
        <td><form:label path="address">Address</form:label></td>
        <td><form:textarea path="address" rows="5" cols="30" /></td>
    </tr>
    <tr>
        <td><form:label path="receivePaper">Subscribe Newsletter</form:label></td>
        <td><form:checkbox path="receivePaper" /></td>
    </tr> <tr>
        <td><form:label path="favoriteFrameworks">Favorite Web
Frameworks</form:label></td>
        <td><form:checkboxes items="${webFrameworkList}" path="favoriteFrameworks"
/></td>
    </tr> <tr>
        <td colspan="2">
            <input type="submit" value="Submit"/>
        </td>
    </tr>
</table>
</form:form>

```

```

        </td> </tr>
    </table>
</form:form>
</body> </html>

```

Here, we are using **<form:checkboxes />** tag to render HTML checkboxes.

For example –

```
<form:checkboxes items="${webFrameworkList}" path="favoriteFrameworks" />
```

It will render the following HTML content.

```

<span>
    <input id="favoriteFrameworks1" name="favoriteFrameworks" type="checkbox"
value="Spring MVC" checked="checked"/>
    <label for="favoriteFrameworks1">Spring MVC</label>
</span>
<span>
    <input id="favoriteFrameworks2" name="favoriteFrameworks" type="checkbox"
value="Struts 1"/>
    <label for="favoriteFrameworks2">Struts 1</label>
</span>
<span>
    <input id="favoriteFrameworks3" name="favoriteFrameworks" type="checkbox"
value="Struts 2" checked="checked"/>
    <label for="favoriteFrameworks3">Struts 2</label>
</span>
<span>
    <input id="favoriteFrameworks4" name="favoriteFrameworks" type="checkbox"
value="Apache Wicket"/>
    <label for="favoriteFrameworks4">Apache Wicket</label>
</span>
<input type="hidden" name="_favoriteFrameworks" value="on"/>

```

users.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
```

```

</head>
<body>

<h2>Submitted User Information</h2> <table>
    <tr>
        <td>Username</td>
        <td>${username}</td>
    </tr> <tr>
        <td>Password</td>
        <td>${password}</td>
    </tr> <tr>
        <td>Address</td>
        <td>${address}</td>
    </tr> <tr>
        <td>Subscribed to Newsletter</td>
        <td>${receivePaper}</td>
    </tr> <tr>
        <td>Favorite Web Frameworks</td>
        <td> <% String[] favoriteFrameworks =
(String[])request.getAttribute("favoriteFrameworks");
for(String framework: favoriteFrameworks) {
    out.println(framework);
}
%></td>
    </tr>
</table>
</body> </html>

```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save your **HelloWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL **http://localhost:8080/HelloWeb/user** and we will see the following screen if everything is fine with the Spring Web Application.

User Name

Password

Address

Subscribe Newsletter

Favorite Web Frameworks Spring MVC Struts 1 Struts 2 Apache Wicket

After submitting the required information, click on the submit button to submit the form. We will see the following screen, if everything is fine with your Spring Web Application.

Username	Mahesh
Password	Password
Address	Sample Address
Subscribed to Newsletter	false
Favorite Web Frameworks	Spring MVC Struts 1 Struts 2

12. SpringMVC–RadioButton

The following example show how to use RadioButton in forms using the Spring Web MVC framework. To start with it, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using Spring Web Framework:

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai;

public class User {

    private String username; private
    String password; private String
    address; private boolean
    receivePaper; private String []
    favoriteFrameworks; private String
    gender;

    public String getUsername() { return
        username;
    }
    public void setUsername(String username) {
```

```
    this.username = username;
}

public String getPassword() { return
    password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getAddress() { return
    address;
}

public void setAddress(String address) { this.address
    = address;
}

public boolean isReceivePaper() { return
    receivePaper;
}

public void setReceivePaper(boolean receivePaper) {
    this.receivePaper = receivePaper;
}

public String[] getFavoriteFrameworks() {
    return favoriteFrameworks;
}

public void setFavoriteFrameworks(String[] favoriteFrameworks) {
    this.favoriteFrameworks = favoriteFrameworks;
}

public String getGender() { return
    gender;
}

public void setGender(String gender) { this.gender
    = gender;
}

}
```

UserController.java

```

package com.ai;

import java.util.ArrayList; import
java.util.List;

import org.springframework.stereotype.Controller;
import
org.springframework.web.bind.annotation.ModelAttribute;
import
org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView; import
org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET) public
    ModelAndView user() {
        User user = new User(); user.setFavoriteFrameworks((new String
        []{"Spring MVC","Struts 2"})); user.setGender("M");
        ModelAndView modelAndView = new ModelAndView("user", "command", user);
        return modelAndView;
    }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("SpringWeb")User user,
    ModelMap model) {
        model.addAttribute("username", user.getUsername());
        model.addAttribute("password", user.getPassword());
        model.addAttribute("address", user.getAddress());
        model.addAttribute("receivePaper", user.isReceivePaper());
        model.addAttribute("favoriteFrameworks", user.getFavoriteFrameworks());
        model.addAttribute("gender", user.getGender()); return
        "users";
    }

    @ModelAttribute("webFrameworkList") public
    List<String> getWebFrameworkList()
}

```

```
{
    List<String> webFrameworkList = new
    ArrayList<String>(); webFrameworkList.add("Spring
    MVC"); webFrameworkList.add("Struts 1");
    webFrameworkList.add("Struts 2");
    webFrameworkList.add("Apache Wicket"); return
    webFrameworkList;
}
}
```

Here, the first service method **user()**, we have passed a blank **User** object in the **ModelAndView** object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when the **user()** method is called, it returns the **user.jsp** view.

The second service method **addUser()** will be called against a POST method on the **HelloWeb/addUser** URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

    <h2>User Information</h2>
    <form:form method="POST" action="/HelloWeb/addUser"> <table>
        <tr>
            <td><form:label path="username">User Name</form:label></td>
            <td><form:input path="username" /></td>
        </tr> <tr>
            <td><form:label path="password">Age</form:label></td>
            <td><form:password path="password" /></td>
        </tr>
        <tr>
            <td><form:label path="address">Address</form:label></td>
            <td><form:textarea path="address" rows="5" cols="30" /></td>
        </tr>
    </table>
</form:form>

```

```

</tr>
<tr>
    <td><form:label path="receivePaper">Subscribe Newsletter</form:label></td>
    <td><form:checkbox path="receivePaper" /></td>
</tr> <tr>
    <td><form:label path="favoriteFrameworks">Favorite Web
Frameworks</form:label></td>
    <td><form:checkboxes items="${webFrameworkList}" path="favoriteFrameworks"
/></td>
</tr> <tr>
    <td><form:label path="gender">Gender</form:label></td> <td>
        <form:radioButton path="gender" value="M" label="Male" />
        <form:radioButton path="gender" value="F" label="Female" />
    </td>
</tr> <tr>
    <td colspan="2">
        <input type="submit" value="Submit"/> </td>
    </tr>
</table>
</form:form>
</body>
</html>

```

Here, we are using **<form:radioButton />** tag to render HTML radiobutton.

For example –

```

<form:radioButton path="gender" value="M" label="Male" />
<form:radioButton path="gender" value="F" label="Female" />

```

It will render following HTML content.

```

<input id="gender1" name="gender" type="radio" value="M" checked="checked"/><label
for="gender1">Male</label>
<input id="gender2" name="gender" type="radio" value="F"/><label
for="gender2">Female</label>

```

users.jsp

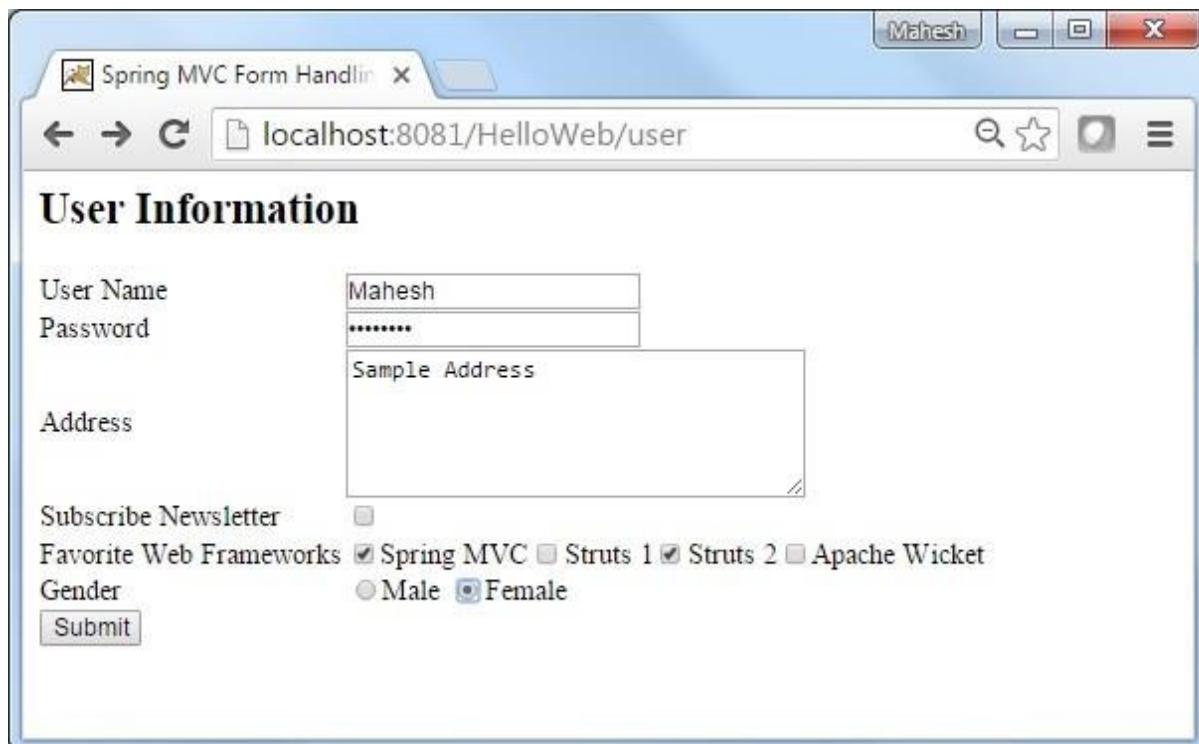
```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted User Information</h2> <table>
    <tr>
        <td>Username</td>
        <td>${username}</td>
    </tr> <tr>
        <td>Password</td>
        <td>${password}</td>
    </tr> <tr>
        <td>Address</td>
        <td>${address}</td>
    </tr> <tr>
        <td>Subscribed to Newsletter</td>
        <td>${receivePaper}</td>
    </tr> <tr>
        <td>Favorite Web Frameworks</td>
        <td> <% String[] favoriteFrameworks =
(String[])request.getAttribute("favoriteFrameworks");
        for(String framework: favoriteFrameworks) {
            out.println(framework);
        }
%></td>
    </tr> <tr>
        <td>Gender</td>
        <td>${(gender=="M"? "Male" : "Female")}</td>
    </tr>
```

```
</table>
</body>
</html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **HelloWeb.war** file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL -**http://localhost:8080/HelloWeb/user** and we will see the following screen, if everything is fine with your Spring Web Application.



After submitting the required information, click on the submit button to submit the form. We will see the following screen, if everything is fine with the Spring Web Application.



13. SpringMVC–RadioButtons

The following example explains how to use RadioButtons in forms using the Spring Web MVC framework. To begin with, let us have a working Eclipse IDE in place and follow the subsequent steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai; public class User {  
    private String username; private String  
    password; private String address;  
    private boolean receivePaper; private  
    String [] favoriteFrameworks; private  
    String gender;  
    private String favoriteNumber;  
  
    public String getUsername() { return  
        username;  
    }  
    public void setUsername(String username) {  
        this.username = username;  
    }
```

```
}

public String getPassword() { return
    password;
}

public void setPassword(String password) {
    this.password = password;
}

public String getAddress() { return
    address;
}

public void setAddress(String address) { this.address
    = address;
}

public boolean isReceivePaper() { return
    receivePaper;
}

public void setReceivePaper(boolean receivePaper) {
    this.receivePaper = receivePaper;
}

public String[] getFavoriteFrameworks() {
    return favoriteFrameworks;
}

public void setFavoriteFrameworks(String[] favoriteFrameworks) {
    this.favoriteFrameworks = favoriteFrameworks;
}

public String getGender() { return
    gender;
}

public void setGender(String gender) { this.gender
    = gender;
}

public String getFavoriteNumber() { return
    favoriteNumber;
}

public void setFavoriteNumber(String favoriteNumber)
{ this.favoriteNumber = favoriteNumber; }

}
```

UserController.java

```

package com.ai;

import java.util.ArrayList; import
java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView; import
org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET) public
    ModelAndView user() {
        User user = new User(); user.setFavoriteFrameworks((new String
        []{"Spring MVC","Struts 2"})); user.setGender("M");
        ModelAndView modelAndView = new ModelAndView("user", "command", user);
        return modelAndView;
    }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("SpringWeb")User user,
    ModelMap model) {
        model.addAttribute("username", user.getUsername());
        model.addAttribute("password", user.getPassword());
        model.addAttribute("address", user.getAddress());
        model.addAttribute("receivePaper", user.isReceivePaper());
        model.addAttribute("favoriteFrameworks", user.getFavoriteFrameworks());
        model.addAttribute("gender", user.getGender());
        model.addAttribute("favoriteNumber", user.getFavoriteNumber());
        return "users";
    }

    @ModelAttribute("webFrameworkList")
}

```

```

public List<String> getWebFrameworkList()
{
    List<String> webFrameworkList = new ArrayList<String>();
    webFrameworkList.add("Spring MVC");
    webFrameworkList.add("Struts 1");
    webFrameworkList.add("Struts 2");
    webFrameworkList.add("Apache Wicket"); return
    webFrameworkList;
}

@ModelAttribute("numbersList") public
List<String> getNumbersList()
{
    List<String> numbersList = new
    ArrayList<String>(); numbersList.add("1");
    numbersList.add("2"); numbersList.add("3");
    numbersList.add("4"); return numbersList;
}
}

```

Here, for the first service method **user()**, we have passed a blank **User** object in the **ModelAndView** object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when **user()** method is called, it returns the **user.jsp** view.

The second service method **addUser()** will be called against a POST method on the **HelloWeb/addUser** URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

    <h2>User Information</h2>
    <form:form method="POST" action="/HelloWeb/addUser">

```

```

<table>
    <tr>
        <td><form:label path="username">User Name</form:label></td>
        <td><form:input path="username" /></td>
    </tr> <tr>
        <td><form:label path="password">Age</form:label></td>
        <td><form:password path="password" /></td>
    </tr>
    <tr>
        <td><form:label path="address">Address</form:label></td>
        <td><form:textarea path="address" rows="5" cols="30" /></td>
    </tr>
    <tr>
        <td><form:label path="receivePaper">Subscribe Newsletter</form:label></td>
        <td><form:checkbox path="receivePaper" /></td>
    </tr> <tr>
        <td><form:label path="favoriteFrameworks">Favorite Web
Frameworks</form:label></td>
        <td><form:checkboxes items="${webFrameworkList}" path="favoriteFrameworks"
/></td>
    </tr>
    <tr>

        <td><form:label path="gender">Gender</form:label></td> <td>
            <form:radiobutton path="gender" value="M" label="Male" />
            <form:radiobutton path="gender" value="F" label="Female" />
        </td>
    </tr>
    <tr>
        <td><form:label path="favoriteNumber">Favorite Number</form:label></td>
        <td>
            <form:radiobuttons path="favoriteNumber" items="${numbersList}" />
        </td>
    </tr> <tr>
        <td colspan="2">
            <input type="submit" value="Submit"/>
        </td>
    </tr>

```

```
</td> </tr>
</table>
</form:form>
</body> </html>
```

Here, we are using **<form:radioButtons />** tag to render the HTML radio buttons. For example –

```
<form:radioButtons path="favoriteNumber" items="${numbersList}" />
```

It will render the following HTML content.

```
<span>
<input id="favoriteNumber1" name="favoriteNumber" type="radio" value="1"/>
<label for="favoriteNumber1">1</label>
</span>
<span>
<input id="favoriteNumber2" name="favoriteNumber" type="radio" value="2"/>
<label for="favoriteNumber2">2</label>
</span>
<span>
<input id="favoriteNumber3" name="favoriteNumber" type="radio" value="3"/>
<label for="favoriteNumber3">3</label>
</span>
<span>
<input id="favoriteNumber4" name="favoriteNumber" type="radio" value="4"/>
<label for="favoriteNumber4">4</label>
</span>
```

users.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted User Information</h2>
<table>
```

```

<tr>
    <td>Username</td>
    <td>${username}</td>
</tr> <tr>
    <td>Password</td>
    <td>${password}</td>
</tr> <tr>
    <td>Address</td>
    <td>${address}</td>
</tr> <tr>
    <td>Subscribed to Newsletter</td>
    <td>${receivePaper}</td>
</tr> <tr>
    <td>Favorite Web Frameworks</td>

    <td> <% String[] favoriteFrameworks =
(String[])request.getAttribute("favoriteFrameworks");
        for(String framework: favoriteFrameworks) {
            out.println(framework);
        }
    %></td>
</tr> <tr>
    <td>Gender</td>
    <td>${(gender=="M"? "Male" : "Female")}</td>
</tr> <tr>
    <td>Favourite Number</td>
    <td>${favoriteNumber}</td>
</tr>
</table>
</body>
</html>

```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the HelloWeb.war file in Tomcat's webapps folder.

Spring MVC

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try the following URL - **http://localhost:8080/HelloWeb/user** and we will see the following screen, if everything is fine with the Spring Web Application.

User Name
Mahesh

Password
.....

Address
Sample Address

Subscribe Newsletter

Favorite Web Frameworks Spring MVC Struts 1 Struts 2 Apache Wicket

Gender Male Female

Favorite Number 1 2 3 4

Submit

After submitting the required information, click on the submit button to submit the form. We will see the following screen, if everything is fine with your Spring Web Application.

Submitted User Information

Username	Mahesh
Password	Password
Address	Sample Address
Subscribed to Newsletter	false
Favorite Web Frameworks	Spring MVC Struts 2
Gender	Male
Favourite Number	2

14. SpringMVC–Dropdown

The following example describes how to use Dropdown in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai; public class User {  
    private String username; private String  
    password; private String address;  
    private boolean receivePaper; private  
    String [] favoriteFrameworks; private  
    String gender; private String  
    favoriteNumber; private String country;  
  
    public String getUsername() { return  
        username;  
    }  
    public void setUsername(String username) {
```

```
this.username = username;
```

```
}

public String getPassword() { return
    password;
}

public void setPassword(String password) { this.password
    = password;
}

public String getAddress() { return
    address;
}

public void setAddress(String address) { this.address
    = address;
}

public boolean isReceivePaper() { return
    receivePaper;
}

public void setReceivePaper(boolean receivePaper) { this.receivePaper
    = receivePaper;
}

public String[] getFavoriteFrameworks() { return
    favoriteFrameworks;
}

public void setFavoriteFrameworks(String[] favoriteFrameworks) {
    this.favoriteFrameworks = favoriteFrameworks;
}

public String getGender() { return
    gender;
}

public void setGender(String gender) { this.gender
    = gender;
}

public String getFavoriteNumber() { return
    favoriteNumber;
}

public void setFavoriteNumber(String favoriteNumber) { this.favoriteNumber
    = favoriteNumber;
}
```

```
}

public String getCountry() {
    return country;
}

public void setCountry(String country) {
    this.country = country;
}

}
```

UserController.java

```
package com.ai;

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

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class UserController {
    @RequestMapping(value = "/user", method = RequestMethod.GET) public
    ModelAndView user() {
        User user = new User(); user.setFavoriteFrameworks((new String
        []{"Spring MVC","Struts 2"}));
        user.setGender("M");

        ModelAndView modelAndView = new ModelAndView("user", "command", user);
        return modelAndView;
    }

    @RequestMapping(value = "/addUser", method = RequestMethod.POST) public
    String addUser(@ModelAttribute("SpringWeb")User user,
        ModelMap model) {
```

```
model.addAttribute("username", user.getUsername());
```

```

model.addAttribute("password", user.getPassword());
model.addAttribute("address", user.getAddress());
model.addAttribute("receivePaper", user.isReceivePaper());
model.addAttribute("favoriteFrameworks", user.getFavoriteFrameworks());
model.addAttribute("gender", user.getGender());
model.addAttribute("favoriteNumber", user.getFavoriteNumber());
model.addAttribute("country", user.getCountry()); return
"users";
}

@ModelAttribute("webFrameworkList") public
List<String> getWebFrameworkList()
{
    List<String> webFrameworkList = new
    ArrayList<String>(); webFrameworkList.add("Spring
MVC"); webFrameworkList.add("Struts 1");
    webFrameworkList.add("Struts 2");
    webFrameworkList.add("Apache Wicket"); return
    webFrameworkList;
}

@ModelAttribute("numbersList") public
List<String> getNumbersList()
{
    List<String> numbersList = new
    ArrayList<String>(); numbersList.add("1");
    numbersList.add("2"); numbersList.add("3");
    numbersList.add("4"); return numbersList;
}

@ModelAttribute("countryList") public
Map<String, String> getCountryList()
{
    Map<String, String> countryList = new HashMap<String, String>();
    countryList.put("US", "United States"); countryList.put("CH",
    "China");
}

```

```
        countryList.put("SG", "Singapore");
        countryList.put("MY", "Malaysia");
        return countryList;
    }
}
```

Here, for the first service method **user()**, we have passed a blank **User** object in the **ModelAndView** object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So when the **user()** method is called, it returns the **user.jsp** view.

The second service method **addUser()** will be called against a POST method on the **HelloWeb/addUser** URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>User Information</h2>
<form:form method="POST" action="/HelloWeb/addUser"> <table>
    <tr>
        <td><form:label path="username">User Name</form:label></td>
        <td><form:input path="username" /></td>
    </tr> <tr>
        <td><form:label path="password">Age</form:label></td>
        <td><form:password path="password" /></td>
    </tr>
    <tr>
        <td><form:label path="address">Address</form:label></td>
        <td><form:textarea path="address" rows="5" cols="30" /></td>
    </tr>
    <tr>
        <td><form:label path="receivePaper">Subscribe Newsletter</form:label></td>
        <td><form:checkbox path="receivePaper" /></td>
    </tr>
</table>
</form:form>
```

</tr> <tr>

```

<td><form:label path="favoriteFrameworks">Favorite Web
Frameworks</form:label></td>
<td><form:checkboxes items="${webFrameworkList}" path="favoriteFrameworks"
/></td>
</tr> <tr>
<td><form:label path="gender">Gender</form:label></td> <td>
<form:radiobutton path="gender" value="M" label="Male" />
<form:radiobutton path="gender" value="F" label="Female" /> </td>
</tr> <tr>
<td><form:label path="favoriteNumber">Favorite Number</form:label></td>
<td>
<form:radiobuttons path="favoriteNumber" items="${numbersList}" />
</td>
</tr> <tr>
<td><form:label path="country">Country</form:label></td> <td>
<form:select path="country">
<form:option value="NONE" label="Select"/>
<form:options items="${countryList}" />
</form:select>
</td>
</tr> <tr>
<td colspan="2">
<input type="submit" value="Submit"/>
</td> </tr>
</table>
</form:form>
</body>
</html>

```

Here, we are using **<form:select />** , **<form:option />** and **<form:options />** tags to render HTML select. For example –

```

<form:select path="country">
<form:option value="NONE" label="Select"/>
<form:options items="${countryList}" />
</form:select>

```

It will render the following HTML content.

```
<select id="country" name="country">
    <option value="NONE">Select</option>
    <option value="US">United States</option>
    <option value="CH">China</option>
    <option value="MY">Malaysia</option>
    <option value="SG">Singapore</option>
</select>
```

users.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>

<body>
<h2>Submitted User Information</h2> <table>
    <tr>
        <td>Username</td>
        <td>${username}</td>
    </tr> <tr>
        <td>Password</td>
        <td>${password}</td>
    </tr> <tr>
        <td>Address</td>
        <td>${address}</td>
    </tr>
    <tr>
```

```

<td>Subscribed to Newsletter</td>
<td>${receivePaper}</td>
</tr> <tr>
    <td>Favorite Web Frameworks</td>
    <td> <% String[] favoriteFrameworks =
(String[])request.getAttribute("favoriteFrameworks");
        for(String framework: favoriteFrameworks) {
            out.println(framework);
        }
    %></td>
</tr> <tr>
    <td>Gender</td>
    <td>${(gender=="M"? "Male" : "Female")}</td>
</tr> <tr>
    <td>Favourite Number</td>
    <td>${favoriteNumber}</td>

</tr> <tr>
    <td>Country</td>
    <td>${country}</td>
</tr>
</table>
</body> </html>

```

Once you are done with creating source and configuration files, export your application. Right click on your application, use the **Export → WAR File** option and save your HelloWeb.war file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL –**<http://localhost:8080>HelloWeb/user>** and we will see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handling". The address bar displays "localhost:8081/HelloWeb/user". The main content area is titled "User Information". It contains the following fields:

- User Name: Mahesh
- Password: (redacted)
- Address: sample address
- Subscribe Newsletter:
- Favorite Web Frameworks:
 - Spring MVC
 - Struts 1
 - Struts 2
 - Apache Wicket
- Gender:
 - Male
 - Female
- Favorite Number:
 - 1
 - 2
 - 3
 - 4
- Country: United States
- Submit button

After submitting the required information, click on the submit button to submit the form. You should see the following screen, if everything is fine with your Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handling". The address bar displays "localhost:8081/HelloWeb/addUser". The main content area is titled "Submitted User Information" and displays the following table of submitted data:

Username	Mahesh
Password	Password
Address	sample address
Subscribed to Newsletter	true
Favorite Web Frameworks	Spring MVC Struts 2
Gender	Male
Favourite Number	2
Country	US

15. SpringMVC–Listbox

The following example shows how to use Listbox in forms using the Spring Web MVC framework. To begin with, let us have a working Eclipse IDE in place and follow the subsequent steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User, UserController under the com.aipackage.
3	Create view files user.jsp, users.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai;

public class User { private String
    username; private String password;
    private String address; private
    boolean receivePaper; private String
    [] favoriteFrameworks; private
    String gender; private String
    favoriteNumber; private String
    country;

    private String [] skills;
    public String getUsername() {
```

```
        return username;
    }

    public void setUsername(String username) { this.username
        = username;
    }

    public String getPassword() { return
        password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) { this.address
        = address;
    }

    public boolean isReceivePaper() {
        return receivePaper;
    }

    public void setReceivePaper(boolean receivePaper) {
        this.receivePaper = receivePaper;
    }

    public String[] getFavoriteFrameworks() {
        return favoriteFrameworks;
    }

    public void setFavoriteFrameworks(String[] favoriteFrameworks) {
        this.favoriteFrameworks = favoriteFrameworks;
    }

    public String getGender() { return
        gender;
    }

    public void setGender(String gender) { this.gender
        = gender;
    }

    public String getFavoriteNumber() { return
        favoriteNumber;
    }
}
```

```
public void setFavoriteNumber(String favoriteNumber) {  
    this.favoriteNumber = favoriteNumber;  
}  
public String getCountry() {  
    return country;  
}  
public void setCountry(String country) {  
    this.country = country;  
}  
public String[] getSkills() { return  
    skills;  
}  
public void setSkills(String[] skills) { this.skills  
    = skills;  
}  
}
```

UserController.java

```
package com.ai;

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

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import org.springframework.ui.ModelMap;

@Controller
public class UserController {

    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public ModelAndView user() {
        User user = new User();
```

```
user.setFavoriteFrameworks((new String []{"Spring MVC","Struts 2"}));
    user.setGender("M");
ModelAndView modelAndView = new ModelAndView("user", "command", user); return modelAndView;
}

@RequestMapping(value = "/addUser", method = RequestMethod.POST) public String
addUser(@ModelAttribute("SpringWeb")User user, ModelMap model) {
    model.addAttribute("username", user.getUsername()); model.addAttribute("password",
    user.getPassword()); model.addAttribute("address", user.getAddress());
    model.addAttribute("receivePaper", user.isReceivePaper());
model.addAttribute("favoriteFrameworks", user.getFavoriteFrameworks());
    model.addAttribute("gender", user.getGender());
    model.addAttribute("favoriteNumber", user.getFavoriteNumber());
    model.addAttribute("country", user.getCountry()); model.addAttribute("skills",
    user.getSkills()); return "users";
}

@ModelAttribute("webFrameworkList") public List<String>
getWebFrameworkList()
{
    List<String> webFrameworkList = new ArrayList<String>();
    webFrameworkList.add("Spring MVC"); webFrameworkList.add("Struts 1");
    webFrameworkList.add("Struts 2"); webFrameworkList.add("Apache Wicket");

    return webFrameworkList;
}

@ModelAttribute("numbersList") public List<String>
getNumbersList()
{
    List<String> numbersList = new ArrayList<String>();
    numbersList.add("1"); numbersList.add("2");
}
```

```

        numbersList.add("3");
        numbersList.add("4");
        return numbersList;
    }

    @ModelAttribute("countryList")
    public Map<String, String> getCountryList()
    {
        Map<String, String> countryList = new HashMap<String,
        String>(); countryList.put("US", "United States");
        countryList.put("CH", "China"); countryList.put("SG",
        "Singapore"); countryList.put("MY", "Malaysia"); return
        countryList;
    }

    @ModelAttribute("skillsList")
    public Map<String, String> getSkillsList()
    {
        Map<String, String> skillList = new HashMap<String, String>();
        skillList.put("Hibernate", "Hibernate");
        skillList.put("Spring", "Spring");
        skillList.put("Apache Wicket", "Apache
        Wicket"); skillList.put("Struts", "Struts");
        return skillList;
    }
}

```

Here, for the first service method **user()**, we have passed a blank **User** object in the **ModelAndView** object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when the **user()** method is called, it returns the **user.jsp** view.

The second service method **addUser()** will be called against a POST method on the **HelloWeb/addUser** URL. You will prepare your model object based on the submitted information. Finally, the "users" view will be returned from the service method, which will result in rendering the users.jsp.

user.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
```

```
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>User Information</h2>
<form:form method="POST" action="/HelloWeb/addUser"> <table>
    <tr>
        <td><form:label path="username">User Name</form:label></td>
        <td><form:input path="username" /></td>
    </tr> <tr>
        <td><form:label path="password">Age</form:label></td>
        <td><form:password path="password" /></td>
    </tr>
    <tr>
        <td><form:label path="address">Address</form:label></td>
        <td><form:textarea path="address" rows="5" cols="30" /></td>
    </tr>
    <tr>
        <td><form:label path="receivePaper">Subscribe Newsletter</form:label></td>
        <td><form:checkbox path="receivePaper" /></td>
    </tr>
    <tr>

        <td><form:label path="favoriteFrameworks">Favorite Web
Frameworks</form:label></td>
        <td><form:checkboxes items="${webFrameworkList}" path="favoriteFrameworks"
/></td>
    </tr> <tr>
        <td><form:label path="gender">Gender</form:label></td> <td>
            <form:radio button path="gender" value="M" label="Male" />
            <form:radio button path="gender" value="F" label="Female" />
        </td>
    </tr>
    <tr>
```

```

<td><form:label path="favoriteNumber">Favorite Number</form:label></td>
<td>
    <form:radioButtons path="favoriteNumber" items="${numbersList}" />
</td>
</tr>
<tr>
    <td><form:label path="country">Country</form:label></td>
    <td>
        <form:select path="country">
            <form:option value="NONE" label="Select"/>
            <form:options items="${countryList}" />
        </form:select>
    </td>
</tr>
<tr>
    <td><form:label path="skills">Skills</form:label></td>
    <td>
        <form:select path="skills" items="${skillsList}"
                     multiple="true" />
    </td>
</tr>
<tr>
    <td colspan="2">
        <input type="submit"
               value="Submit"/> </td> </tr>
</table>
</form:form>
</body>
</html>

```

Here, we are using a **<form:select />** tag , with the attribute **multiple=true** to render an HTML listbox. For example –

<form:select path="skills" items="\${skillsList}" multiple="true" />
--

It will render the following HTML content.

```
<select id="skills" name="skills" multiple="multiple">
```

```
    <option value="Struts">Struts</option>
    <option value="Hibernate">Hibernate</option>
    <option value="Apache Wicket">Apache Wicket</option>
    <option value="Spring">Spring</option>
</select>
<input type="hidden" name="_skills" value="1"/>
```

users.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted User Information</h2> <table>
<tr>
    <td>Username</td>
    <td>${username}</td>
</tr> <tr>
    <td>Password</td>
    <td>${password}</td>
</tr> <tr>
    <td>Address</td>

    <td>${address}</td>
</tr> <tr>
    <td>Subscribed to Newsletter</td>
    <td>${receivePaper}</td>
</tr> <tr>
    <td>Favorite Web Frameworks</td>
    <td> <% String[] favoriteFrameworks =
(String[])request.getAttribute("favoriteFrameworks");
        for(String framework: favoriteFrameworks) {
```

```

        out.println(framework);
    }
%></td>
</tr> <tr>
<td>Gender</td>
<td>${(gender=="M"? "Male" : "Female")}</td>
</tr> <tr>
<td>Favourite Number</td>
<td>${favoriteNumber}</td>
</tr> <tr>
<td>Country</td>
<td>${country}</td>
</tr>
<tr>
<td>Skills</td>
<td> <% String[] skills =
        (String[])request.getAttribute("skills"); for(String skill:
skills) { out.println(skill);
}
%></td>
</tr>
</table>
</body>
</html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the HelloWeb.war file in Tomcat's webapps folder.

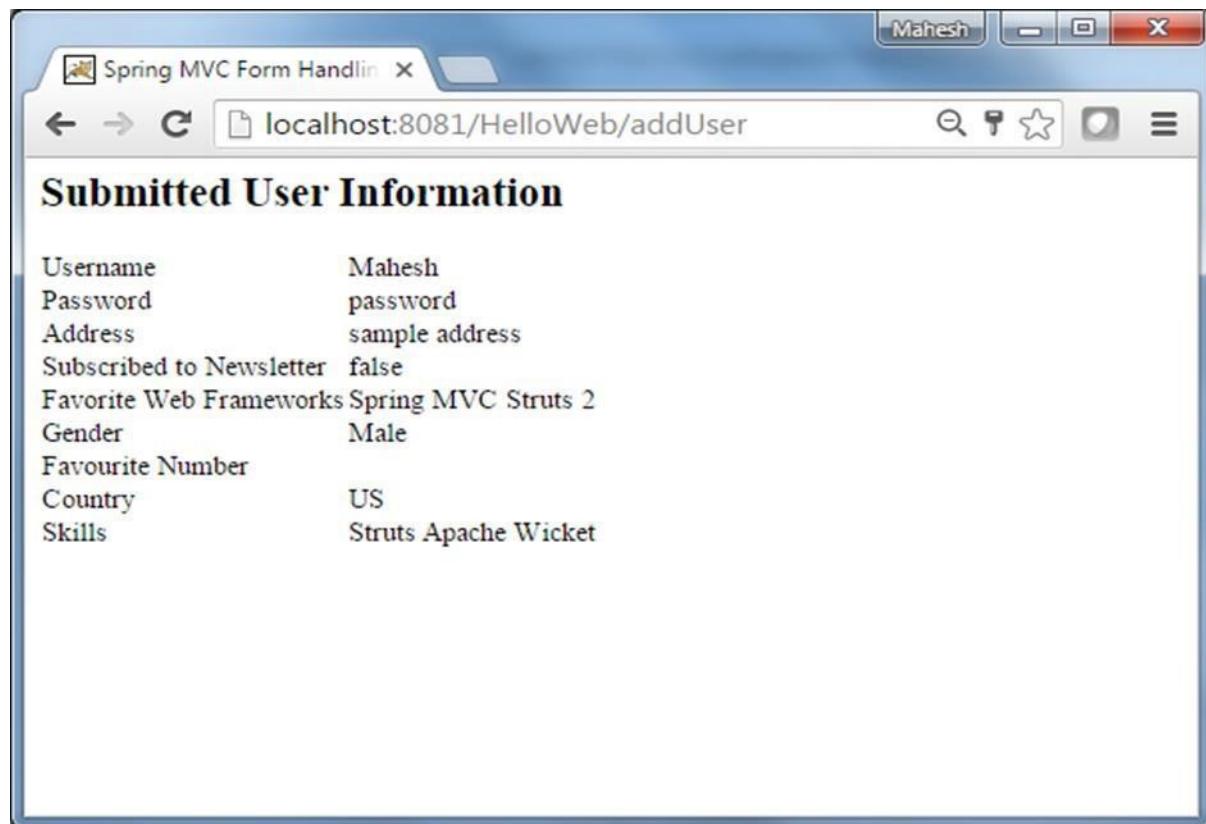
Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL – **<http://localhost:8080>HelloWeb/user>** and we will see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handler". The address bar displays "localhost:8081/HelloWeb/user". The main content area is titled "User Information". The form fields include:

- User Name: Mahesh
- Password: (redacted)
- Address: sample address
- Subscribe Newsletter: (unchecked)
- Favorite Web Frameworks:
 - Spring MVC (checked)
 - Struts 1 (unchecked)
 - Struts 2 (checked)
 - Apache Wicket (unchecked)
- Gender:
 - Male (radio button checked)
 - Female (radio button)
- Favorite Number:
 - 1 (radio button)
 - 2 (radio button)
 - 3 (radio button)
 - 4 (radio button)
- Country: United States
- Skills:
 - Struts
 - Hibernate
 - Apache Wicket (selected)
 - Spring

A "Submit" button is located at the bottom left of the form.

After submitting the required information, click on the submit button to submit the form. You should see the following screen, if everything is fine with your Spring Web Application.



16. SpringMVC– HiddenField

The following example describes how to use a Hidden Field in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and consider the following steps to develop a Dynamic Form based Web Application using Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes Student, StudentController under the com.ai package.
3	Create view files student.jsp, result.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

Student.java

```
package com.ai;

public class Student { private
    Integer age; private String
    name; private Integer id;

    public void setAge(Integer age) { this.age
        = age;
    }
    public Integer getAge() { return
        age;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }
    public String getName() { return
        name;
    }
    public void setId(Integer id) { this.id
        = id;
    }
    public Integer getId() { return
        id;
    }
}
```

StudentController.java

```

package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.ModelMap;

@Controller
public class StudentController {
    @RequestMapping(value = "/student", method = RequestMethod.GET)
    public ModelAndView student() { return new
        ModelAndView("student", "command", new Student());
    }

    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute("SpringWeb")Student student, ModelMap
model) {
        model.addAttribute("name", student.getName());
        model.addAttribute("age", student.getAge());
        model.addAttribute("id", student.getId());
        return "result";
    }
}

```

Here, for the first service method **student()**, we have passed a blank **Studentobject** in the ModelAndView object with the name "command", because the spring framework expects an object with the name "command", if you are using <form:form> tags in your JSP file. So, when the **student()** method is called, it returns the **student.jsp** view.

The second service method **addStudent()** will be called against a POST method on the **HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally, a "result" view will be returned from the service method, which will result in rendering result.jsp

student.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

    <h2>Student Information</h2>
    <form:form method="POST" action="/HelloWeb/addStudent">
        <table> <tr>
            <td><form:label path="name">Name</form:label></td>
            <td><form:input path="name" /></td>
        </tr> <tr>
            <td><form:label path="age">Age</form:label></td>
            <td><form:input path="age" /></td>
        </tr> <tr>
            <td></td>
            <td><form:hidden path="id" value="1" /></td>
        </tr> <tr>
            <td colspan="2">
                <input type="submit" value="Submit"/> </td>
            </tr>
        </table>
    </form:form>
</body>
</html>

```

Here, we are using the **<form:hidden />** tag to render a HTML hidden field.

For example –

```
<form:hidden path="id" value="1"/>
```

It will render the following HTML content.

```
<input id="id" name="id" type="hidden" value="1"/>
```

result.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted Student Information</h2>
<table> <tr>
    <td>Name</td>
    <td>${name}</td>
</tr> <tr>
    <td>Age</td>
    <td>${age}</td>
</tr> <tr>
    <td>ID</td>
    <td>${id}</td>
</tr>
</table>
</body>
</html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application and use **Export > WAR File** option and save your **HelloWeb.war** file in Tomcat's *webapps* folder.

Now start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL – **<http://localhost:8080/HelloWeb/student>** and we will see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handlin". The address bar displays "localhost:8081/HelloWeb/student". The page content is titled "Student Information". It contains two input fields: "Name" with value "Mahesh" and "Age" with value "30". A "Submit" button is present below the inputs.

After submitting the required information, click on the submit button to submit the form. We will see the following screen, if everything is fine with your Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handlin". The address bar displays "localhost:8081/HelloWeb/addStudent". The page content is titled "Submitted Student Information". It displays the submitted data: "Name Mahesh", "Age 30", and "ID 1".

17. SpringMVC – ErrorHandling

The following example shows how to use Error Handling and Validators in forms using the Spring Web MVC Framework. To start with, let us have a working Eclipse IDE in place and consider the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes Student, StudentController and StudentValidator under the com.ai package.
3	Create view files addStudent.jsp, result.jsp under the jsp sub-folder.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

Student.java

```
package com.ai;

public class Student { private
    Integer age; private String
    name; private Integer id;

    public void setAge(Integer age) { this.age
        = age;
    }
    public Integer getAge() { return
        age;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }
    public String getName() { return
        name;
    }

    public void setId(Integer id) { this.id
        = id;
    }
    public Integer getId() { return
        id;
    }
}
```

StudentValidator.java

```
package com.ai;

import org.springframework.validation.Errors; import
org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class StudentValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) { return
        Student.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors,
            "name", "required.name","Field name is required.");
    }
}
```

StudentController.java

```
package com.ai;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Controller; import
org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.validation.Validator;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.WebDataBinder; import
org.springframework.web.bind.annotation.InitBinder; import
org.springframework.web.bind.annotation.ModelAttribute;
import
org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class StudentController {
    @Autowired
    @Qualifier("studentValidator") private
    Validator validator;

    @InitBinder
    private void initBinder(WebDataBinder binder) {
        binder.setValidator(validator);
    }

    @RequestMapping(value = "/addStudent", method = RequestMethod.GET)
    public ModelAndView student() { return new
    ModelAndView("addStudent", "command", new Student());
    }

    @ModelAttribute("student")
    public Student createStudentModel() { return
    new Student();
    }

    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute("student") @Validated Student student,
```

```

        BindingResult bindingResult, Model model)
{   if (bindingResult.hasErrors()) { return
"addStudent";
    }
    model.addAttribute("name", student.getName());
    model.addAttribute("age", student.getAge());
    model.addAttribute("id", student.getId());

    return "result";
}
}

```

HelloWeb-servlet.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

    <bean id="studentValidator" class="com.ai.StudentValidator" /> </beans>

```

Here, for the first service method **student()**, we have passed a blank Studentobject in the ModelAndView object with name "command", because the spring framework expects an object with name "command", if you are using `<form:form>` tags in your JSP file. So, when **student()** method is called, it returns **addStudent.jsp** view.

The second service method **addStudent()** will be called against a POST method on the **HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally, a "result" view will be returned from the service method, which will result in rendering the result.jsp. In case there are errors generated using

validator then same view "addStudent" is returned, Spring automatically injects error messages from **BindingResult** in view.

addStudent.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<style>
.error { color:
    #ff0000;
}

.errorblock { color:
    #000;
background-color: #ffEEEE;
border: 3px solid #ff0000;
padding: 8px; margin:
16px;
}
</style>
<body>

<h2>Student Information</h2>
<form:form method="POST" action="/HelloWeb/addStudent" commandName="student">
    <form:errors path="*" cssClass="errorblock" element="div" />
    <table> <tr>
        <td><form:label path="name">Name</form:label></td>
        <td><form:input path="name" /></td>
        <td><form:errors path="name" cssClass="error" /></td>
    </tr> <tr>
        <td><form:label path="age">Age</form:label></td>
        <td><form:input path="age" /></td>
    </tr> <tr>
        <td><form:label path="id">id</form:label></td>
        <td><form:input path="id" /></td>
    </tr>
</table>
</form:form>
```

```
</tr> <tr>
    <td colspan="2">
        <input type="submit" value="Submit"/> </td>
    </tr>
</table>
</form:form>
</body>
</html>
```

Here, we are using **<form:errors />** tag with path="*" to render error messages. For example –

```
<form:errors path="*" cssClass="errorblock" element="div" />
```

It will render the error messages for all input validations. We are using **<form:errors />** tag with path="name" to render error message for name field. For example –

```
<form:errors path="name" cssClass="error" />
```

It will render error messages for the name field validations.

result.jsp

```

<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted Student Information</h2>
<table> <tr>
    <td>Name</td>
    <td>${name}</td>
</tr> <tr>
    <td>Age</td>
    <td>${age}</td>
</tr>
<tr>

```

```

    <td>ID</td>
    <td>${id}</td>
</tr>
</table>
</body> </html>

```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **HelloWeb.war** file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from webapps folder using a standard browser. Try a URL –**http://localhost:8080/HelloWeb/addStudent** and we will see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handlin" with the URL "localhost:8081/HelloWeb/addStudent". The page content is titled "Student Information" and contains three input fields: "Name", "Age", and "id", each with a corresponding text input box. Below these fields is a "Submit" button. The browser's address bar also displays "Mahesh".

After submitting the required information, click on the submit button to submit the form. You should see the following screen, if everything is fine with the Spring Web Application.

The screenshot shows a web browser window titled "Spring MVC Form Handling". The URL in the address bar is "localhost:8081/HelloWeb/addStudent". The page content is titled "Student Information". A red error message "Field name is required." is displayed above the form fields. The form contains three input fields: "Name" (empty), "Age" (30), and "id" (1). Below the form is a "Submit" button.

Field name is required.

Name

Age

id

Submit

18. SpringMVC– FileUpload

The following example shows how to use File Upload Control in forms using the Spring Web MVC framework. To start with, let us have a working Eclipse IDE in place and adhere to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name HelloWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes FileModel, FileUploadController under the com.ai package.
3	Create view files fileUpload.jsp, success.jsp under the jsp sub-folder.

4	Create a folder temp under the WebContent sub-folder.
5	Download Apache Commons FileUpload library commons-fileupload.jar and Apache Commons IO library commons-io.jar . Put them in your CLASSPATH.
6	The final step is to create the content of the source and configuration files and export the application as explained below.

FileModel.java

```
package com.ai;
import org.springframework.web.multipart.MultipartFile;

public class FileModel { private
    MultipartFile file; public
    MultipartFile getFile() { return
    file;
}
```

```
public void setFile(MultipartFile file) {
    this.file = file;
}
}
```

FileUploadController.java

```
package com.ai;

import java.io.File;
import java.io.IOException;

import javax.servlet.ServletContext;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller; import
org.springframework.ui.ModelMap; import
org.springframework.util.FileCopyUtils; import
org.springframework.validation.BindingResult; import
org.springframework.validation.annotation.Validated; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestMethod; import
org.springframework.web.multipart.MultipartFile;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class FileUploadController {
    @Autowired
    ServletContext context;

    @RequestMapping(value = "/fileUploadPage", method = RequestMethod.GET)
    public ModelAndView fileUploadPage() {
        FileModel file = new FileModel();
        ModelAndView modelAndView = new ModelAndView("fileUpload", "command", file);
        return modelAndView;
    }
}
```

```
@RequestMapping(value="/fileUploadPage", method = RequestMethod.POST)
public String fileUpload(@Validated FileModel file, BindingResult result,
ModelMap model) throws IOException {
```

```
if (result.hasErrors()) {  
    System.out.println("validation errors");  
    return "fileUploadPage";  
}  
else {  
    System.out.println("Fetching file");  
    MultipartFile multipartFile = file.getFile();  
    String uploadPath = context.getRealPath("") + File.separator + "temp" +  
    File.separator;  
    // Now do something with file...  
    FileCopyUtils.copy(file.getFile().getBytes(), new  
    File(uploadPath+file.getFile().getOriginalFilename()));  
    String fileName =  
        multipartFile.getOriginalFilename();  
    model.addAttribute("fileName", fileName);  
    return  
    "success";  
}  
}  
}
```

HelloWeb-servlet.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       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
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

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

Here, for the first service method **fileUploadPage()**, we have passed a blank **FileModel** object in the **ModelAndView** object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when **fileUploadPage()** method is called, it returns **fileUpload.jsp** view.

The second service method **fileUpload()** will be called against a POST method on the **HelloWeb/fileUploadPage** URL. You will prepare the file to be uploaded based on the submitted information. Finally, a "success" view will be returned from the service method, which will result in rendering success.jsp.

fileUpload.jsp

```

<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
<head>
<title>File Upload Example</title>
</head> <body>
<form:form method="POST" modelAttribute="fileUpload" enctype="multipart/form-
data">
    Please select a file to upload :
    <input type="file" name="file" />
    <input type="submit" value="upload" /> </form:form>
</body>
</html>

```

Here, we are using **modelAttribute** attribute with value="fileUpload" to map the file Upload control with the server model.

success.jsp

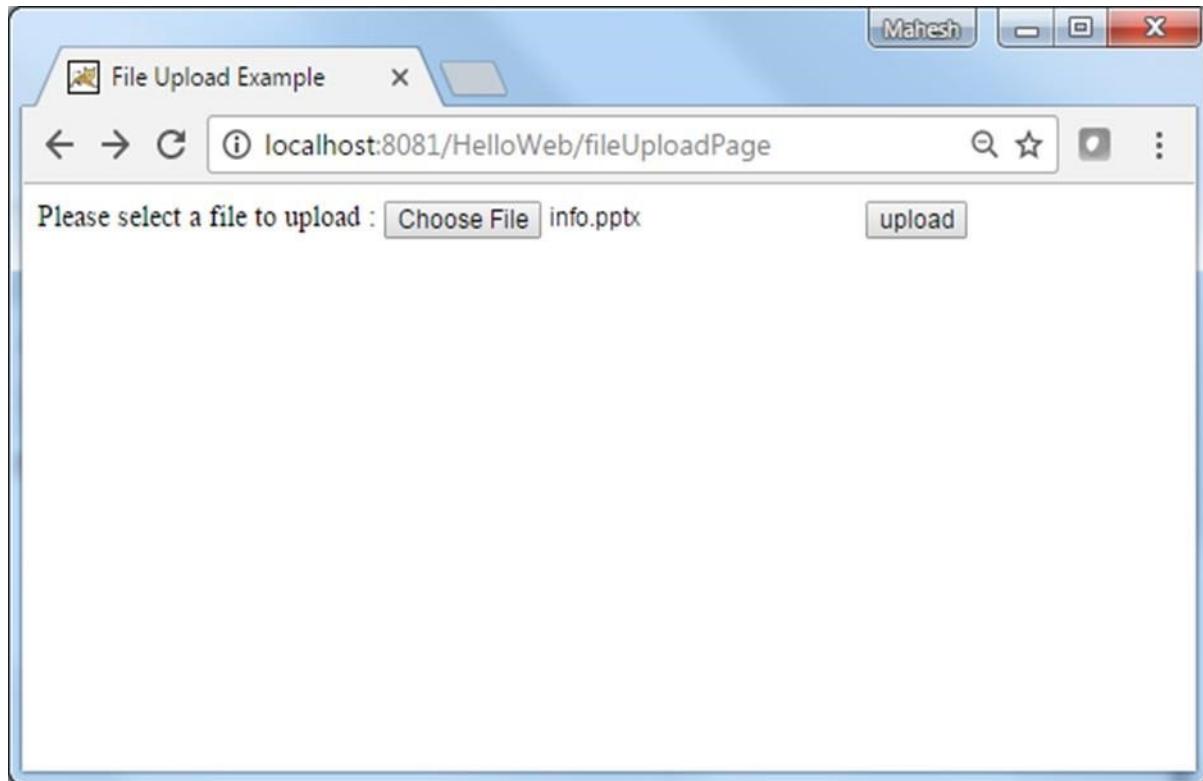
```

<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
<title>File Upload Example</title>
</head>
<body>
    FileName :
    <b> ${fileName} </b> - Uploaded Successfully.
</body>
</html>

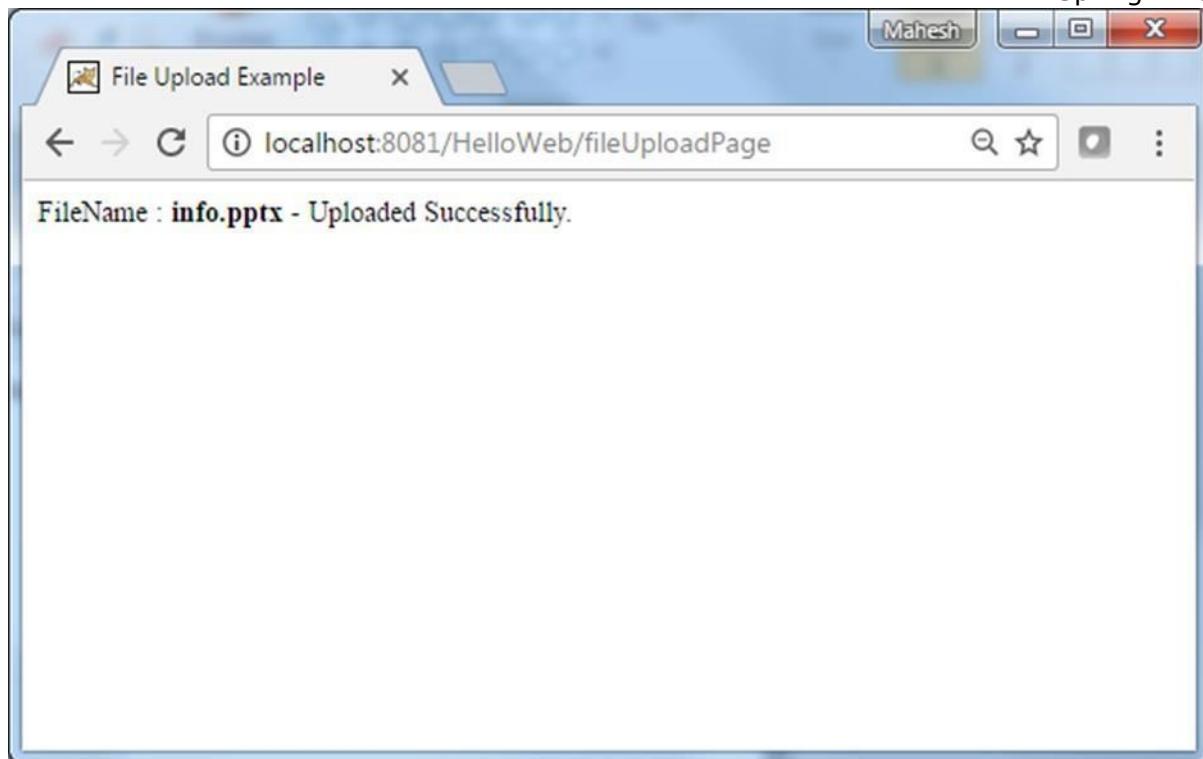
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the HelloWeb.war file in the Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL- **http://localhost:8080/HelloWeb/fileUploadPage** and we will see the following screen, if everything is fine with the Spring Web Application.



After submitting the required information, click on the submit button to submit the form. You should see the following screen, if everything is fine with the Spring Web Application.



Spring MVC – Integration

30. SpringMVC– HibernateValidator

The following example shows how to use Error Handling and Validators in forms using the Spring Web MVC framework. To begin with, let us have a working Eclipse IDE in place and adhere to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with the name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes Student, StudentController and StudentValidator under the com.ai package.
3	Create view files addStudent.jsp and result.jsp under the jsp sub-folder.
4	Download Hibernate Validator library Hibernate Validator . Extract hibernatevalidator-5.3.4.Final.jar and required dependencies present under the required folder of the downloaded zip file. Put them in your CLASSPATH.
5	Create a properties file messages.properties under the SRC folder.
6	The final step is to create the content of the source and configuration files and export the application as explained below.

Student.java

```
package com.ai;

import org.hibernate.validator.constraints.NotEmpty;
import org.hibernate.validator.constraints.Range;

public class Student {

    @Range(min = 1, max = 150)
    private Integer age;
    @NotEmpty
    private String name;
    private Integer id;

    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }

    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    public void setId(Integer id) {
        this.id = id;
    }
    public Integer getId() {
        return id;
    }
}
```

StudentController.java

```
package com.ai;

import org.springframework.stereotype.Controller; import
org.springframework.ui.Model;
import      org.springframework.validation.BindingResult;      import
org.springframework.validation.annotation.Validated;      import
org.springframework.web.bind.annotationModelAttribute;      import
org.springframework.web.bind.annotation.RequestMapping;      import
org.springframework.web.bind.annotation.RequestMethod;      import
org.springframework.web.servlet.ModelAndView;

@Controller
public class StudentController {

    @RequestMapping(value = "/addStudent", method = RequestMethod.GET) public
    ModelAndView student() {
```

```

        return new ModelAndView("addStudent", "command", new Student());
    }

    @ModelAttribute("student")
    public Student createStudentModel() {
        return new Student();
    }

    @RequestMapping(value = "/addStudent", method = RequestMethod.POST)
    public String addStudent(@ModelAttribute("student") @Validated Student
        student, BindingResult bindingResult, Model model) { if
        (bindingResult.hasErrors()) { return "addStudent";
        }
        model.addAttribute("name", student.getName());
        model.addAttribute("age", student.getAge());
        model.addAttribute("id", student.getId());

        return "result";
    }
}

```

messages.properties

NotEmpty.student.name = Name is required!
Range.student.age = Age value must be between 1 and 150!

Here, the key is <Annotation>.<object-name>.<attribute>. Value is the message to be displayed.

TestWeb-servlet.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       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
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd

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

    <context:component-scan base-package="com.ai" />
    <mvc:annotation-driven />
    <bean class="org.springframework.context.support.ResourceBundleMessageSource"
          id="messageSource">
        <property name="basename" value="messages" />
    </bean>
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

Here, for the first service method **student()**, we have passed a blank **Studentobject** in the ModelAndView object with name "command", because the spring framework expects an object with name "command", if you are using **<form:form>** tags in your JSP file. So, when the **student()** method is called, it returns **addStudent.jsp** view.

The second service method **addStudent()** will be called against a POST method on the **HelloWeb/addStudent** URL. You will prepare your model object based on the submitted information. Finally, a "result" view will be returned from the service method, which will result in rendering the result.jsp. In case there are errors generated using validator then same view "addStudent" is returned, Spring automatically injects error messages from **BindingResult** in view.

addStudent.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head> <style>
.error { color: #ff0000;
}

.errorblock { color: #000;
background-color: #ffEEEE; border: 3px
solid #ff0000; padding: 8px;
```

```

        margin: 16px;
    }
</style>
<body>

<h2>Student Information</h2>
<form:form method="POST" action="/TestWeb/addStudent" commandName="student">
    <form:errors path="*" cssClass="errorblock" element="div" />
    <table>
        <tr>
            <td><form:label path="name">Name</form:label></td>
            <td><form:input path="name" /></td>
            <td><form:errors path="name" cssClass="error" /></td>
        </tr>
        <tr>
            <td><form:label path="age">Age</form:label></td>
            <td><form:input path="age" /></td>
            <td><form:errors path="age" cssClass="error" /></td>
        </tr>
        <tr>
            <td><form:label path="id">id</form:label></td>
            <td><form:input path="id" /></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Submit"/>
            </td>
        </tr>
    </table>
</form:form>
</body>
</html>

```

Here, we are using the **<form:errors />** tag with path="*" to render error messages. For example –

```
<form:errors path="*" cssClass="errorblock" element="div" />
```

It will render error messages for all input validations. We are using **<form:errors />** tag with path="name" to render error message for the name field.

For example–

```
<form:errors path="name" cssClass="error" />
<form:errors path="age" cssClass="error" />
```

It will render error messages for name and age field validations.

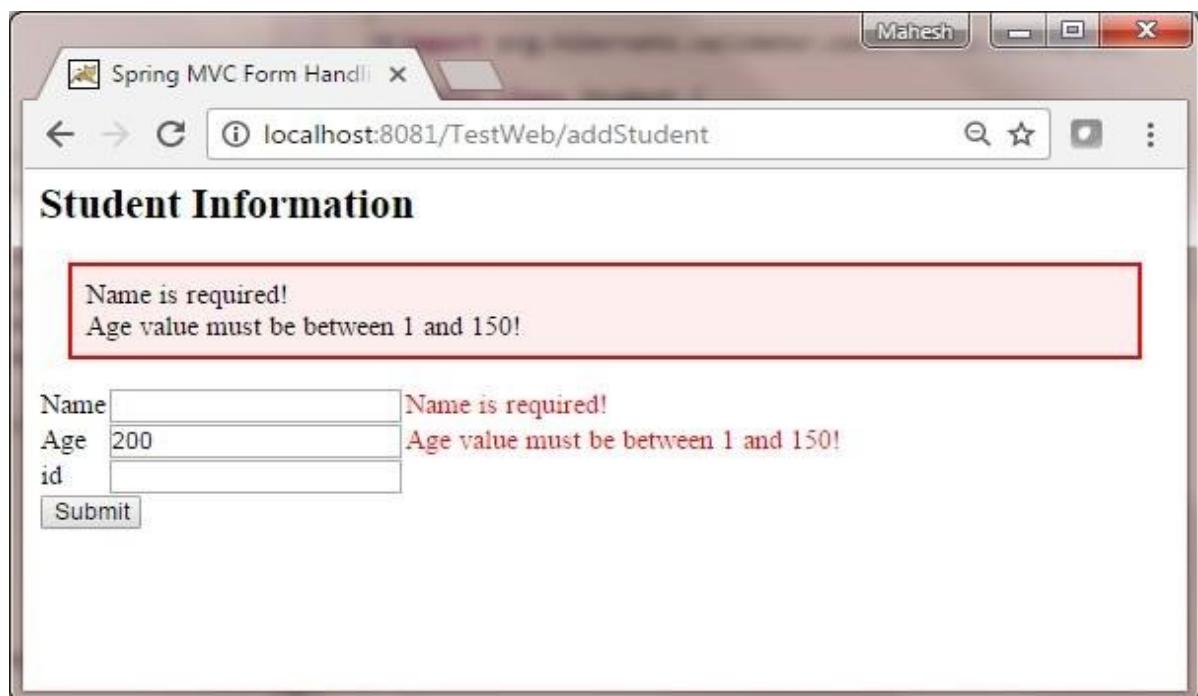
result.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring MVC Form Handling</title>
</head>
<body>

<h2>Submitted Student Information</h2>
<table> <tr>
    <td>Name</td>
    <td>${name}</td>
</tr> <tr>
    <td>Age</td>
    <td>${age}</td>
</tr> <tr>
    <td>ID</td>
    <td>${id}</td>
</tr>
</table>
</body>
</html>
```

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **HelloWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL - **http://localhost:8080/TestWeb/addStudent** and we will see the following screen, if you have entered invalid values.



31. Spring MVC – Generate RSS Feed

The following example shows how to generate RSS Feed using the Spring Web MVC Framework. To start with, let us have a working Eclipse IDE in place and then consider the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with the name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes RSSMessage, RSSFeedViewer and RSSController under the com.ai package.
3	Download the Rome library – Rome and its dependencies rome-utils, jdom and slf4j from the same maven repository page. Put them in your CLASSPATH.
4	Create a properties file messages.properties under the SRC folder.
5	The final step is to create the content of the source and configuration files and export the application as explained below.

RSSMessage.java

```
package com.ai;

import java.util.Date;

public class RSSMessage {
    String title;
    String url;
    String summary;    Date
    createdDate; public String
    getTitle() { return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getUrl() {

        return url;
    }
    public void setUrl(String url) { this.url
        = url;
    }
    public String getSummary() { return
        summary;
    }
    public void setSummary(String summary) {
        this.summary = summary;
    }
    public Date getCreatedDate() { return
        createdDate;
    }
    public void setCreatedDate(Date createdDate)
    { this.createdDate = createdDate; }
}
```

RSSFeedViewer.java

```
package com.ai;

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.view.feed.AbstractRssFeedView;

import com.rometools.rome.feed.rss.Channel;
import com.rometools.rome.feed.rss.Content;
import com.rometools.rome.feed.rss.Item;

public class RSSFeedViewer extends AbstractRssFeedView {

    @Override
    protected void buildFeedMetadata(Map<String, Object> model, Channel feed,
```

```

        HttpServletRequest request) {

    feed.setTitle("Ai Dot Com");
    feed.setDescription("Java Tutorials and Examples");
    feed.setLink("http://www.ai.com");

    super.buildFeedMetadata(model, feed, request);
}

@Override
protected List<Item> buildFeedItems(Map<String, Object> model,
    HttpServletRequest request, HttpServletResponse response) throws Exception {

    List<RSSMessage> listContent = (List<RSSMessage>) model.get("feedContent");
    List<Item> items = new ArrayList<Item>(listContent.size());

    for(RSSMessage tempContent : listContent ){

        Item item = new Item();

        Content content = new Content();
        content.setValue(tempContent.getSummary());
        item.setContent(content);

        item.setTitle(tempContent.getTitle()); item.setLink(tempContent.getUrl());
        item.setPubDate(tempContent.getCreatedDate());

        items.add(item);
    }

    return items;
}
}

```

RSSController.java

```
package com.ai;
```

```
import java.util.ArrayList;
import java.util.Date; import
java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class RSSController {
    @RequestMapping(value="/rssfeed", method = RequestMethod.GET)
    public ModelAndView getFeedInRss() {
        List<RSSMessage> items = new ArrayList<RSSMessage>();

        RSSMessage content = new RSSMessage();
        content.setTitle("Spring Tutorial");
        content.setUrl("http://www.ai/spring");
        content.setSummary("Spring tutorial summary...");
        content.setCreatedDate(new Date()); items.add(content);

        RSSMessage content2 = new RSSMessage();
        content2.setTitle("Spring MVC");
        content2.setUrl("http://www.ai/springmvc");
        content2.setSummary("Spring MVC tutorial summary...");
        content2.setCreatedDate(new Date()); items.add(content2);

        ModelAndView mav = new ModelAndView();
        mav.setViewName("rssViewer");
        mav.addObject("feedContent", items);

        return mav;
    }
}
```

TestWeb-servlet.xml

```

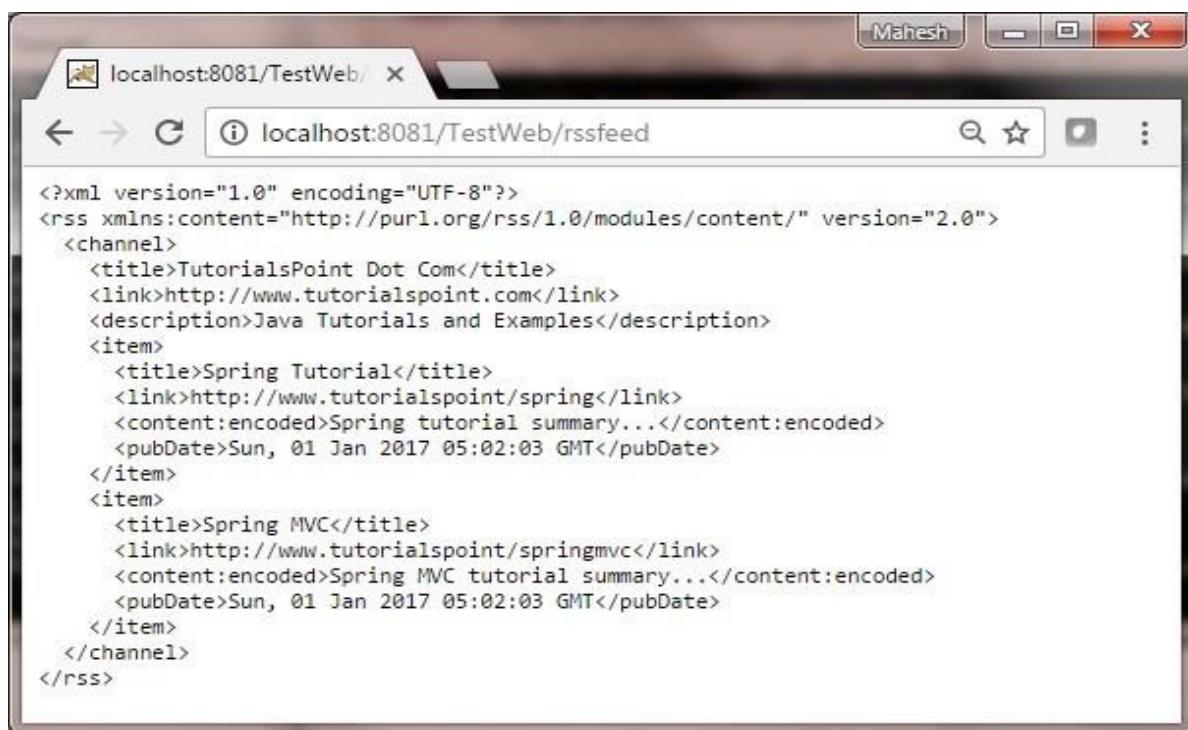
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:component-scan base-package="com.ai" />

    <bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />
    <bean id="rssViewer" class="com.ai.RSSFeedViewer" />
</beans>
```

Here, we have created a RSS feed POJO RSSMessage and a RSS Message Viewer, which extends the **AbstractRssFeedView** and overrides its method. In RSSController, we have generated a sample RSS Feed.

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **TestWeb.war** file in Tomcat's webapps folder.

Now, start your Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL – **http://localhost:8080/TestWeb/rssfeed** and we will see the following screen.



The screenshot shows a web browser window titled "localhost:8081/TestWeb" with the URL "localhost:8081/TestWeb/rssfeed". The page content is an XML RSS feed. The XML code is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:content="http://purl.org/rss/1.0/modules/content/" version="2.0">
  <channel>
    <title>TutorialsPoint Dot Com</title>
    <link>http://www.tutorialspoint.com</link>
    <description>Java Tutorials and Examples</description>
    <item>
      <title>Spring Tutorial</title>
      <link>http://www.tutorialspoint/spring</link>
      <content:encoded>Spring tutorial summary...</content:encoded>
      <pubDate>Sun, 01 Jan 2017 05:02:03 GMT</pubDate>
    </item>
    <item>
      <title>Spring MVC</title>
      <link>http://www.tutorialspoint/springmvc</link>
      <content:encoded>Spring MVC tutorial summary...</content:encoded>
      <pubDate>Sun, 01 Jan 2017 05:02:03 GMT</pubDate>
    </item>
  </channel>
</rss>
```

32. SpringMVC – Generate XML Example

The following example shows how to generate XML using the Spring Web MVC Framework. To begin with, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User and UserController under the com.aipackage.
3	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "user")
public class User { private
String name; private int id;
public String getName() {
return name;
}
@XmlElement
public void setName(String name) { this.name
= name;
}
public int getId() { return
id;
}
@XmlElement
public void setId(int id) {
```

Spring MVC

```
    this.id = id;
}
}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc" xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
```

UserController.java

```
package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/user") public
class UserController {

    @RequestMapping(value="{name}", method = RequestMethod.GET)
    public @ResponseBody User getUser(@PathVariable String name) {

        User user = new User();

        user.setName(name);
        user.setId(1); return
        user;
    }
}
```

TestWeb-servlet.xml

Spring MVC

```
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
<context:component-scan base-package="com.ai" />
<mvc:annotation-driven />
</beans>
```

Here, we have created an XML Mapped POJO User and in the UserController, we have returned the User. Spring automatically handles the XML conversion based on **RequestMapping**.

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save your **TestWeb.war** file in Tomcat's webapps folder.

33. SpringMVC– GenerateJSON

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL – **http://localhost:8080/TestWeb/mahesh** and we will see the following screen.



The following example shows how to generate JSON using the Spring Web MVC Framework. To start with, let us have a working Eclipse IDE in place and consider the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description

1	Create a project with a name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes User and UserController under the com.ai package.
3	Download Jackson libraries – Jackson Core , Jackson Databind and Jackson Annotations from maven repository page. Put them in your CLASSPATH.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

User.java

```
package com.ai;

public class User { private
    String name; private int
    id; public String
    getName() { return name;
    }
    public void setName(String name) { this.name
        = name;
    }
    public int getId() { return
        id;
    }
    public void setId(int id) { this.id
        = id;
    } }
```

Spring MVC

UserController.java

```
package com.ai;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/user")
public class UserController {

    @RequestMapping(value = "{name}", method = RequestMethod.GET)
    public @ResponseBody User getUser(@PathVariable String name) {

        User user = new User();
        user.setName(name);
        user.setId(1);
        return user;
    }
}
```

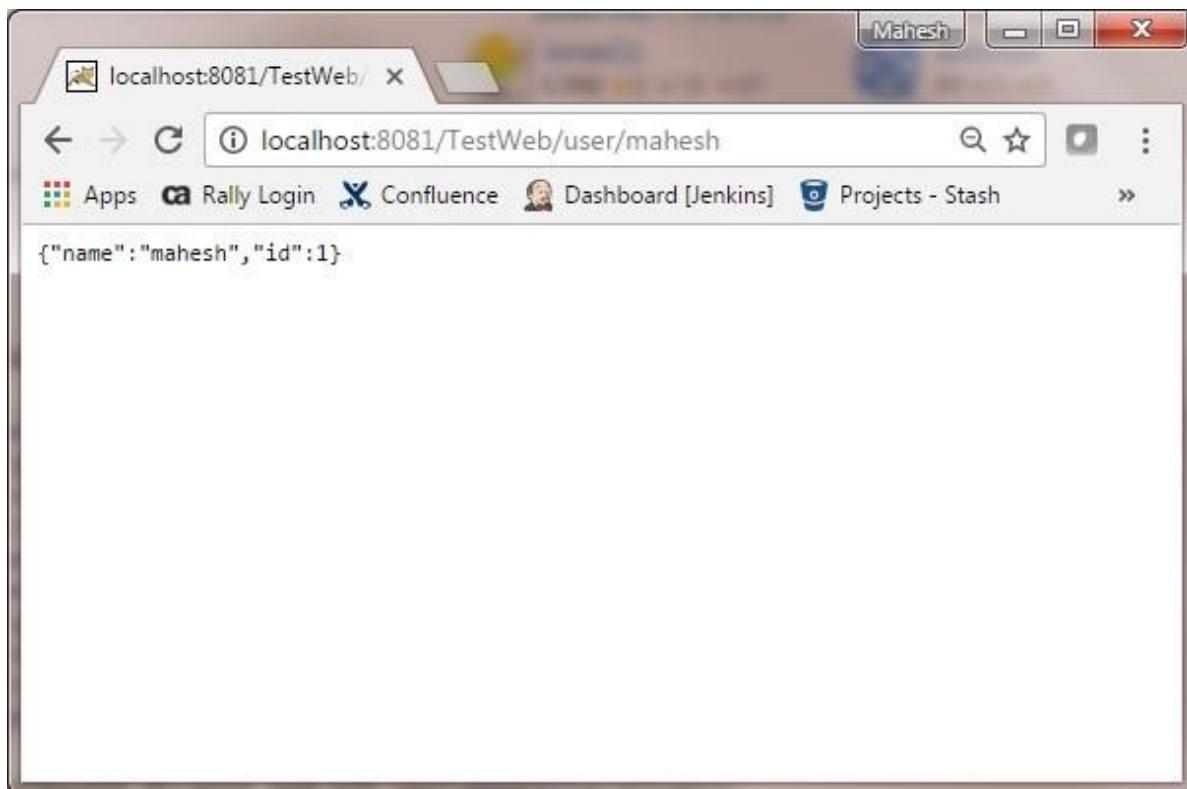
TestWeb-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc" xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
    <context:component-scan base-package="com.ai" />
    <mvc:annotation-driven />
</beans>
```

Here, we have created a Simple POJO User and in UserController we have returned the User. Spring automatically handles the JSON conversion based on RequestMapping and Jackson jar present in the classpath.

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export** → **WAR File** option and save your **TestWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL – **http://localhost:8080/TestWeb/mahesh** and we will see the following screen.



The following example shows how to generate Excel using the Spring Web MVC Framework. To begin with, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

34. SpringMVC– GenerateExcel

Step	Description
1	Create a project with a name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes UserExcelView and ExcelController under the com.ai package.
3	Download the Apache POI library – Apache POI from the maven repository page. Put it in your CLASSPATH.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

ExcelController.java

```
package com.ai;

import java.util.HashMap; import
java.util.Map;

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 ExcelController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        //user data
        Map<String, String> userData = new HashMap<String, String>();
```

```

        userData.put("1", "Mahesh");
        userData.put("2", "Suresh");
        userData.put("3", "Ramesh");
        userData.put("4", "Naresh");
        return new ModelAndView("UserSummary", "userData", userData);
    }
}

```

UserExcelView.java

```

package com.ai;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.springframework.web.servlet.view.document.AbstractExcelView;

public class UserExcelView extends AbstractExcelView {

    @Override
    protected void buildExcelDocument(Map<String, Object> model,
        HSSFWorkbook workbook, HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        Map<String, String> userData = (Map<String, String>) model.get("userData");
        //create a wordsheet
        HSSFSheet sheet = workbook.createSheet("User Report");

        HSSFRow header = sheet.createRow(0);
        header.createCell(0).setCellValue("Roll No");
        header.createCell(1).setCellValue("Name");

        int rowNum = 1;
        for (Map.Entry<String, String> entry : userData.entrySet()) {
            //create the row data
            HSSFRow row = sheet.createRow(rowNum++);
            row.createCell(0).setCellValue(entry.getKey());
            row.createCell(1).setCellValue(entry.getValue());
        }
    }
}

```

```

        row.createCell(0).setCellValue(entry.getKey());
        row.createCell(1).setCellValue(entry.getValue());
    }

}
}

```

TestWeb-servlet.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc" xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <bean

        class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />

    <bean class="com.ai.ExcelController" />

    <bean class="org.springframework.web.servlet.view.XmlViewResolver"> <property
        name="location">
        <value>/WEB-INF/views.xml</value>
    </property>
    </bean>
</beans>

```

views.xml

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

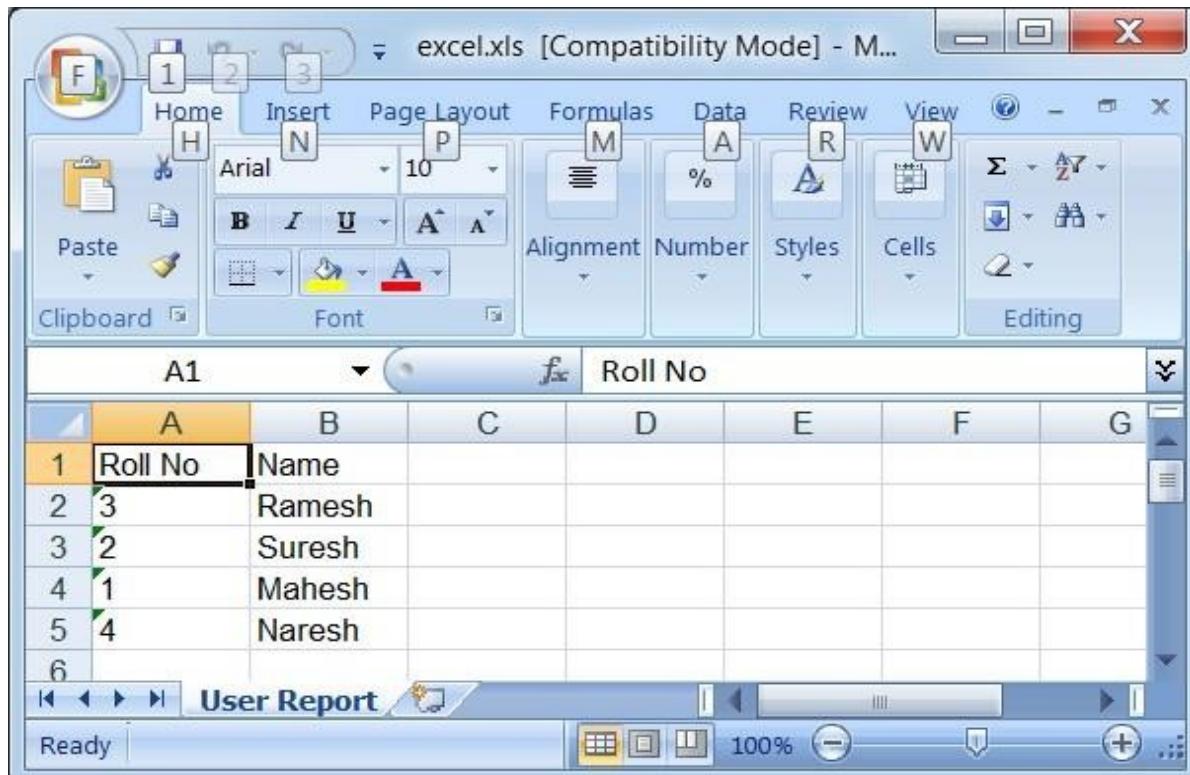
```
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
```

```
<bean id="UserSummary" class="com.ai.UserExcelView"></bean> </beans>
```

Here, we have created an ExcelController and an ExcelView. Apache POI library deals with Microsoft Office file formats and will convert the data to an excel document.

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **TestWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL – **http://localhost:8080/TestWeb/excel** and we will see the following screen.



35. SpringMVC– GeneratePDF

The following example shows how to generate a PDF using the Spring Web MVC Framework. To start with, let us have a working Eclipse IDE in place and adhere to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

Step	Description
1	Create a project with a name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create Java classes UserPDFView and PDFController under the com.ai package.
3	Download the iText library – iText from the maven repository page. Put it in your CLASSPATH.
4	The final step is to create the content of the source and configuration files and export the application as explained below.

PDFController.java

```
package com.ai;

import java.util.HashMap; import
java.util.Map;

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 PDFController extends AbstractController {
    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        //user data
        Map<String, String> userData = new HashMap<String, String>();
        userData.put("1", "Mahesh");
    }
}
```

```
        userData.put("2", "Suresh");
        userData.put("3", "Ramesh");
        userData.put("4", "Naresh");
        return new ModelAndView("UserSummary", "userData", userData);
    }
}
```

UserExcelView.java

```

package com.ai;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.view.document.AbstractPdfView;

import com.lowagie.text.Document; import
com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class UserPDFView extends AbstractPdfView { protected void
buildPdfDocument(Map<String, Object> model, Document document, PdfWriter
pdfWriter, HttpServletRequest request, HttpServletResponse response) throws
Exception {
    Map<String, String> userData = (Map<String, String>) model.get("userData");

    Table table = new Table(2);
    table.addCell("Roll No");
    table.addCell("Name");

    for (Map.Entry<String, String> entry : userData.entrySet()) {
        table.addCell(entry.getKey()); table.addCell(entry.getValue());
    }
    document.add(table);
}
}

```

TestWeb-servlet.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc" xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">
    <bean

        class="org.springframework.web.servlet.support.ControllerClassNameHandlerMapping" />

    <bean class="com.ai.PDFController" />
    <bean class="org.springframework.web.servlet.view.XmlViewResolver"> <property
        name="location">
        <value>/WEB-INF/views.xml</value>
    </property>
    </bean>
</beans>
```

views.xml

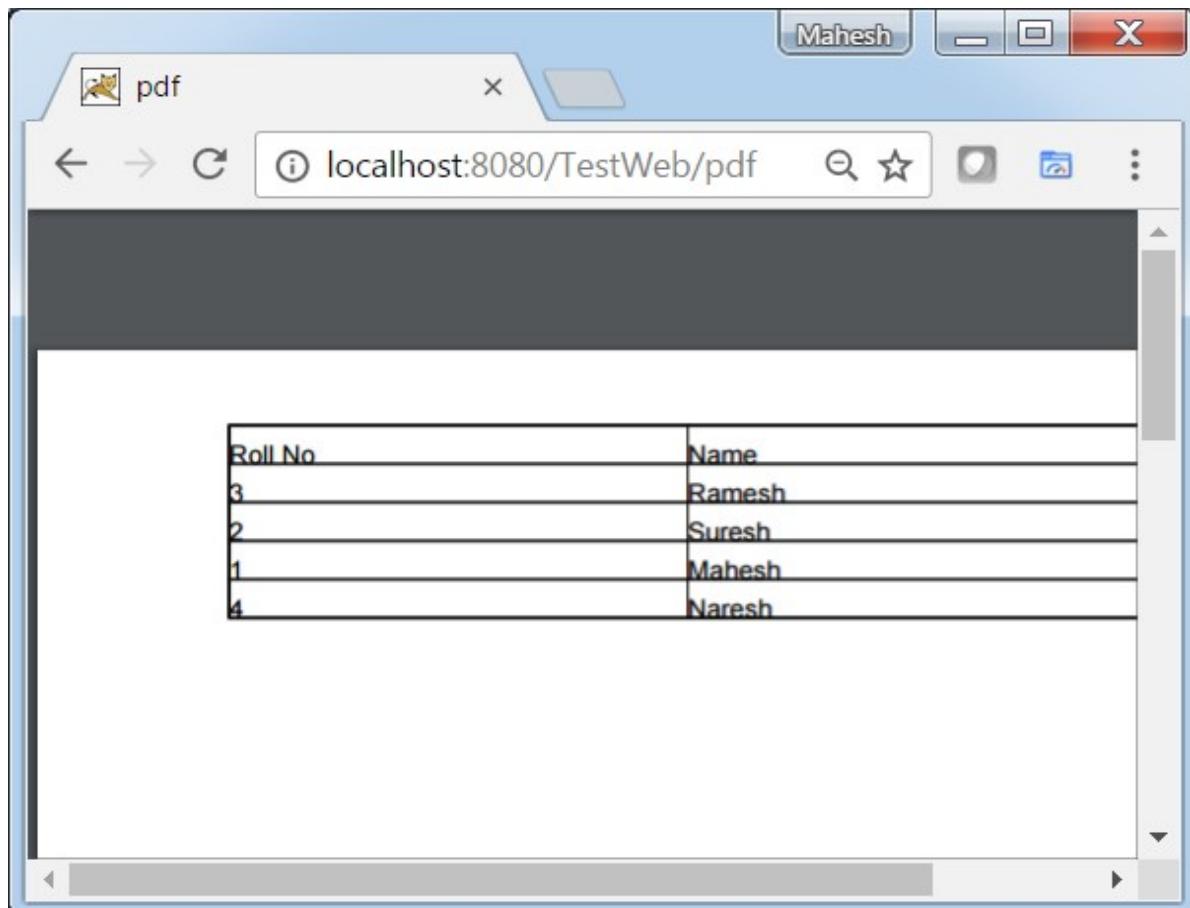
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="UserSummary" class="com.ai.UserPDFView"></bean>
</beans>
```

Here, we have created a PDFController and UserPDFView. iText library deals with the PDF file formats and will convert the data to a PDF document.

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save the **TestWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. We can also try the following URL – **http://localhost:8080/TestWeb/pdf** and if all goes as planned, we will see the following screen.



The following example shows how to integrate LOG4J using the Spring Web MVC Framework. To start with, let us have a working Eclipse IDE in place and stick to the following steps to develop a Dynamic Form based Web Application using the Spring Web Framework.

36. SpringMVC– IntegrateLOG4J

Step	Description
1	Create a project with the name TestWeb under a package com.ai as explained in the Spring MVC - Hello World chapter.
2	Create a Java class HelloController under the com.aipackage.
3	Download the log4j library – LOG4J from the maven repository page. Put it in your CLASSPATH.
4	Create a log4j.properties under the SRC folder.
5	The final step is to create the content of the source and configuration files and export the application as explained below.

HelloController.java

```
package com.ai;

import org.apache.log4j.Logger;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod; import
org.springframework.ui.ModelMap;

@Controller
@RequestMapping("/hello") public class HelloController{ private static final
Logger LOGGER = Logger.getLogger(HelloController.class);
@RequestMapping(method = RequestMethod.GET) public
String printHello(ModelMap model) {
    LOGGER.info("printHello started.");
```

```

//logs debug message if(LOGGER.isDebugEnabled()){
    LOGGER.debug("Inside: printHello");
}
//logs exception
LOGGER.error("Logging a sample exception", new Exception("Testing"));

model.addAttribute("message", "Hello Spring MVC
Framework!"); LOGGER.info("printHello ended."); return
"hello";
}
}

```

Spring MVC

log4j.properties

```

# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L
- %m%n

# Redirect log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
#outputs to Tomcat home
log4j.appender.file.File=${catalina.home}/logs/myapp.log
log4j.appender.file.MaxFileSize=5MB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L -
%m%n

```

TestWeb-servlet.xml

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mvc="http://www.springframework.org/schema/mvc"

```

```

 xsi:schemaLocation="
 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
 http://www.springframework.org/schema/context
 http://www.springframework.org/schema/context/spring-context-3.0.xsd
 http://www.springframework.org/schema/mvc
 http://www.springframework.org/schema/spring-mvc-3.0.xsd">
<context:component-scan base-package="com.ai" />

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

hello.jsp

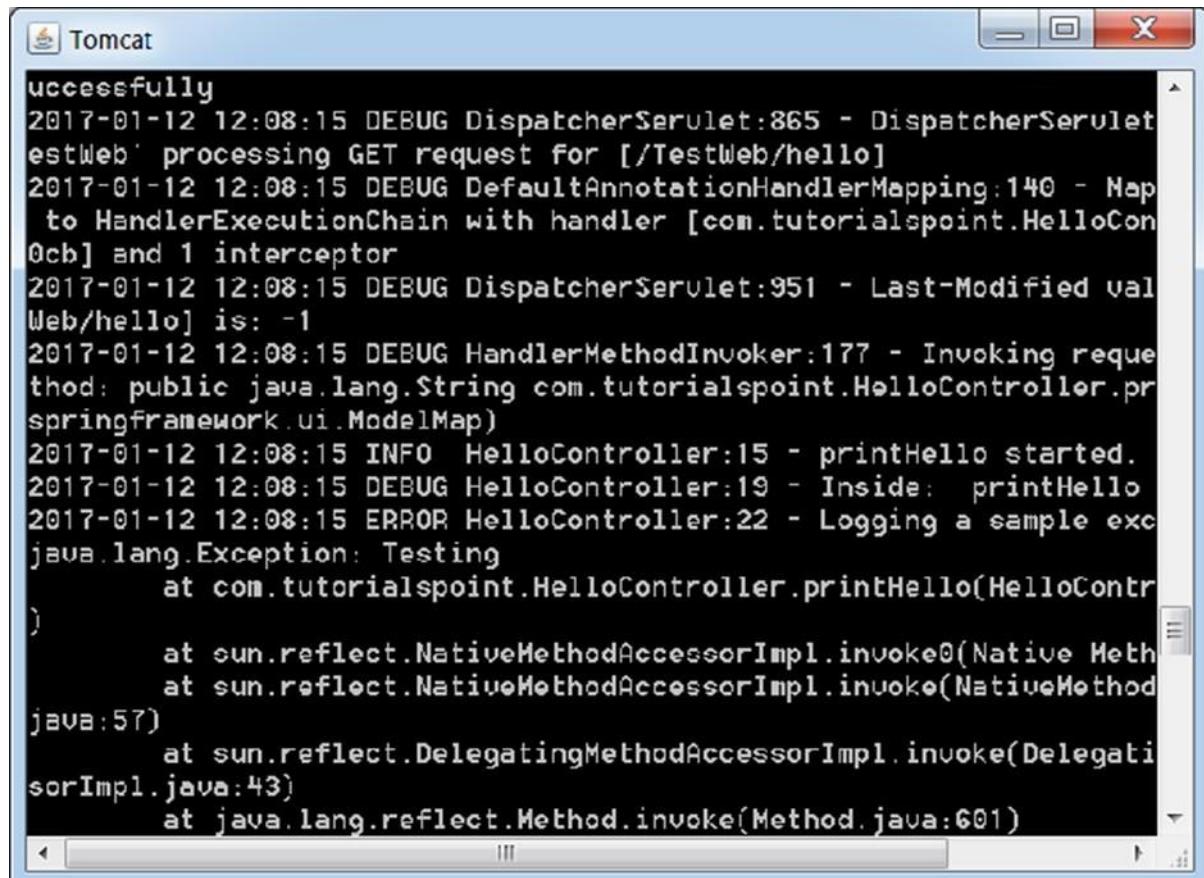
```

<%@ page contentType="text/html; charset=UTF-8" %>
<html>
<head>
<title>Hello World</title>
</head> <body>
<h2>${message}</h2>
</body>
</html>
```

Here, we have configured the LOG4J to log details on the Tomcat console and in the file present in &t; tomcat home → logs as myapp.log.

Once you are done with creating source and configuration files, export your application. Right click on your application, use **Export → WAR File** option and save your **TestWeb.war** file in Tomcat's webapps folder.

Now, start the Tomcat server and make sure you are able to access other webpages from the webapps folder using a standard browser. Try a URL –<http://localhost:8080/TestWeb/hello> and we will see the following screen on Tomcat's log.



```
Successfully
2017-01-12 12:08:15 DEBUG DispatcherServlet:865 - DispatcherServlet
estWeb' processing GET request for [/TestWeb/hello]
2017-01-12 12:08:15 DEBUG DefaultAnnotationHandlerMapping:140 - Map
to HandlerExecutionChain with handler [com.tutorialspoint.HelloCon
troller] and 1 interceptor
2017-01-12 12:08:15 DEBUG DispatcherServlet:951 - Last-Modified val
Web/hello] is: -1
2017-01-12 12:08:15 DEBUG HandlerMethodInvoker:177 - Invoking reque
st method: public java.lang.String com.tutorialspoint.HelloController.pr
springframework.ui.ModelMap)
2017-01-12 12:08:15 INFO HelloController:15 - printHello started.
2017-01-12 12:08:15 DEBUG HelloController:19 - Inside: printHello
2017-01-12 12:08:15 ERROR HelloController:22 - Logging a sample exc
java.lang.Exception: Testing
        at com.tutorialspoint.HelloController.printHello(HelloContr
)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethod
java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Delegati
orImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:601)
```