ACE Inspiration

Spring MVC

- Tutorial

Objective

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.

Contents

 Form Handling

 Page Redirection

 Hibernate Validator

What is Spring MVC

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**.

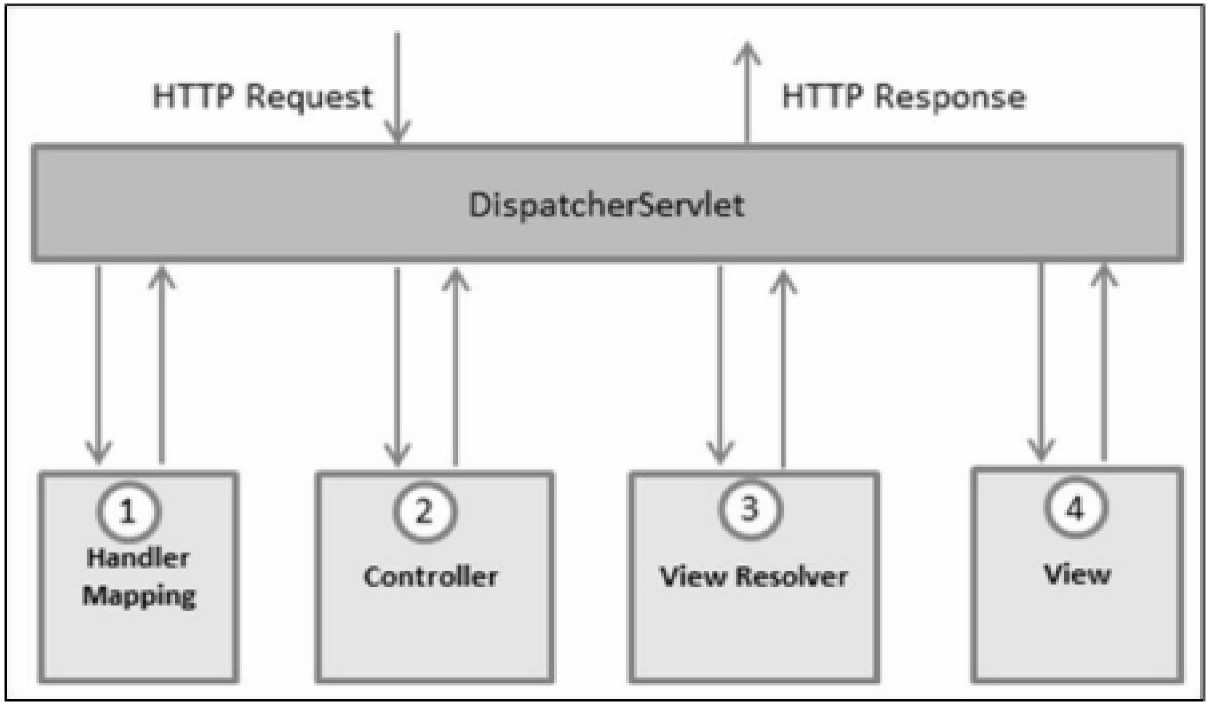
 The **View** is responsible for rendering the model data and in general, it generates

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.

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.



DispatcherServlet

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:

```
<servlet>
<servlet-name>HelloWeb</servlet-name>

<servlet-class>
       org.springframework.web.servlet.DispatcherServlet
</servlet-class>
```

```
</servlet>

<servlet-mapping>
<servlet-name>HelloWeb</servlet-name>
<url-pattern>/</url-pattern>
</servlet-mapping>
```

Required Configuration

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 willtry to load the application context from a file named **[servlet-name]-servlet.xml** located in the application'sWebContent/WEB-INF directory. In this case, our file will be **HelloWeb-servlet.xml**.

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

Required Configuration

```
<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>
```

Required Configuration

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.

Required Configuration

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";
        }
}
```

Required Configuration

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";
            }
}
```

Required Configuration

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.

Required Configuration

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.

Hello World

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.

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-INFfolder. 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.

Hello World

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.

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-INFfolder. 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";
        }
}
```

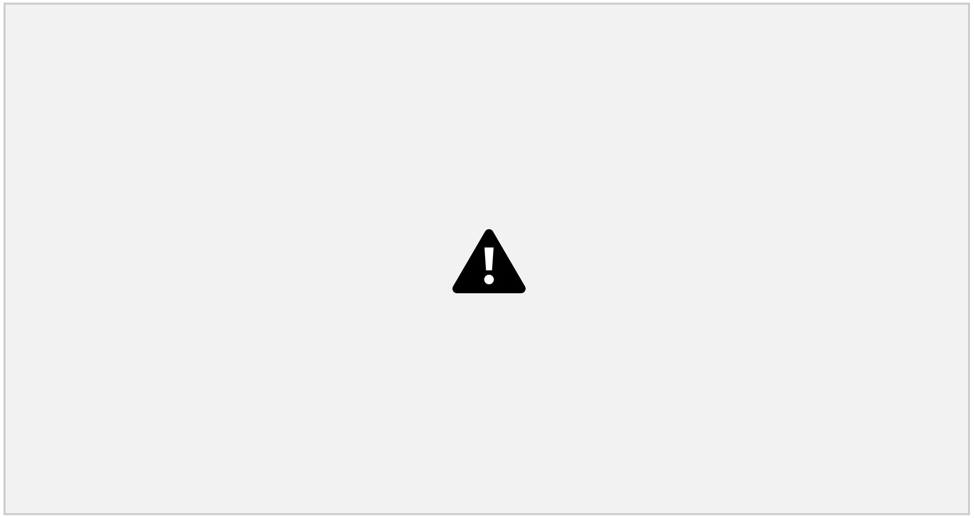
Hello World

hello.jsp

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

Hello World

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.

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:

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;
//generate getter
setter }
```

Form Handling

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", "student", , new Student());
}
```

Form Handling

StudentController.java

```
@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". 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.

Form Handling

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"
modelAttribute="student">
<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>
```

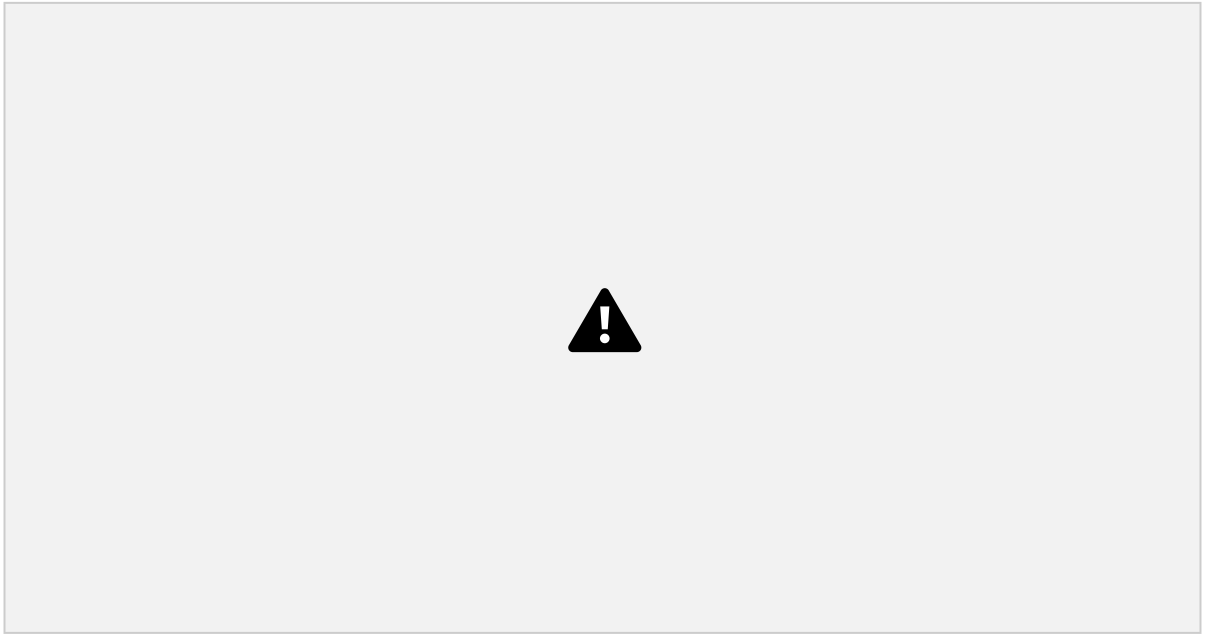
Form Handling

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>
```

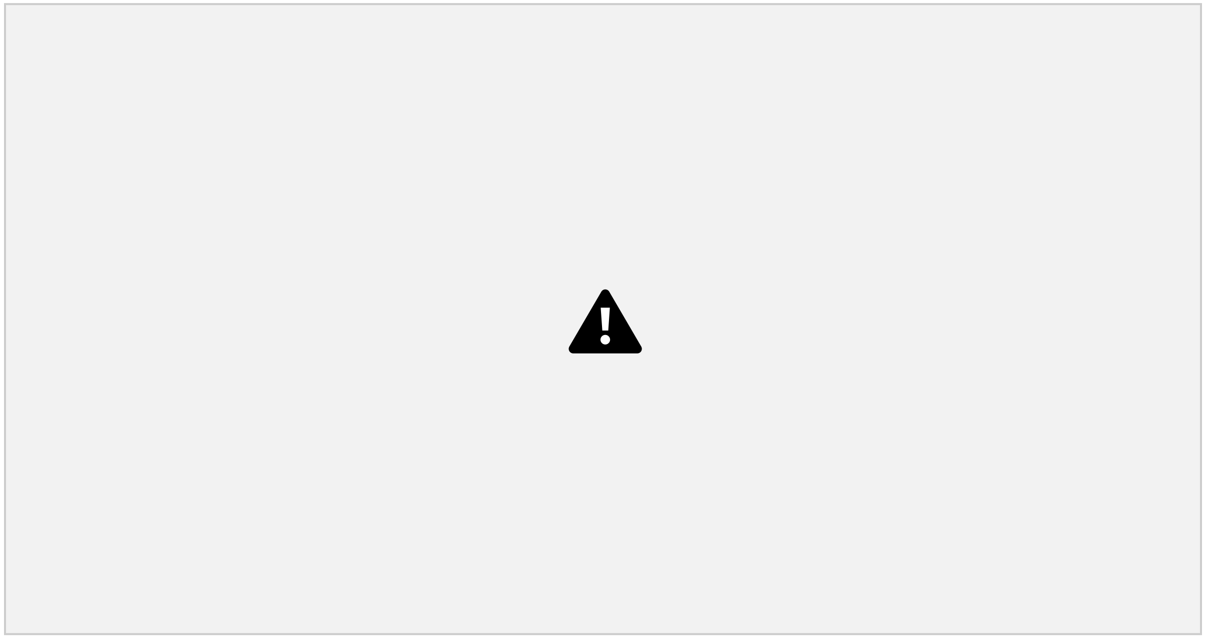
Form Handling

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.



Form Handling

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.



Page Redirection

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:

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.

Page Redirection

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"; } }
```

Page Redirection

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>
```

Page Redirection

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>
</head>
```

Hibernate Validator

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.

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 hibernate

validator-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.

Hibernate Validator

```
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;

//generate getter setter
```

Hibernate Validator

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

```
        return new ModelAndView("addStudent", "student", new Student());
    }
```

Hibernate Validator

```
@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";
        }
}
```

Hibernate Validator

messages.properties

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

TestWeb-servlet.xml

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

Hibernate Validator

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"
modelAttribute="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>
```

Hibernate Validator

```
<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>
```

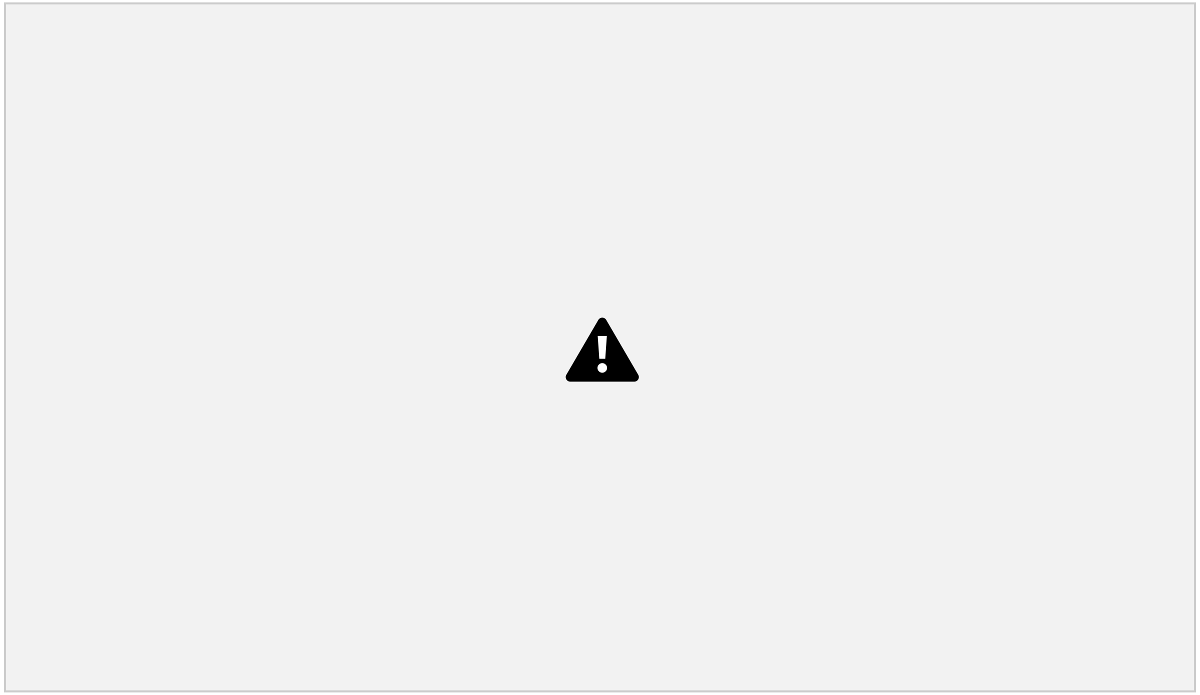
Hibernate Validator

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>
```

Hibernate Validator

Try a URL – **http://localhost:8080/TestWeb/addStudent** and we will see the following screen, if you have entered invalid values.



Thank

you!!

Q&As

References

- https://www.javatpoint.com/spring-mvc tutorial#:~:text=A%20Spring%20MVC%20is%20a,Inversion%20of%20Control%2C%2 0Dependency%20Injection.
- https://www.baeldung.com/spring-mvc-tutorial