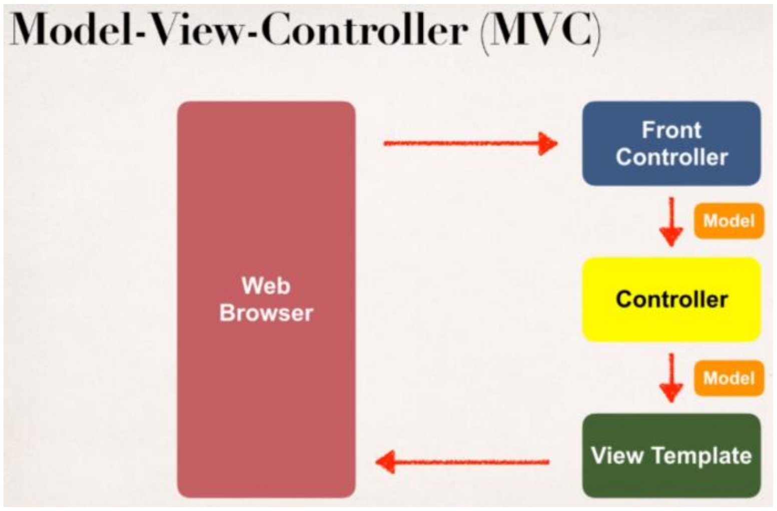
Section 11: Spring MVC - Building Spring Web Apps

0 / 8 | 26min

➢ Overview:

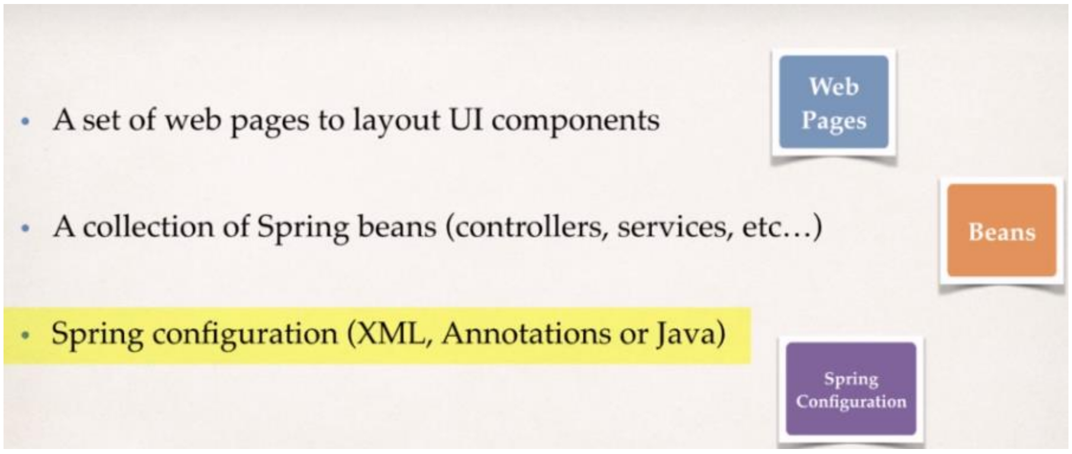
- Framework for building web applications in Java
- Based on Model-View-Controller design pattern.
- Leverages features of the core Spring Framework (IoC, DI).



Basically, we have an incoming request coming from a browser and it will encounter the Spring MVC front controller and it will actually delegate the request of to a Controller code that contains the business logic and then it will create a model and send back to the Front Controller and then it will pass the model to our View Template and View Template is like a JSP page or HTML page which will render the data and give as a response to the browser.

➢ Spring MVC Behind the Scenes:

Components of a Spring MVC Application



- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc…)
- Spring configuration (XML, Annotations or Java)

Web Pages
Beans
Spring Configuration

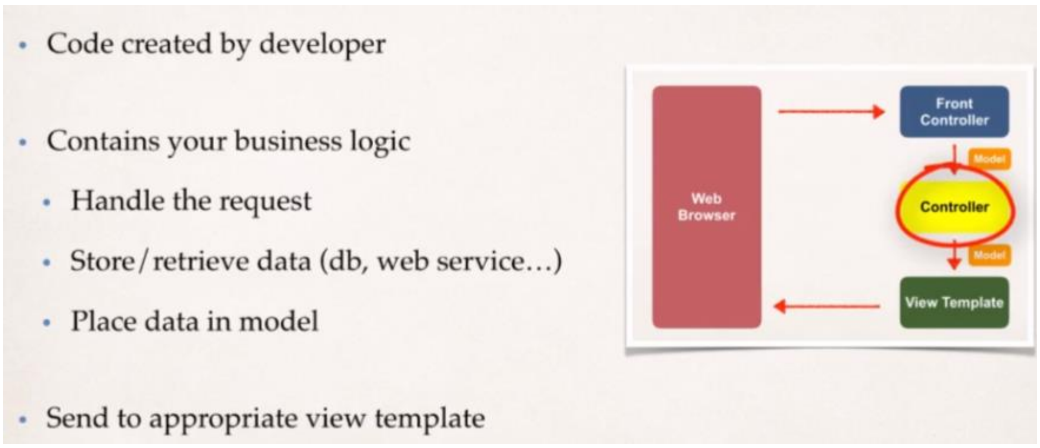
Spring MVC Front Controller:

- Front Controller known as ***DispatcherServlet*.**
  - Part of the Spring Framework.
  - Already developed by Spring Dev Team

  This Front Controller will actually delegate the requests coming from browser to some other objects in our system.

- So as a developer we will create
  - **M**odel Objects
  - **V**iew Templates (like JSP page)
  - **C**ontroller Classes (Actual business logic)

Controller:

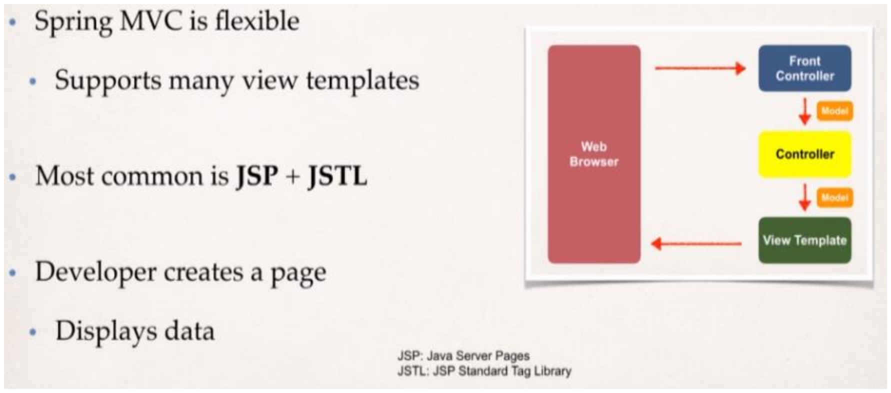


- Code created by developer
- Contains your business logic
  - Handle the request
  - Store/retrieve data (db, web service…)
  - Place data in model
- Send to appropriate view template

Web Browser
Front Controller
Model
Controller
Model
View Template

View Template:

Spring MVC is very flexible and there is many view template types. The model data comes to our JSP page and view template will read that data and display it. Other view Templates supported are Thymeleaf, Groovy, Velocity, Freemarker, etc.



- Spring MVC is flexible
  - Supports many view templates
- Most common is **JSP + JSTL**
- Developer creates a page
  - Displays data

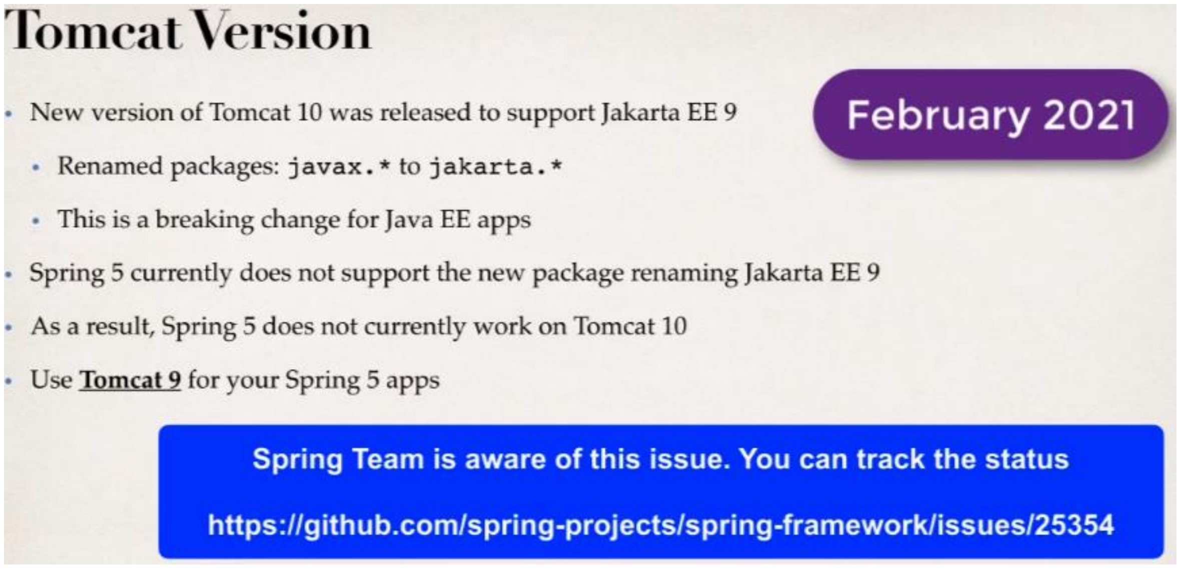
JSP: Java Server Pages
JSTL: JSP Standard Tag Library

Web Browser
Front Controller
Model
Controller
Model
View Template

Development Environment Checkpoint:

We should have installed:

- ✓ Apache Tomcat
- ✓ Eclipse
- ✓ Connected Eclipse to Tomcat



Tomcat Version

- New version of Tomcat 10 was released to support Jakarta EE 9
  - Renamed packages: `javax.*` to `jakarta.*`
  - This is a breaking change for Java EE apps
- Spring 5 currently does not support the new package renaming Jakarta EE 9
- As a result, Spring 5 does not currently work on Tomcat 10
- Use Tomcat 9 for your Spring 5 apps

February 2021

Spring Team is aware of this issue. You can track the status

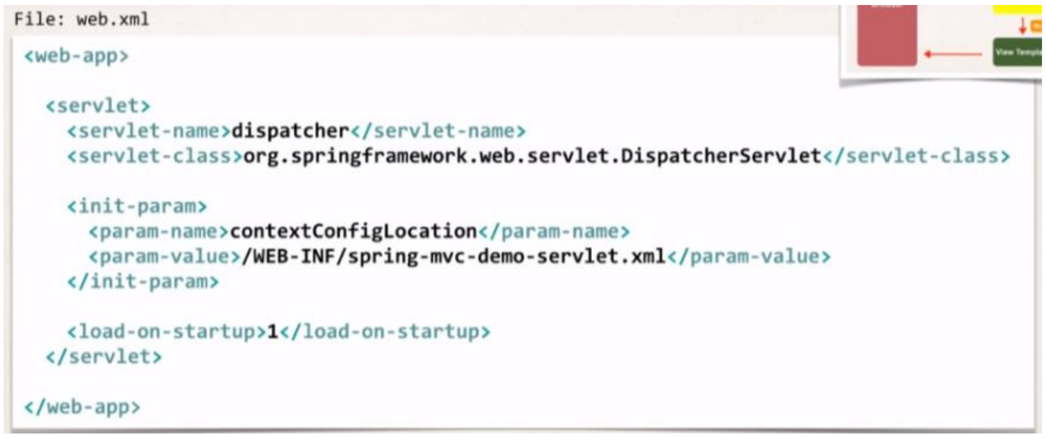
https://github.com/spring-projects/spring-framework/issues/25354

Spring MVC Configuration:

Part 1: Add Configurations to file: WEB-INF/web.xml

- Configure Spring MVC Dispatcher Servlet.

  Here we will add an entry for Spring Dispatcher Servlet or the Front Controller within <servlet-name> tag. And we don't need to do anything for Spring Dispatcher servlet as we will get it from spring jar files.
  And then we will set up the initial parameter in which we tell where our spring context config file is located.

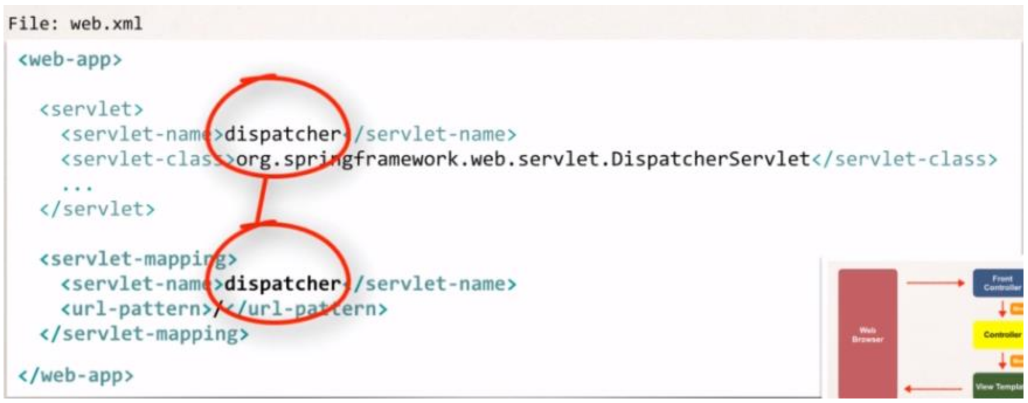


File: web.xml

```
<web-app>

  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
    </init-param>

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

</web-app>
```

- Set up URL mappings to Spring MVC Dispatcher Servlet

  Next thing is we want to tell the Spring that for any URL pattern coming in, we like it to pass to our Dispatcher Servlet. We can give what ever pattern we want till we are consistent with it.

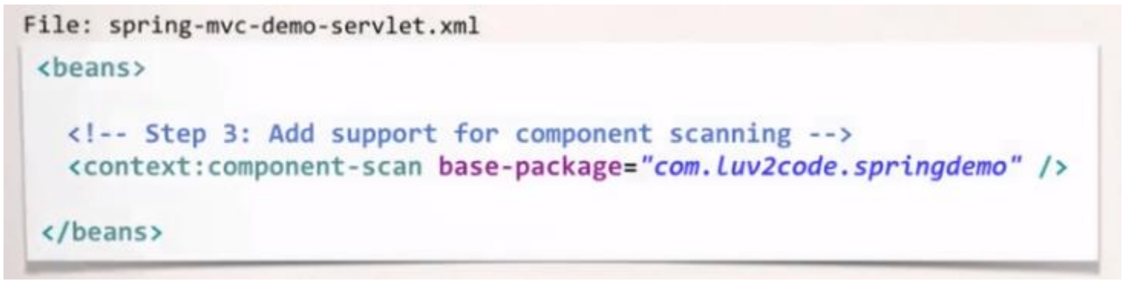


```
File: web.xml
<web-app>

  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    ...
  </servlet>

  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

  And the servlet-name while defining URL Pattern must matches with the servlet reference that we have set up above in step 1.

Part 2: Add configurations to File: WEB_INF/spring-mvc-demo-servlet.xml

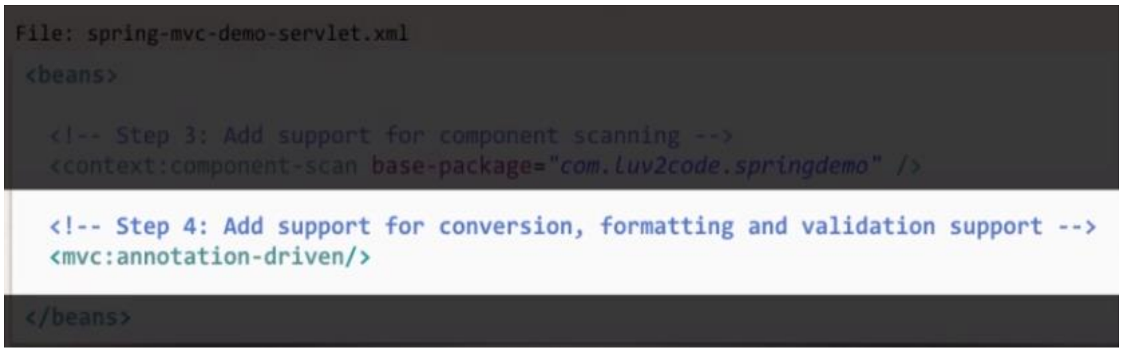
- Add support for Spring component scanning.



```
File: spring-mvc-demo-servlet.xml
<beans>

  <!-- Step 3: Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

</beans>
```

- Also add Support for conversion, formatting and validation.

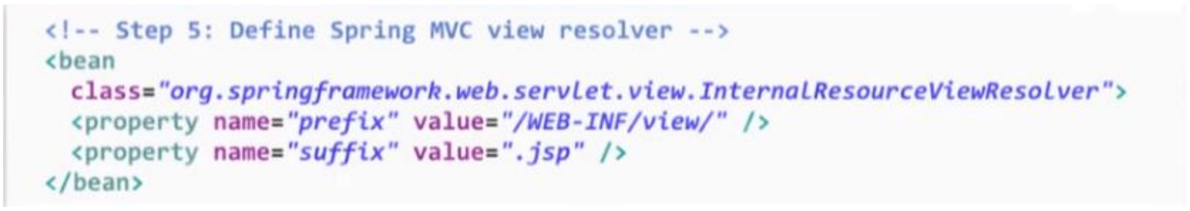
When we are using Spring MVC then it can perform formatting of form data and can also perform form validation and in order to get the support we need to add this into our configuration xml file.



```
File: spring-mvc-demo-servlet.xml
<beans>

  <!-- Step 3: Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

  <!-- Step 4: Add support for conversion, formatting and validation support -->
  <mvc:annotation-driven/>

</beans>
```

- Finally Configure the Spring MVC View Resolver.

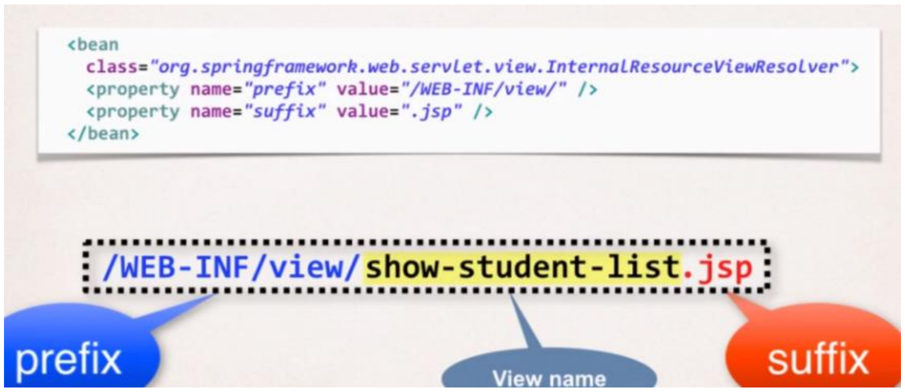
In the final step, we will tell the spring where to look for files to actually render the view for our application.



```xml
<!-- Step 5: Define Spring MVC view resolver -->
<bean
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/view/" />
  <property name="suffix" value=".jsp" />
</bean>
```

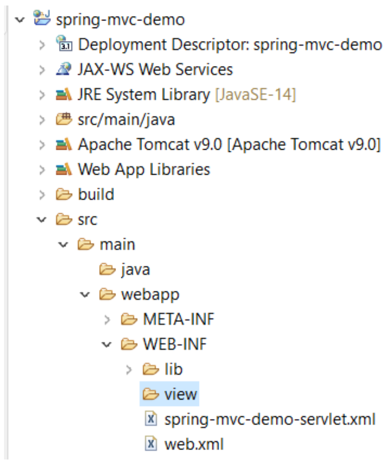
When our app provides a “view” name, Spring MVC will

- ✓ Prepend the prefix
- ✓ Append the suffix to that view name automatically.



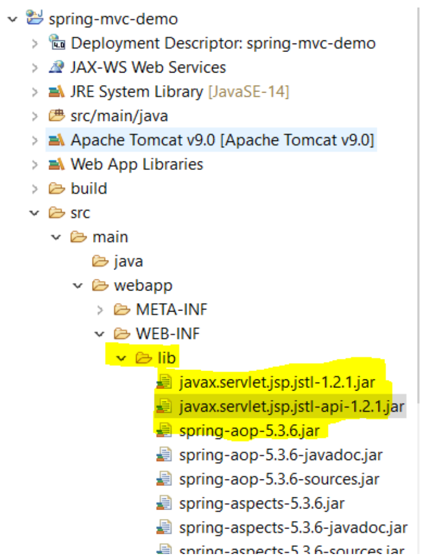
```xml
<bean
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/view/" />
  <property name="suffix" value=".jsp" />
</bean>
```

And we need to create this folder i.e., ***/WEB_INF/view/*** inside our project in Eclipse.



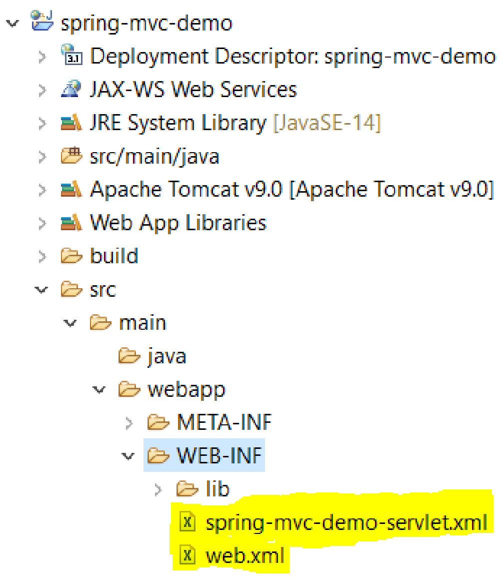
```
spring-mvc-demo
  Deployment Descriptor: spring-mvc-demo
  JAX-WS Web Services
  JRE System Library [JavaSE-14]
  src/main/java
  Apache Tomcat v9.0 [Apache Tomcat v9.0]
  Web App Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
          view
          spring-mvc-demo-servlet.xml
          web.xml
```

- During the Development on Eclipse, add the Spring framework jar file as well as JSTL jar files and it will automatically add those jars into our classpath of this project.


spring-mvc-demo
Deployment Descriptor: spring-mvc-demo
JAX-WS Web Services
JRE System Library [JavaSE-14]
src/main/java
Apache Tomcat v9.0 [Apache Tomcat v9.0]
Web App Libraries
build
src
main
java
webapp
META-INF
WEB-INF
lib
javax.servlet.jsp.jstl-1.2.1.jar
javax.servlet.jsp.jstl-api-1.2.1.jar
spring-aop-5.3.6.jar
spring-aop-5.3.6-javadoc.jar
spring-aop-5.3.6-sources.jar
spring-aspects-5.3.6.jar
spring-aspects-5.3.6-javadoc.jar
spring-aspects-5.3.6-sources.jar

- Now we are going to add two starter files inside the WEB-INF directory in our project.

  Here spring-mvc-demo-servlet.xml is just our configuration file where we usually mention the Component scanning packages.


spring-mvc-demo
Deployment Descriptor: spring-mvc-demo
JAX-WS Web Services
JRE System Library [JavaSE-14]
src/main/java
Apache Tomcat v9.0 [Apache Tomcat v9.0]
Web App Libraries
build
src
main
java
webapp
META-INF
WEB-INF
lib
spring-mvc-demo-servlet.xml
web.xml

Section 12: Spring MVC - Creating Controllers and Views

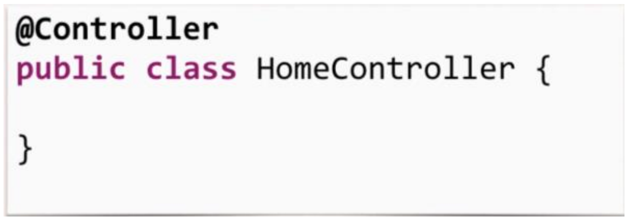
1 / 15 | 46min

- **Creating a Spring Home Controller and View:**

  <u>Development Process:</u>

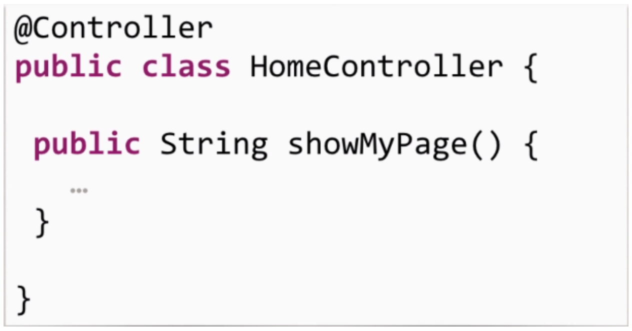
  - Create a Controller class

    Annotate class with ***@Controller***. And @Controller says that it's a Spring MVC controller and ***@Controller inherits from @Component,*** and Spring does the scanning for @Component then it does pick @Controller as they extend from @Component.



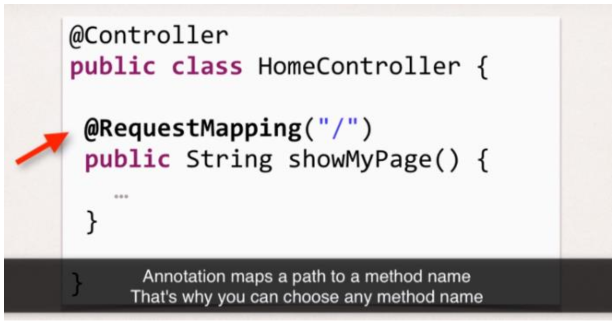
```
@Controller
public class HomeController {

}
```

- Define a controller method.



```
@Controller
public class HomeController {

 public String showMyPage() {
   ...
 }

}
```

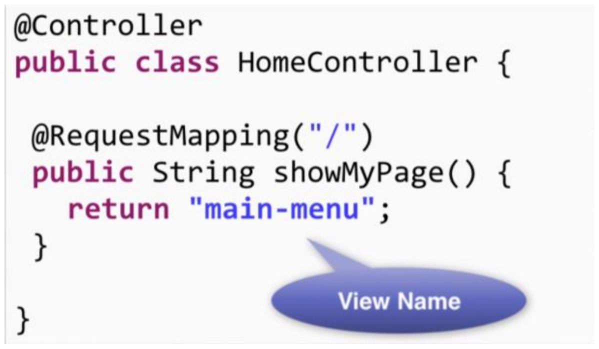
- Add the Request Mapping to the Controller method.



```
@Controller
public class HomeController {

 @RequestMapping("/")
 public String showMyPage() {
   ...
 }

}
```

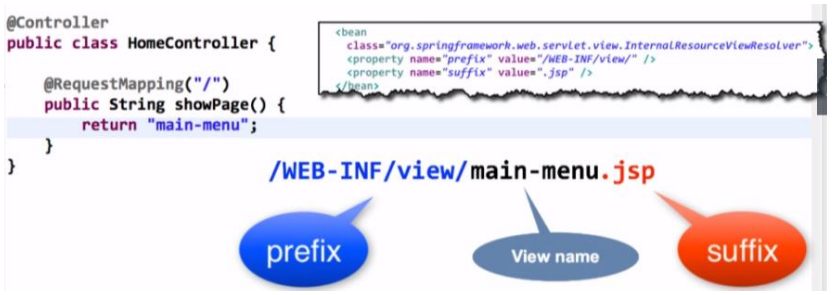
Annotation maps a path to a method name
That's why you can choose any method name

- Return View Name



```
@Controller
public class HomeController {

 @RequestMapping("/")
 public String showMyPage() {
   return "main-menu";
 }

}
```

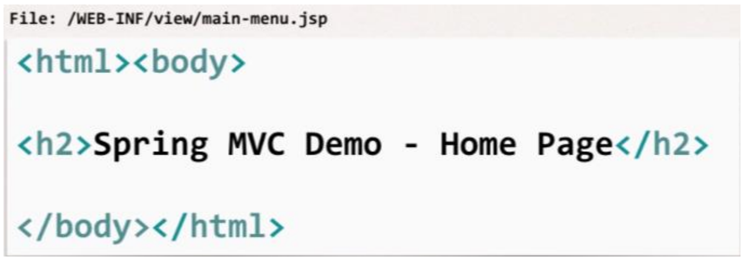
Remember There is some magic that's going to happen in the background; Spring is going to take the information from it's configuration file and it will actually find the view page. So based on the config it is going to look into the prefix directory and it will use the view name and then it will append the suffix ***(i.e., ".jsp" in our case)***



```
@Controller
public class HomeController {

    @RequestMapping("/")
    public String showPage() {
        return "main-menu";
    }
}
```

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

- Develop the View Page.

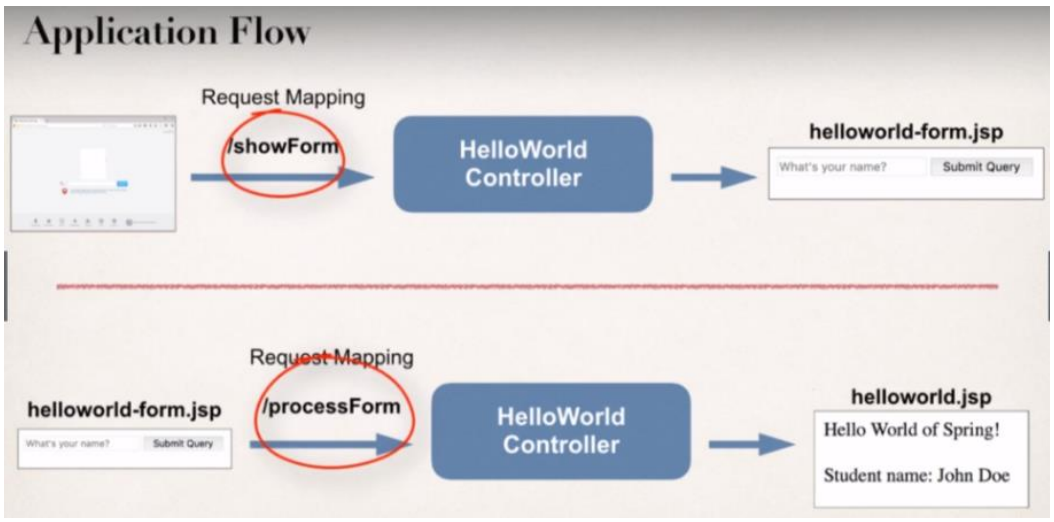


File: /WEB-INF/view/main-menu.jsp

```
<html><body>

<h2>Spring MVC Demo - Home Page</h2>

</body></html>
```

➢ **<u>How to Read form Data with Spring MVC:</u>**

Here We will have only one Controller with two Request Mappings, where user will first ask for the form and then submit the data and later controller will process it and then show it on the JSP page.



Development Process:

1. Create Controller Class.

2. **Show HTML form**

2.1. Create controller method to show HTML form.

```
 //need a controller method to show the initial HTML form
@RequestMapping("/showForm")
public String showForm() {
   return "helloworld-form";

 }

//need a controller method to process the HTML form
@RequestMapping("/processForm")
 public String processForm() {
          return "helloWorld";
 }
```

2.2. Create View Page for HTML form.

File: helloworld-form.jsp

```
<form action="processForm" method="GET">
       <input type="text" name="studentName" placeholder="What's your name?"/>
       <input type="submit"/>
</form>
```

3. Process HTML Form

3.1. Create Controller method to process HTML Form.
3.2. Develop View Page for Confirmation

File: helloWorld.jsp:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
   <head>
     <meta charset="ISO-8859-1">
     <title>Insert title here</title>
   </head>
   <body>

     <h3>Hello world of Spring!</h3>

     Student Name: ${param.studentName}
   </body>
</html>
```

➢ **<u>Adding Data to Spring Model</u>**

So, the Spring model is just a container to your application data.

In our Controller we can put any type of data
Like String, objects, info from the Database, etc. and our View page (JSP) can access data from the **model**.

Passing Model to your Container:

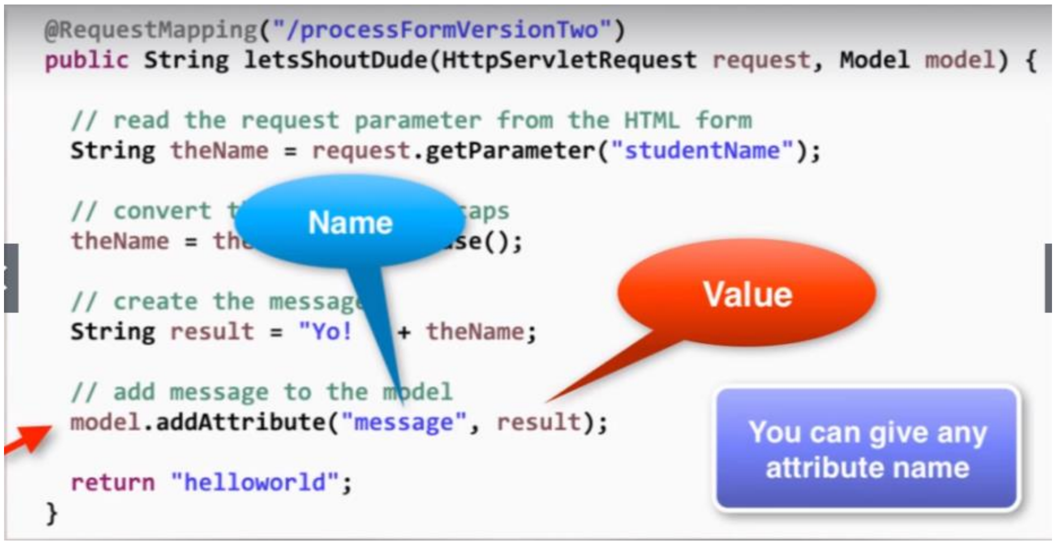
Code Example-

1. We want to create a new method to process form data
2. Read the form data: Student's name
3. Convert the name to upper case
4. Add the uppercase version to the model.

In Spring MVC, when we create our controller method and we need to read our form data in controller then we pass any Http server request and it works like any Servlet Request and also, we can pass our model.

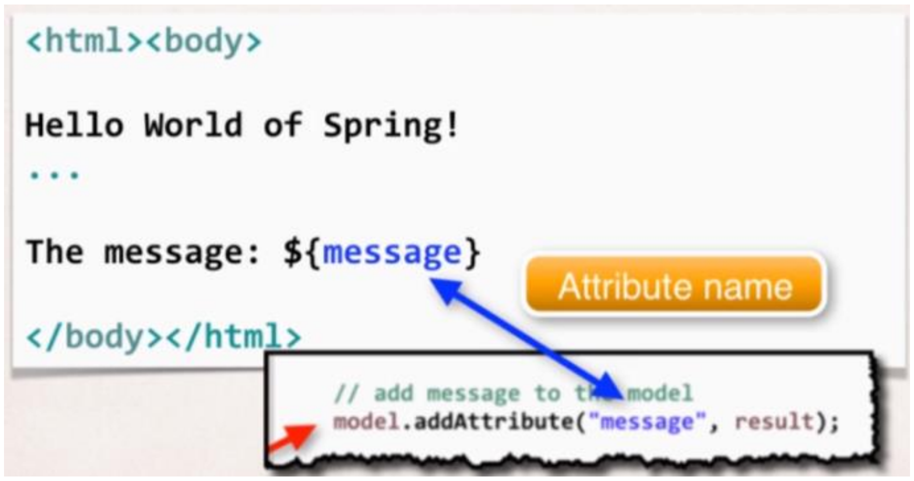
When we need to add something to the model then we call

model.addAttribute() and in the parameter we give model attribute name and then it's value (that's simply the name and value pair inside the addAttribute() method).



```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

  // read the request parameter from the HTML form
  String theName = request.getParameter("studentName");

  // convert t                  aps
  theName = the            se();

  // create the messag
  String result = "Yo!     + theName;

  // add message to the model
  model.addAttribute("message", result);

  return "helloworld";
}
```

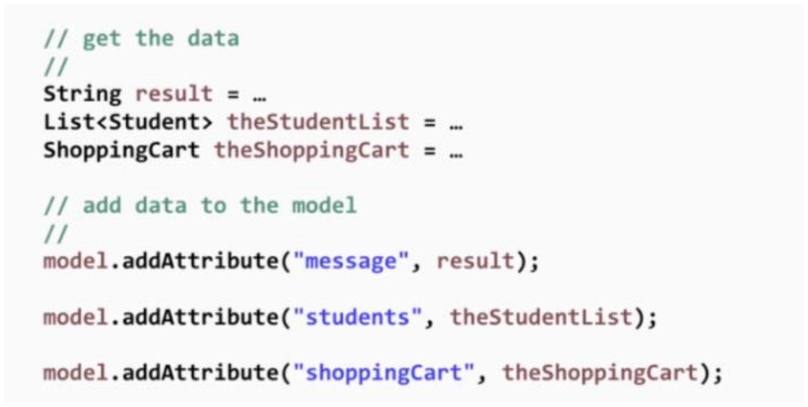
And on the View Page (JSP) we can access the data from the model using the attribute name that has been set in the controller class.



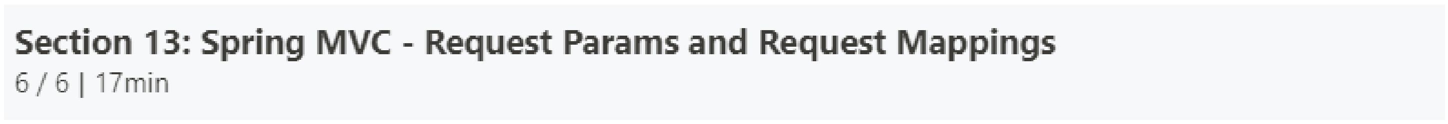
```
<html><body>

Hello World of Spring!
...

The message: ${message}

</body></html>
```

```
// add message to the model
model.addAttribute("message", result);
```

And we can add any amount of data to the model, here is the example shown below:



```
// get the data
//
String result = …
List<Student> theStudentList = …
ShoppingCart theShoppingCart = …

// add data to the model
//
model.addAttribute("message", result);

model.addAttribute("students", theStudentList);

model.addAttribute("shoppingCart", theShoppingCart);
```



Section 13: Spring MVC - Request Params and Request Mappings

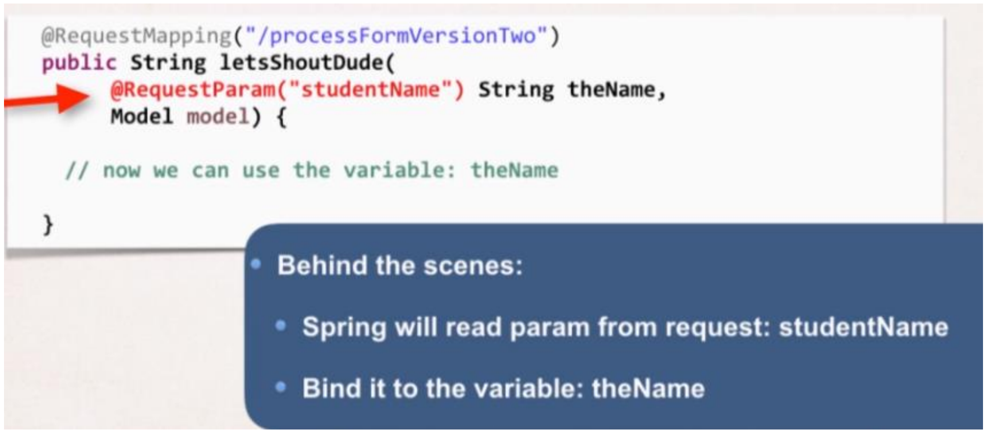
6 / 6 | 17min

➢ Reading HTML Form Data with *@RequestParam* Annotation

Earlier we were getting the form data using ***HttpServletRequest*** and then we're using *getParameter("studentName")* but now we will read the form data i.e., "studentName" using ***@RequestParam.***

Spring has this special annotation which will read the form data from request Parameters and then it will take that data and it will bind with the variable like here we have ***theName*** in our example.

And then we are good to perform rest of the operations on the form data which is inside the ***theName*** variable.



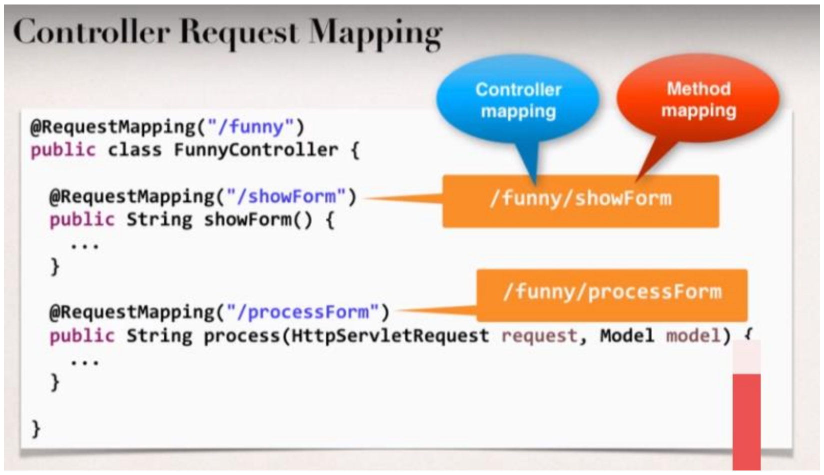
```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(
        @RequestParam("studentName") String theName,
        Model model) {

    // now we can use the variable: theName

}
```

- Behind the scenes:
  - Spring will read param from request: studentName
  - Bind it to the variable: theName

- **Controller Level Request Mapping:**

Adding Request Mappings to controller:

- Serves as parent Mapping for Controller.
- All request Mappings on methods in the controller are relative.

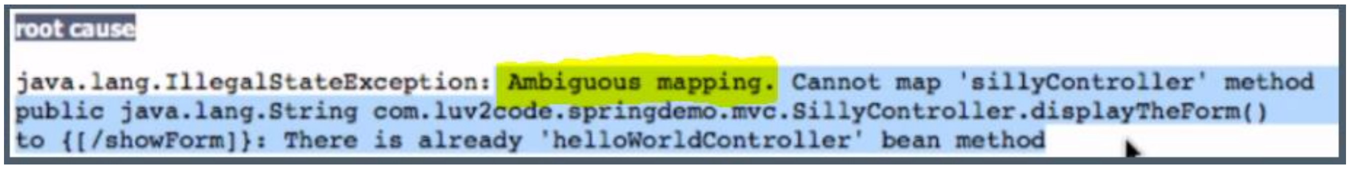


Controller Request Mapping

```
@RequestMapping("/funny")
public class FunnyController {

  @RequestMapping("/showForm")
  public String showForm() {
    ...
  }

  @RequestMapping("/processForm")
  public String process(HttpServletRequest request, Model model) {
    ...
  }
}
```

<u>Here we will see the conflict that we can get when we have two request mappings with similar name.</u>

This is the **error** we will get when we have the above-mentioned scenario;



```
root cause

java.lang.IllegalStateException: Ambiguous mapping. Cannot map 'sillyController' method
public java.lang.String com.luv2code.springdemo.mvc.SillyController.displayTheForm()
to {[/showForm]}: There is already 'helloWorldController' bean method
```

FAQ: How does "processForm" work for "/hello"?

Question: Can you please clarify how /hello is getting appended to the jsp file action for "processForm"?

Answer

You can use "**processForm**" because it is a relative path to the controller "/hello" request mapping. Here is how it works.

1. When you wish to view the form, the HTML link points to "hello/showForm". This calls the controller and it displays the form.

2. At this point the browser URL/path is: *http://localhost:8080/spring-mvc-demo/hello*

3. The HTML form uses "processForm" for the form action. Notice that it does not have a forward slash, as a result, this will be relative to the current browser URL. Since the current browser URL is

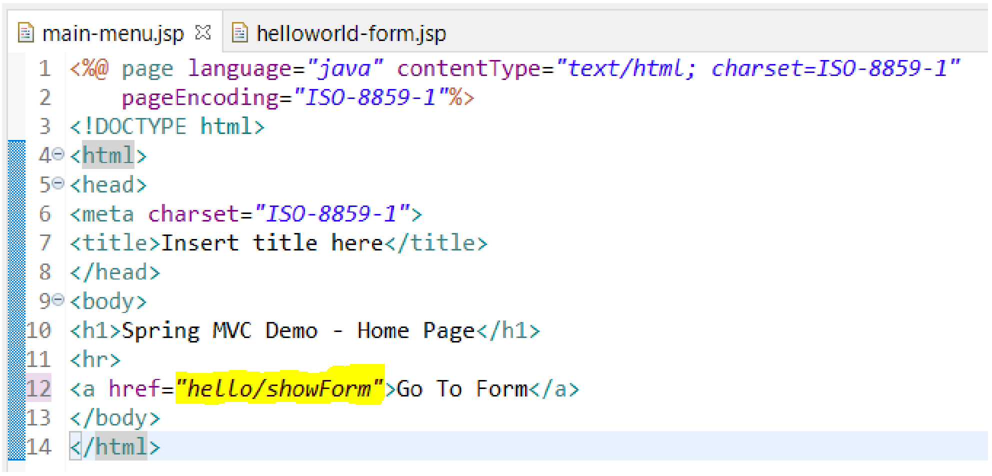
http://localhost:8080/spring-mvc-demo/hello

Then the actual form URL submission will send it to

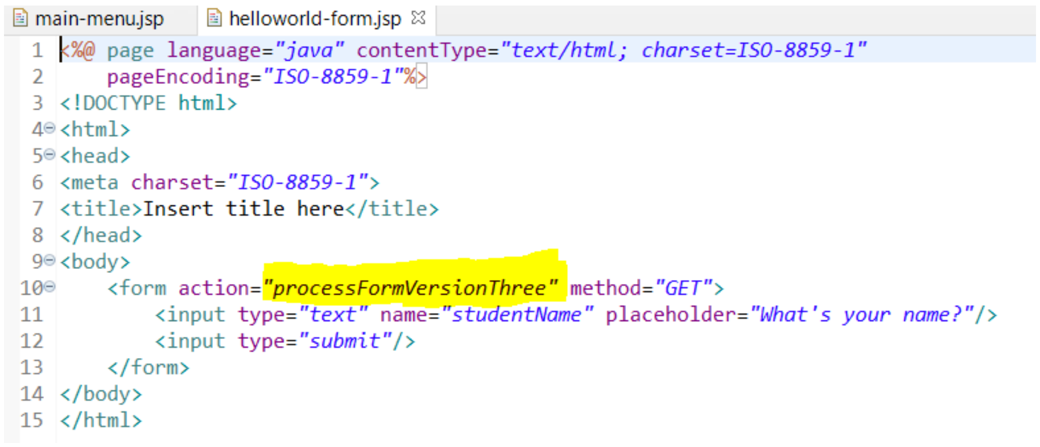
http://localhost:8080/spring-mvc-demo/hello/processForm

The part in bold with map to the controller with top-level request mapping "/hello" and then map to request mapping in that class "/processForm"

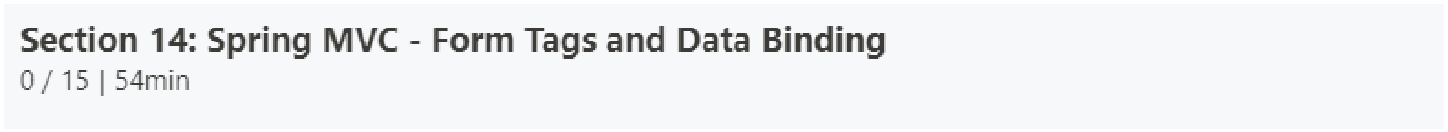
The key here is relative path of showing the form and then submitting to relative path.



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<h1>Spring MVC Demo - Home Page</h1>
<hr>
<a href="hello/showForm">Go To Form</a>
</body>
</html>
```



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <form action="processFormVersionThree" method="GET">
        <input type="text" name="studentName" placeholder="What's your name?"/>
        <input type="submit"/>
    </form>
</body>
</html>
```



Section 14: Spring MVC - Form Tags and Data Binding

0 / 15 | 54min

➢ **Spring MVC Form Tags Overview**

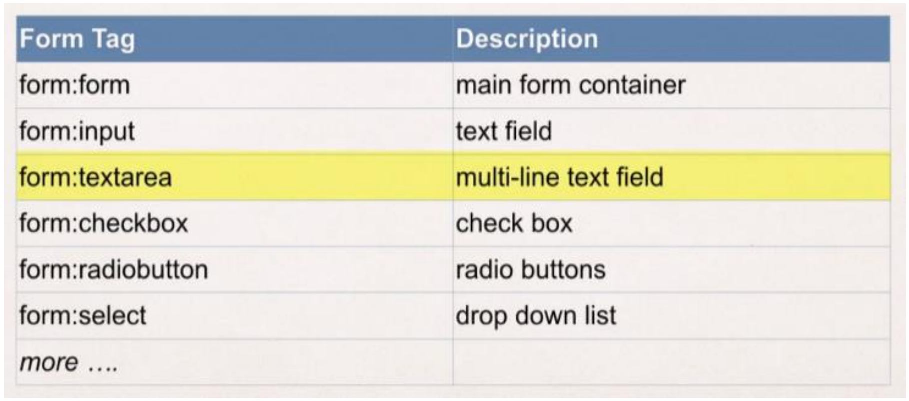
Spring MVC has support for Form tags and these form tags are configurable and we can reuse them for a web page.

Data Binding

Spring MVC Form Tags support data binding.

It allows us to automatically set data and retrieve data from a Java objects and beans.

- Spring MVC Form tags:



Form Tag	Description
form:form	main form container
form:input	text field
form:textarea	multi-line text field
form:checkbox	check box
form:radiobutton	radio buttons
form:select	drop down list
more	

We'll make use of normal JSP page and then we can simply drop the Spring MVC Form tags.

How to reference Spring MVC Form Tags?

Specify the Spring namespace at the beginning of the JSP file.

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

Once that's get setup, we can make use of Spring MVC form tag in our JSP page.

➢ **Spring MVC Form Tag for TextField:**

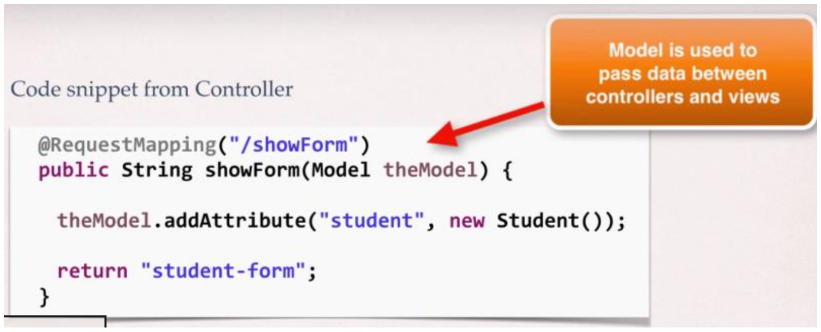
In our Spring Controller:

- We must add ***model Attribute.***
- This is a bean that will actually hold form data and this will give us the support for data binding.

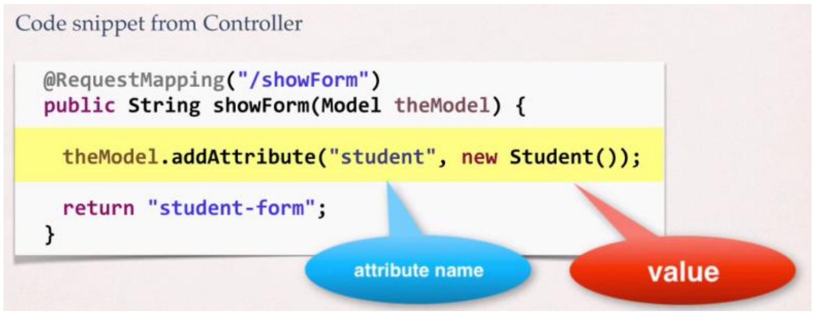
So, when we show the form, we need to add the model attribute and here we have Model parameter and model will pass the data between controllers and views.

In this example, we are giving the *"model name"* as *"student"* and then as value we're giving the object of student type (empty object), that will get pass to the form so that form make up with Data Binding.

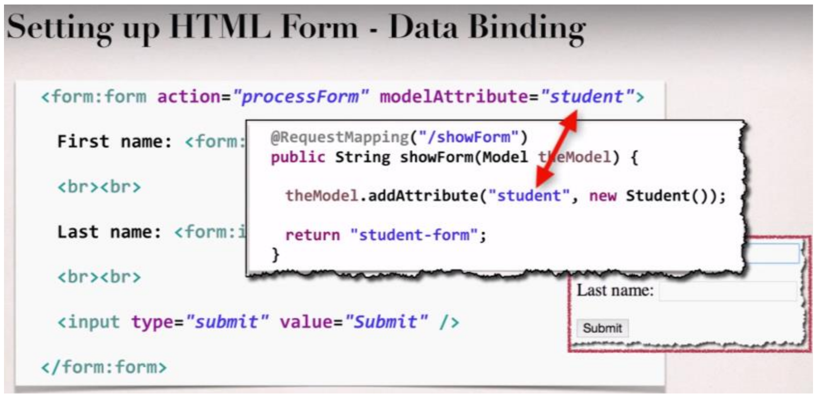
And our Form should use the name as given in model name i.e., *student* to reference the student Object.


Code snippet from Controller
Model is used to pass data between controllers and views

```
@RequestMapping("/showForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```



```
@RequestMapping("/showForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```



```
<form:form action="processForm" modelAttribute="student">

  First name: <form:

  <br><br>

  Last name: <form:i

  <br><br>

  <input type="submit" value="Submit" />

</form:form>
```

```
@RequestMapping("/showForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```

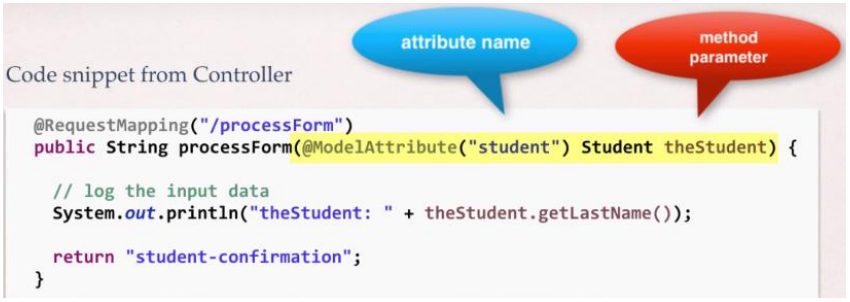
```
First Name: <form:input path="firstName"/>
<br><br>
Last Name: <form:input path="lastName"/>
```

The Highlighted part is actually binding this from field with the property of the bean. When is form is first loaded what Spring MVC will do behind the scene is it will actually populate the form Fields using getter methods of the Object property and if it returns null then the from will remain empty.

And when form get submitted Spring will call setter methods and it will use whatever data entered by the user into the form fields.

Handling Form Submission in the Controller:

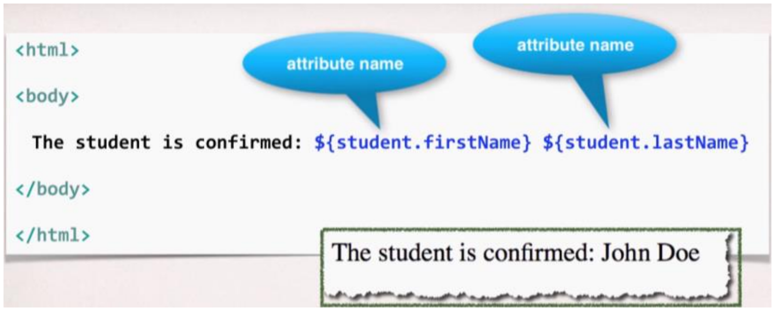
We will make use of new Spring annotation called *@ModelAttribute*.



```
@RequestMapping("/processForm")
public String processForm(@ModelAttribute("student") Student theStudent) {

  // log the input data
  System.out.println("theStudent: " + theStudent.getLastName());

  return "student-confirmation";
}
```

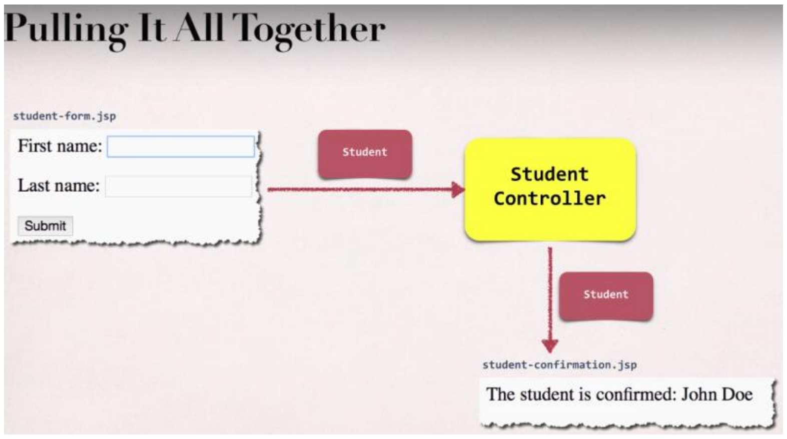
Now that annotation will actually get the object using the Attribute name and pass it to the parameter ***Student theStudent*** and then we can use it in our controller method.

So, we don't need to do manually ***request.getParameter*** for each Form fields and Spring will handle all that for us.

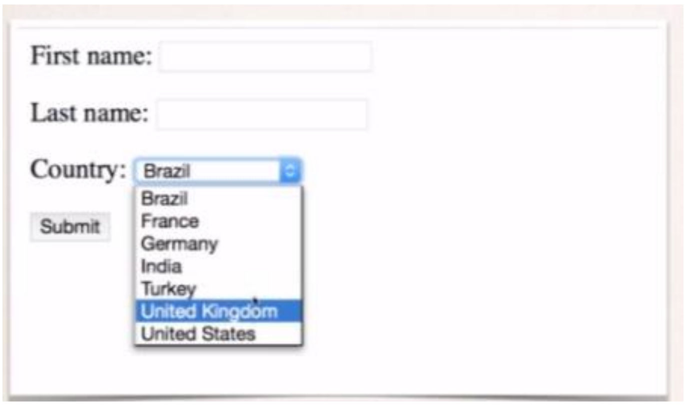


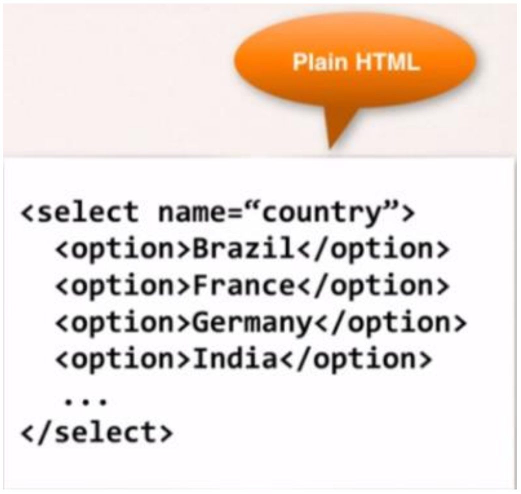
```
<html>

<body>

 The student is confirmed: ${student.firstName} ${student.lastName}

</body>

</html>
```

The student is confirmed: John Doe



➢ Spring MVC Form Tag – Drop Down List





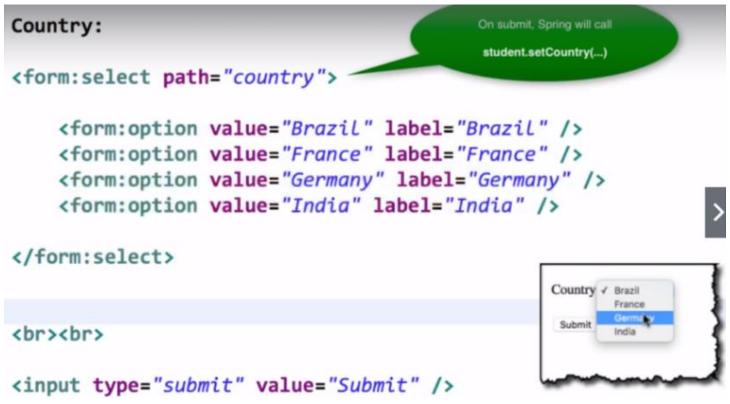
```
<select name="country">
  <option>Brazil</option>
  <option>France</option>
  <option>Germany</option>
  <option>India</option>
   ...
</select>
```

In the Spring MVC, Drop down list is represented by `<form:select>`

In `<form:select>`, we basically specify the path to the property of the Object and then we give the options in which value is the actual value that will get set using setter method of the property mentioned in the path attribute whereas label will be displayed on the view only.

Example:

```
Country:
        <form:select path="country">
                <form:option value="Brazil" label="Brazil"/>
                <form:option value="France" label="France"/>
                <form:option value="Germany" label="Germany"/>
                <form:option value="India" label="India"/>
        </form:select>
```



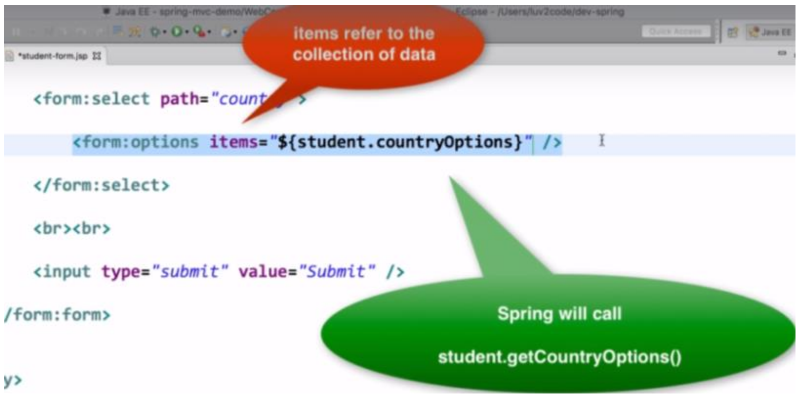
```
Country:
<form:select path="country">

    <form:option value="Brazil" label="Brazil" />
    <form:option value="France" label="France" />
    <form:option value="Germany" label="Germany" />
    <form:option value="India" label="India" />

</form:select>

<br><br>

<input type="submit" value="Submit" />
```

<u>Read the List of Countries from a Java class</u>

1. Define countryOptions inside the class.

```
public class Student {

  String firstName;
  String lastName;
  String country;
  String favoriteLanguage;
  private LinkedHashMap<String, String> countryOptions;

  public Student() {
        //populate country options: used ISO country code
        countryOptions = new LinkedHashMap<String, String>();
        countryOptions.put("BR", "Brazil");
        //"BR" is key/code and "Brazil" is value/label
        countryOptions.put("FR", "France");
        countryOptions.put("DE", "Germany");
        countryOptions.put("IN", "India");
        countryOptions.put("US", "USA");
  }
.
.    //getter and setters
.
.

  public LinkedHashMap<String, String> getCountryOptions() {
        return countryOptions;
  }
}
```

Here we will need only the getter method for `countryOptions`, as when form is loaded, Spring will call ***student.getCountryOptions()*** method, to load the drop down items into the list.


Java EE - spring-mvc-demo/WebC... Eclipse - /Users/luv2code/dev-spring
*student-form.jsp
items refer to the collection of data
Spring will call student.getCountryOptions()

```
<form:select path="count  >

    <form:options items="${student.countryOptions}" />

</form:select>

<br><br>

<input type="submit" value="Submit" />

/form:form>

y>
```

So, this way we're reading all the data form the Java class and we are not hardcoding it in the JSP page, and that java class can read the data from the property file or from Database or external services.

FAQ: Use properties file to load country options:

Question:
How to use properties file to load country options

Answer:
This solution will show you how to place the country options in a properties file. The values will no longer be hard coded in the Java code.

1. **Create a properties file to hold the countries. It will be a name value pair. Country code is name. Country name is the value.**

   New text file: WEB-INF/countries.properties

```
BR=Brazil
FR=France
CO=Colombia
IN=India
```

   Note the location of the properties file is very important. It must be stored in

   WEB-INF/countries.properties

2. Update header section for Spring config file
We are going to use a new set of Spring tags for <util>. As a result, you need to update the header information in the Spring config file.

File: spring-mvc-dmo-servlet.xml

Remove the previous header and add this.

```
        <?xml version="1.0" encoding="UTF-8"?>
<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:util="http://www.springframework.org/schema/util"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context.xsd
            http://www.springframework.org/schema/mvc
            http://www.springframework.org/schema/mvc/spring-mvc.xsd
            http://www.springframework.org/schema/util
            http://www.springframework.org/schema/util/spring-util.xsd">
```

3. Load the country options properties file in the Spring config file. Bean id: countryOptions
File: spring-mvc-demo-servlet.xml.

Add the following lines:

```
        <util:properties  id="countryOptions"
location="classpath:../countries.properties" />
```

4.1 In StudentController.java, add the following import statement:

```
import java.util.Map;
```

4.2 Inject the properties values into your Spring Controller: StudentController.java

```java
@Value("#{countryOptions}")
private Map<String, String> countryOptions;
```

5. Add the country options to the Spring MVC model. Attribute name: theCountryOptions

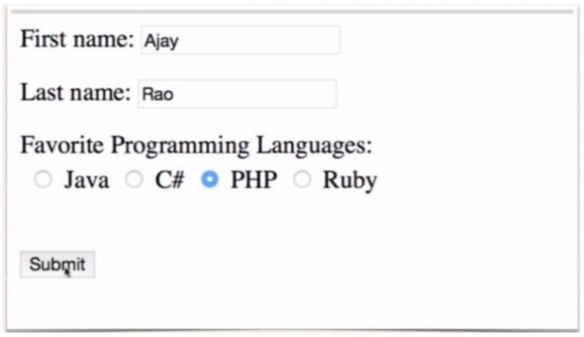
```java
@RequestMapping("/showForm")
public String showForm(Model theModel) {

    // create a student object Student
    Student theStudent = new Student();

    // add student object to the model
    theModel.addAttribute("student", theStudent);

    // add the country options to the model
    theModel.addAttribute("theCountryOptions", countryOptions);

    return "student-form";
}
```

6. Update the JSP page, student-form.jsp, to use the new model attribute for the drop-down list: theCountryOptions

```jsp
<form:select path="country">
 <form:options items="${theCountryOptions}" />
</form:select>
```

7. Remove all references to country option from your Student.java.

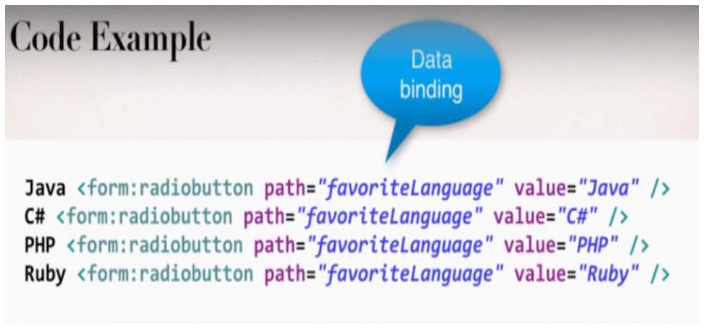
➢ **Radio Buttons:**



First name:	Ajay
Last name:	Rao

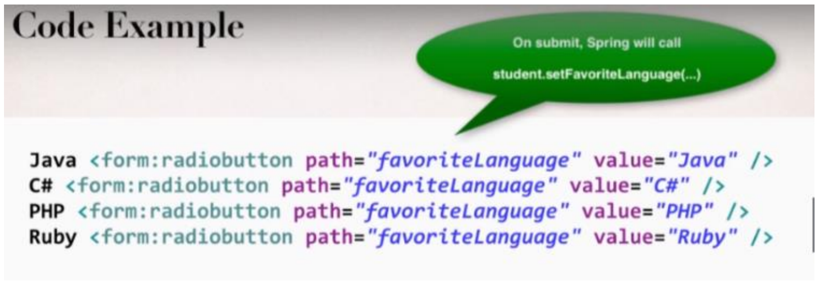
Favorite Programming Languages:

Java ○ C# ○ [✓ PHP] ○ Ruby ○



```jsp
Java <form:radiobutton path="favoriteLanguage" value="Java" />
C# <form:radiobutton path="favoriteLanguage" value="C#" />
PHP <form:radiobutton path="favoriteLanguage" value="PHP" />
Ruby <form:radiobutton path="favoriteLanguage" value="Ruby" />
```

Spring MVC Tag for A Radio Button is represented by the tag *<form:radiobutton>*. And we need to have the setter methods in our class for the above mentioned property


Code Example
On submit, Spring will call student.setFavoriteLanguage(...)

```jsp
Java <form:radiobutton path="favoriteLanguage" value="Java" />
C# <form:radiobutton path="favoriteLanguage" value="C#" />
PHP <form:radiobutton path="favoriteLanguage" value="PHP" />
Ruby <form:radiobutton path="favoriteLanguage" value="Ruby" />
```

Inside the Student Java class add the property *favoriteLanguage*.

```
public class Student {

   String firstName;
   String lastName;
   String country;
   String favoriteLanguage;
   .
   . //getters and setters
   .
   .
   public void setFavoriteLanguage(String favouriteLanguage) {
       this.favoriteLanguage = favouriteLanguage;
   }

   public String getFavoriteLanguage() {
       return favoriteLanguage;
   }
}
```

And update the confirmation page.

FAQ: How to populate radiobuttons with items from Java class?

FAQ: How to populate radiobuttons with items from Java class like we did with selectlist?
You can follow a similar approach that we used for the drop-down list.

Here are the steps

1. Set up the data in your Student class

Add a new field

```
private LinkedHashMap<String, String> favoriteLanguageOptions;
```

In your constructor, populate the data

```
        // populate favorite language options
        favoriteLanguageOptions = new LinkedHashMap<>();
        // parameter order: value, display label
        //
        favoriteLanguageOptions.put("Java", "Java");
        favoriteLanguageOptions.put("C#", "C#");
        favoriteLanguageOptions.put("PHP", "PHP");
        favoriteLanguageOptions.put("Ruby", "Ruby");
```

Add getter method

```
public LinkedHashMap<String, String> getFavoriteLanguageOptions() {
        return favoriteLanguageOptions;
}
```

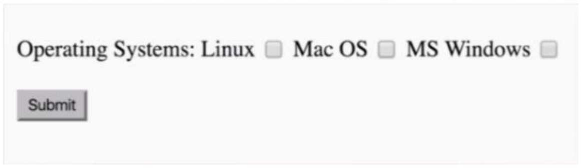
2. Reference the data in your form

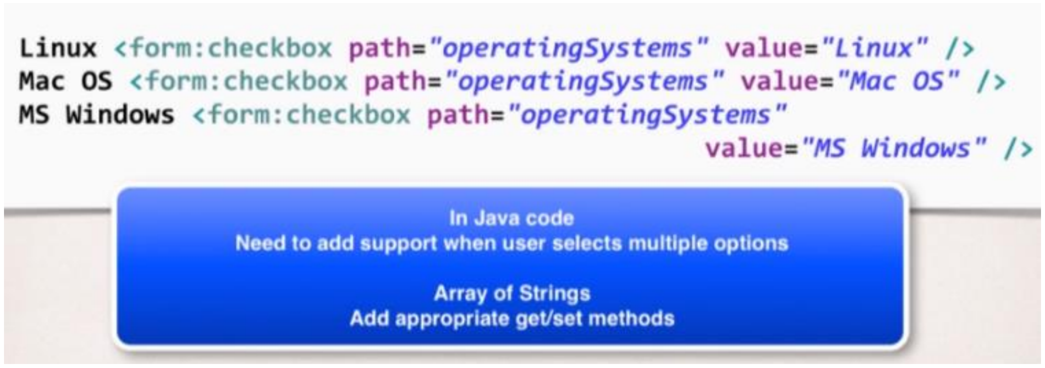
Favorite Language:

```
        <form:radiobuttons path="favoriteLanguage"
items="${student.favoriteLanguageOptions}"  />
```

- **Spring MVC form tags for Checkboxes:**

A check box is represented by the tag *`<form:checkbox>`*





```
Linux <form:checkbox path="operatingSystems" value="Linux" />
Mac OS <form:checkbox path="operatingSystems" value="Mac OS" />
MS Windows <form:checkbox path="operatingSystems"
                                value="MS Windows" />
```

In Java code
Need to add support when user selects multiple options

Array of Strings
Add appropriate get/set methods

- Will update our ***`student-form.jsp`***:

```
Operating Systems:

Linux <form:checkbox path="operatingSystems" value="Linux"/>
Windows <form:checkbox path="operatingSystems" value="Windows"/>
MAC OS <form:checkbox path="operatingSystems" value="MAC OS"/>

<br><br>
```

- Now update the ***`Student.java`***

```
public class Student {

      String firstName;
      String lastName;
      String country;
      String favoriteLanguage;
```

```
    ArrayList<String> operatingSystems;
    private LinkedHashMap<String, String> countryOptions;
    .
    .
    .
    . //getters and setters and constructors
    .
    .

    public ArrayList<String> getOperatingSystems() {
            return operatingSystems;
    }

    public void setOperatingSystems(ArrayList<String> operatingSystems) {
            this.operatingSystems = operatingSystems;
    }
}
```

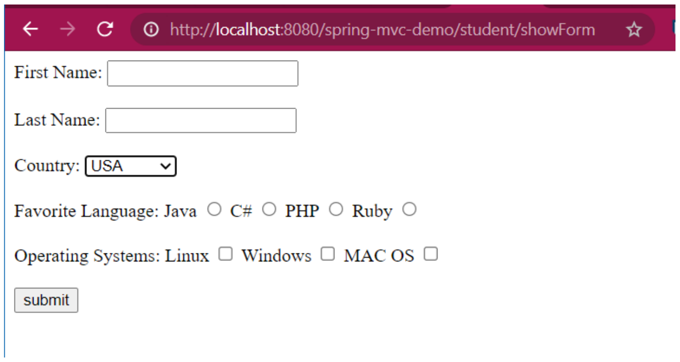
- Now we need to update the ***student-confirmation.jsp*** page

  Add the ***JSTL core tags*** which will allow us to loop over the collections.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

```
    <br><br>

    Operating Systems User used:

    <ul>
            <c:forEach var="temp" items="${student.operatingSystems}">

            <li>${temp}</li>

            </c:forEach>
    </ul>
```

"items:" will call the getter methods for ***operatingSystems*** property of the model object and it will display all the selected operating systems in the bullet points structure.


http://localhost:8080/spring-mvc-demo/student/showForm

First Name:

Last Name:

Country: USA

Favorite Language: Java ○ C# ○ PHP ○ Ruby ○

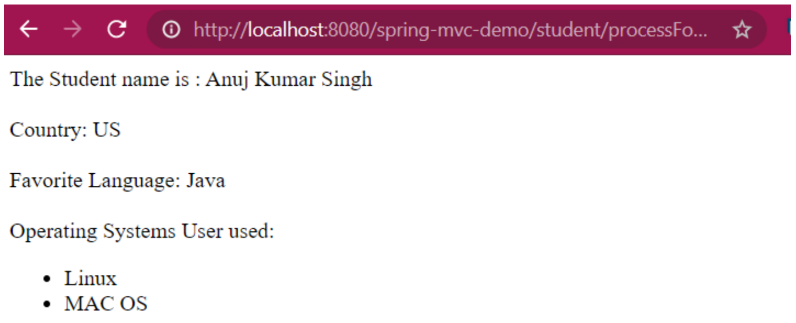
Operating Systems: Linux ☐ Windows ☐ MAC OS ☐

submit

We will get to look our student-from.jsp page on hitting the url:

http://localhost:8080/spring-mvc-demo/student/showForm

And the *student-confirmation.jsp* will look like, (after processing the form)



The Student name is : Anuj Kumar Singh

Country: US

Favorite Language: Java

Operating Systems User used:

- Linux
- MAC OS

Section 15: Spring MVC Form Validation - Applying Built-In Validation Rules

13 / 13 | 51min

- **Spring Form Validation**

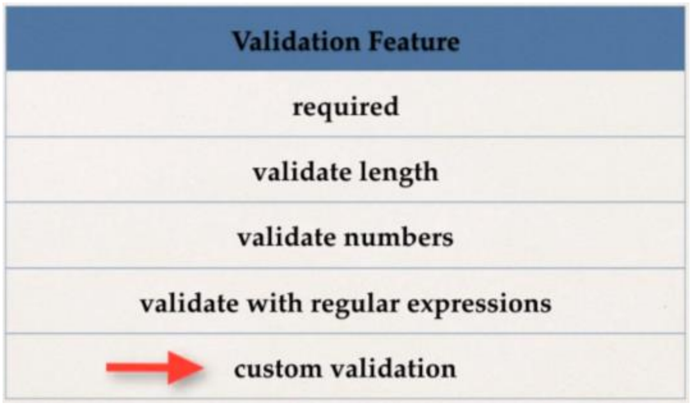
We can use the Java's standard bean validation API.
It defines a metadata model and API for entity validation.
And this bean validation API is available for service side applications and also client-side applications.

For more information: http://www.beanvalidation.org

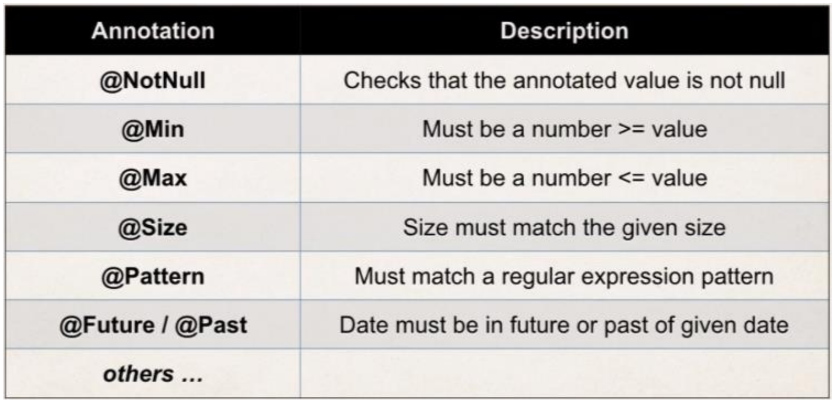
Spring and Validation

Spring version 4 or higher supports the Bean Validation API.
And it's actually the preferred method for validation when building Spring apps.
We will simply add the Validation JARs to our project and use specified annotations and we are ready to go.

Bean validation Features



Validation Feature
required
validate length
validate numbers
validate with regular expressions
custom validation



Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number >= value
@Max	Must be a number <= value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others ...	

Our Road Map

1. Set up our development Environment
2. Require field
3. Validate number range: min, max
4. Validate using regular expression (regexp)
5. Custom validation

- **Setting up the development Environment:**

  Java's standard Bean validation API is only a specification, it's vendor independent and portable however, we still need an implementation.

  So, this is where Hibernate team comes to the rescue! And they have a separate project for validation. Hibernate Validator is fully complaint with Java's standard bean validation API.

  Website: ***http://www.hibernate.org/validator***

  About the versions!
  1. Hibernate Validator 7 is based on Jakarta EE 9.
  2. Jakarta EE is the community version of Java EE (rebranded, relicensed)
  3. And Java EE is collection of enterprise API like servlets, JSP, JDBC, Enterprise java beans, Java Message Service, etc.
  4. Allows innovation of Jakarta EE with community driven approach.
  5. Jakarta EE does not replace Java EE.

  Jakarta EE

  1. At the moment main change with **Jakarta EE** is package renaming
  2. ***javax.**** packages are renamed to ***jakarta.****

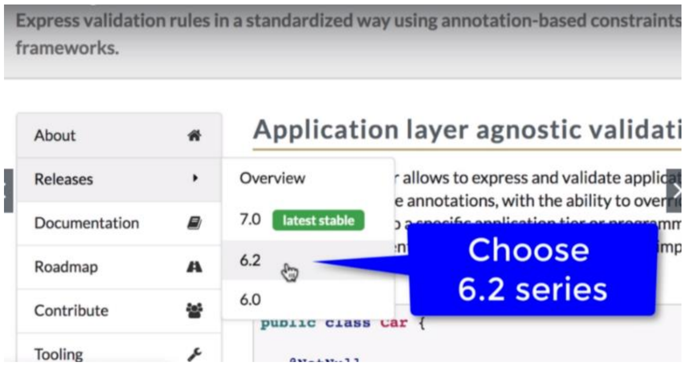
What impact on Hibernate Validator?

- Hibernate Validator 7 is based on **Jakarta EE 9**
- Spring 5 is still based on some components of **Java EE (*javax.**).**
- Spring may use Jakarta EE components in the future but no news yet.

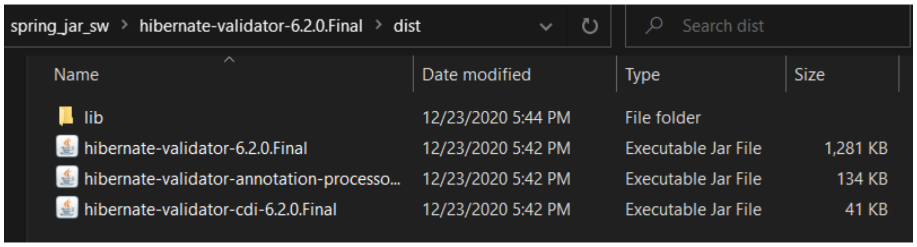
- As a result, Spring 5 is not compatible with Hibernate validator 7

✓ **Hibernate Validator 6.2** is compatible with Spring 5.
✓ **Hibernate Validator 6.2** has the same features as Hibernate Validator 7.

Download the Hibernate Validator 6.2 jar zip archive file from www.hibernate.org by going into the Validator section


Express validation rules in a standardized way using annotation-based constraints
frameworks.
About
Releases
Documentation
Roadmap
Contribute
Tooling
Overview
7.0 latest stable
6.2
6.0
Application layer agnostic validati
Choose 6.2 series

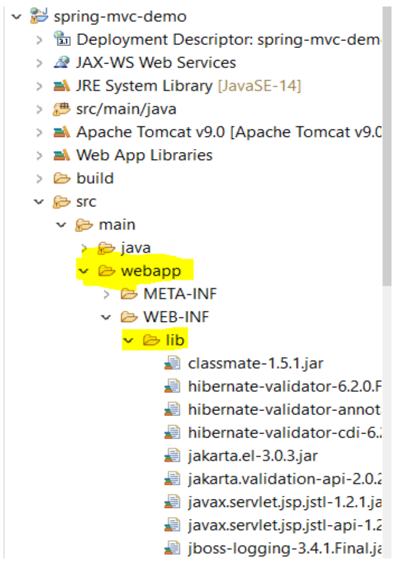
✓ On opening the zip archive file, go to **dist** directory and then copy all the three jar files present in that directory and then paste the jar files in the lib directory in the webapps folder of our project.



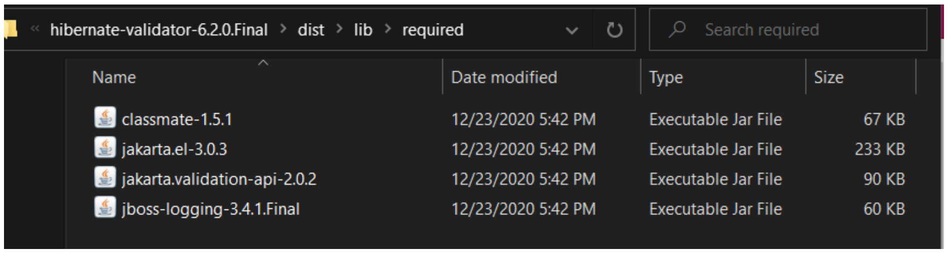
spring_jar_sw > hibernate-validator-6.2.0.Final > dist

Name	Date modified	Type	Size
lib	12/23/2020 5:44 PM	File folder	
hibernate-validator-6.2.0.Final	12/23/2020 5:42 PM	Executable Jar File	1,281 KB
hibernate-validator-annotation-processo...	12/23/2020 5:42 PM	Executable Jar File	134 KB
hibernate-validator-cdi-6.2.0.Final	12/23/2020 5:42 PM	Executable Jar File	41 KB

✓ And we will need some other required files as well which is present in ***hibernate-validator-6.2.0.Final\dist\lib\required*** and then copy them and paste into the webapps directory of our project.



```
spring-mvc-demo
  Deployment Descriptor: spring-mvc-dem
  JAX-WS Web Services
  JRE System Library [JavaSE-14]
  src/main/java
  Apache Tomcat v9.0 [Apache Tomcat v9.0
  Web App Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
            classmate-1.5.1.jar
            hibernate-validator-6.2.0.F
            hibernate-validator-annot
            hibernate-validator-cdi-6.
            jakarta.el-3.0.3.jar
            jakarta.validation-api-2.0.2
            javax.servlet.jsp.jstl-1.2.1.ja
            javax.servlet.jsp.jstl-api-1.2
            jboss-logging-3.4.1.Final.ja
```

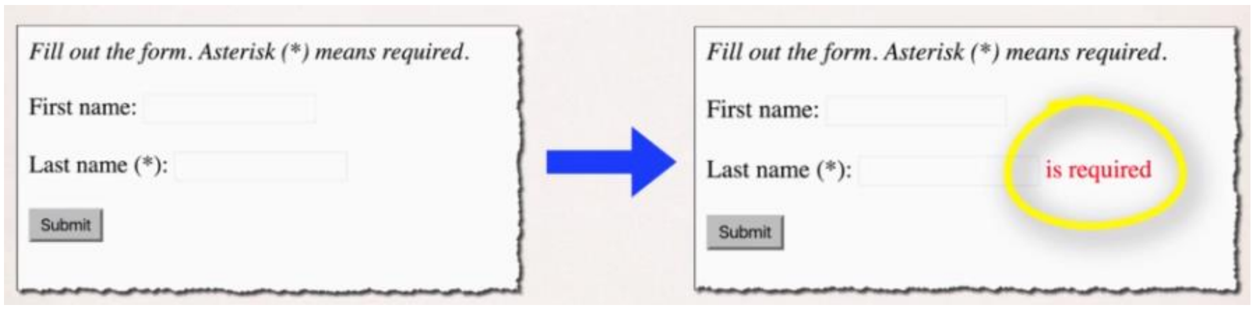


hibernate-validator-6.2.0.Final > dist > lib > required

Name	Date modified	Type	Size
classmate-1.5.1	12/23/2020 5:42 PM	Executable Jar File	67 KB
jakarta.el-3.0.3	12/23/2020 5:42 PM	Executable Jar File	233 KB
jakarta.validation-api-2.0.2	12/23/2020 5:42 PM	Executable Jar File	90 KB
jboss-logging-3.4.1.Final	12/23/2020 5:42 PM	Executable Jar File	60 KB

➢ Spring MVC Form Validation for Required Fields

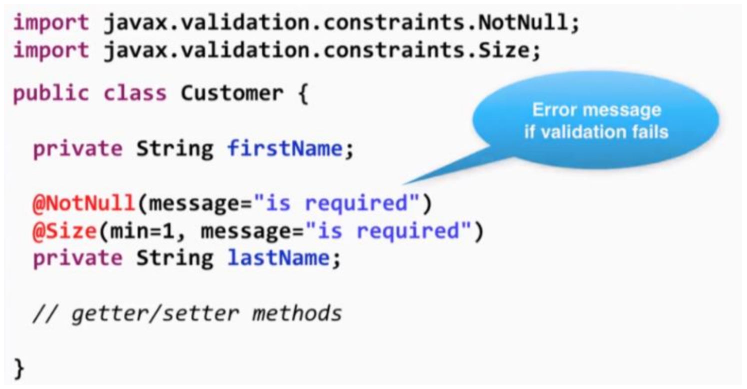
In this example we will make the ***Last Name*** required for user to fill in.


Fill out the form. Asterisk (*) means required.
First name:
Last name (*):
Submit
Fill out the form. Asterisk (*) means required.
First name:
Last name (*):
is required
Submit

Development Process:

1. Add Validation rule to Customer Class
2. Display error messages on HTML Form
3. Perform Validation in the Customer Class
4. Update Confirmation Page

- **Add validation rule to Customer Class**



```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Customer {

  private String firstName;

  @NotNull(message="is required")
  @Size(min=1, message="is required")
  private String lastName;

  // getter/setter methods

}
```

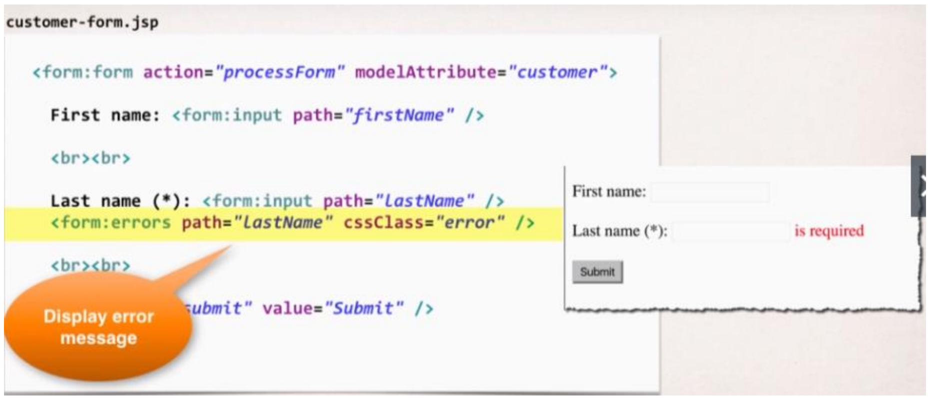
- **Display the Error Message on HTML Form**

Also add the reference for the Spring Form tag library.

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

With that add this **CSS styling** into the web page to show the error in red color

```
<head>
        <meta charset="ISO-8859-1">
        <title>Customer Registration Form</title>
        <style>
                .error {color:red}
        </style>
</head>
```

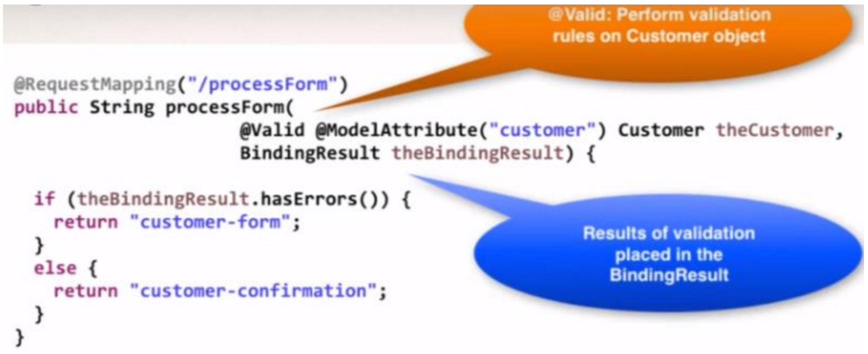


```
customer-form.jsp

<form:form action="processForm" modelAttribute="customer">

  First name: <form:input path="firstName" />

  <br><br>

  Last name (*): <form:input path="lastName" />
  <form:errors path="lastName" cssClass="error" />

  <br><br>

           ubmit" value="Submit" />
```

We will see the **error message** only if the validation gets fail.

✓ **Perform Validation in the Customer Class**

`@Valid` will perform the validation rules using the ***`Hiibernate Valdiator API`*** on the object that we are getting from the form and then the result of the Validation test will be placed in ***`theBindingRessult`***.
Here we are checking if the ***`bindingResult`*** is having any error then will return to the same customer-form or else simply send them to the Confirmation page/success page of our application.



```
@RequestMapping("/processForm")
public String processForm(
                  @Valid @ModelAttribute("customer") Customer theCustomer,
                  BindingResult theBindingResult) {

  if (theBindingResult.hasErrors()) {
    return "customer-form";
  }
  else {
    return "customer-confirmation";
  }
}
```

@Valid: Perform validation rules on Customer object
Results of validation placed in the BindingResult

✓ **Update the Confirmation page**

```
<html>
<head>
<meta charset="ISO-8859-1">
<title>Customer Confirmation</title>
</head>
<body>
       The Customer is confirmed: ${customer.firstName}  ${customer.lastName}
       <br><br>

</body>
</html>
```

Special Note about BindingResult Parameter Order

When performing Spring MVC validation, the location of the *`BindingResult`* parameter is very important. In the method signature, **the *`BindingResult`* parameter must appear immediately after the model attribute**.

If you place it in any other location, Spring MVC validation will not work as desired. In fact, your validation rules will be ignored.

```
        @RequestMapping("/processForm")
        public String processForm(
               @Valid @ModelAttribute("customer") Customer theCustomer,
               BindingResult theBindingResult) {
            ...
        }
```

Here is the relevant section from the Spring Reference Manual

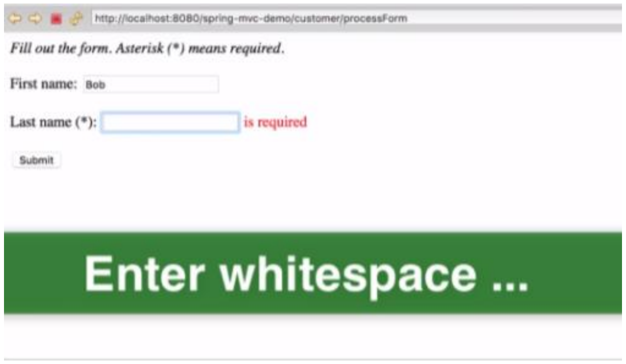
---
Defining @RequestMapping methods
@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.
...
The `Errors` or `BindingResult` parameters have to follow the model object that is being bound immediately ...

Let’s experiment and Break it up!

In *CustomerController .java*; let’s add some logging message

```
@RequestMapping("/processForm")
public String processForm(@Valid @ModelAttribute("customer") Customer theCustomer,
            BindingResult theBindingResult) {

        System.out.println("Last Name: |"+ theCustomer.getLastName()+"|");


        if(theBindingResult.hasErrors()) {
            return "customer-form";
        }
        else
        return "customer-confirmation";
}
```

Now let’s go to the customer-form.jsp on web page;
We will try to enter Whitespace in last Name TextField and the submit it to check whether our validation will allow it pass through or not and that why we have added that logging message into the controller method.

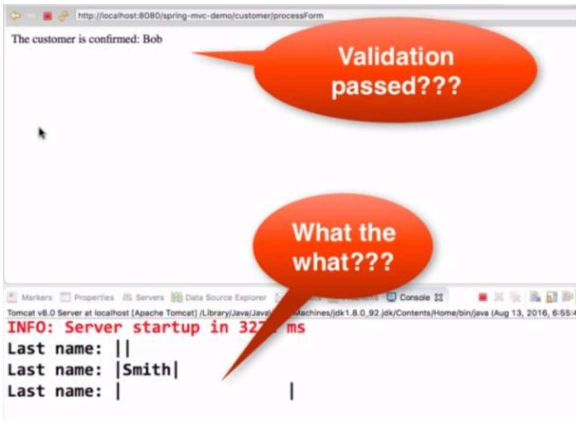

http://localhost:8080/spring-mvc-demo/customer/processForm

Fill out the form. Asterisk (*) means required.

First name: Bob

Last name (*): is required

Well, lastName field with **whitespace passes the validation** and let's check the logging message into the console.
So, how does it pass the validation as we have added @NotNull and @Size annotations?



The customer is confirmed: Bob

```
Tomcat v8.0 Server at localhost [Apache Tomcat] /Library/Java/Java... Machines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Aug 13, 2016, 6:55:4
INFO: Server startup in 32.. ms
Last name: ||
Last name: |Smith|
Last name: |                 |
```

The Solution is: ***@InitBinder***

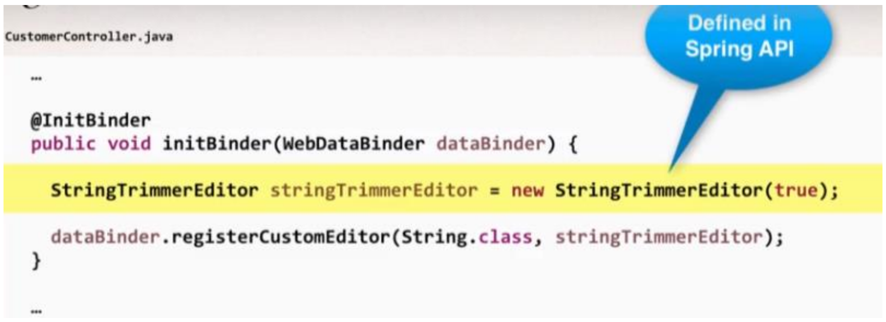
what we need to do is trim the whitespace from the input fields.

1. @InitBinder annotation works as a pre-processor
2. Every Web request that will come to our Controller, this annotation code will execute first
3. Therefore, we'll annotate a method with @InitBinder, then all requests coming in will get pre-processed.

So, we are going to use @InitBinder to trim the strings i.e., Remove the leading and tailing white space, and incase the string having just the whitespaces then we will trim it all the way to make it null object. This will actually resolve our validation issue.

Development Process:

1. Register the custom Editor in Controller and we will do that by using @InitBinder



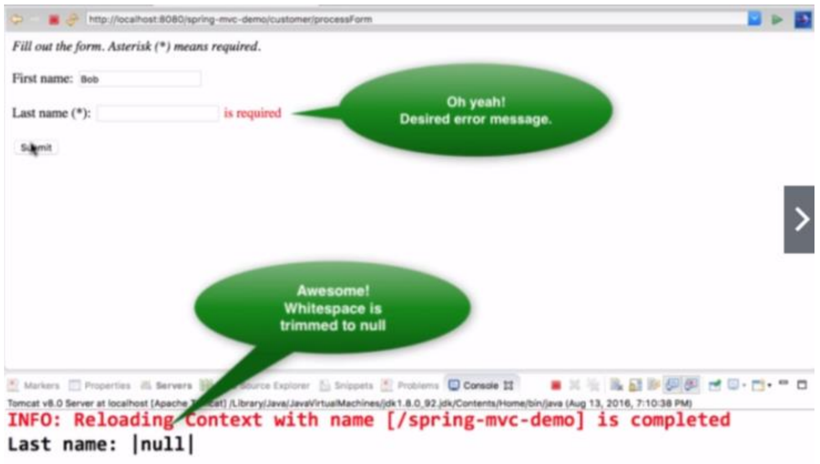
CustomerController.java

```
...

@InitBinder
public void initBinder(WebDataBinder dataBinder) {

  StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

  dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
}

...
```

Here, we're using the ***StringTrimmerEditor*** Class which is defined in Spring API, It's purpose is to trim Strings.

And in the constructor, we are keeping True Boolean value so that it can trim the String value to null. Once the ***StringTrimmerEditor*** Object is created then we're registering it to the ***dataBinder*** as a custom Editor. Actually, *we are saying that every String class apply the **StringTrimmerEditor**.*

2. Now, All the incoming requests will pass through the method annotated with @InitBinder and it will resolve our issue of whitespace with Last Name textField.



```
http://localhost:8080/spring-mvc-demo/customer/processForm
```

Fill out the form. Asterisk (*) means required.

First name: Bob

Last name (*): is required

```
Tomcat v8.0 Server at localhost [Apache Tomcat] /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java (Aug 13, 2016, 7:10:38 PM)
INFO: Reloading Context with name [/spring-mvc-demo] is completed
Last name: |null|
```

Section 16: Spring MVC Form Validation - Validating Number Ranges and Regular Expressions

9 / 9 | 37min

➢ Validating a Number Range

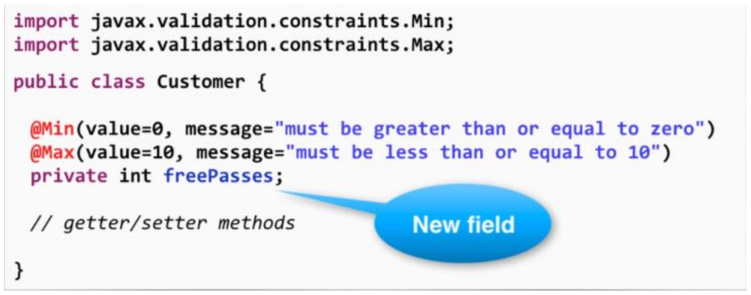
Will use @Min and @Max.

What are we going to do here?

- Add a new Input Field on our form for: **freePasses.**
- User can only enter a range: 0 to 10.

Development Process

- Add Validation rule to Customer class



```
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;

public class Customer {

  @Min(value=0, message="must be greater than or equal to zero")
  @Max(value=10, message="must be less than or equal to 10")
  private int freePasses;

  // getter/setter methods

}
```

Message attribute is actually the error message that will generate if the validation failed.

- Display error messages on HTML form

File: customer-form.jsp

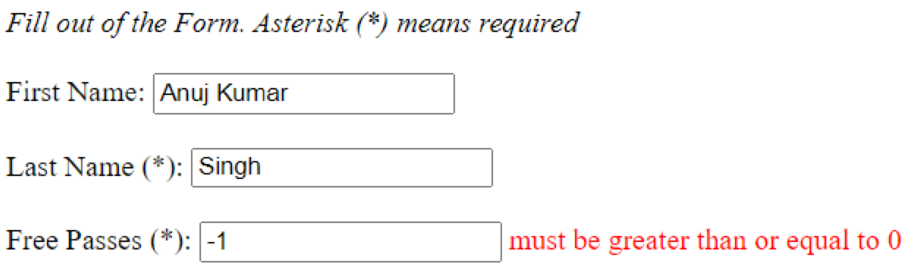
```
<br><br>

Free Passes (*): <form:input path="freePasses"/>
<form:errors path="freePasses" cssClass="error"/>
<br><br>
```

- Perform Validation in the Controller class
  //nothing to do here, keep everything as it is

- Update Confirmation Page

  File: customer-confirmation.jsp

```
<br><br>
Free Passes : ${customer.freePasses}
```


Fill out of the Form. Asterisk (*) means required

First Name: Anuj Kumar

Last Name (*): Singh

Free Passes (*): -1 must be greater than or equal to 0

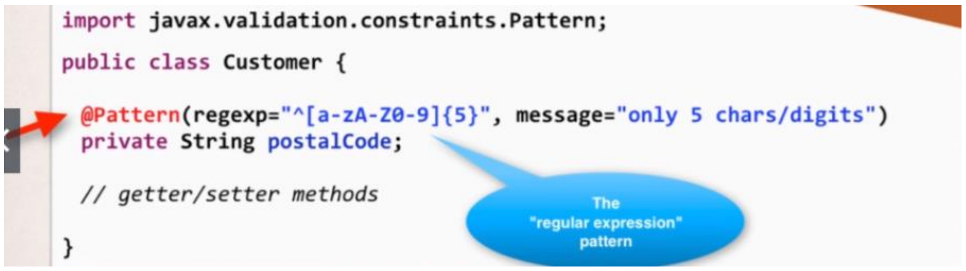
➢ **Applying Regular Expressions**

We're going to validate the postal code field.

<u>Validation Process</u>

- Add a new input field on your form for: Postal Code
- User can only enter 5 chars/digit neither more than that nor less than; hence we will write the regular expression for that.

✓ <u>Adding the Validation rule to Customer class:</u>



```
import javax.validation.constraints.Pattern;
public class Customer {

  @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")
  private String postalCode;

  // getter/setter methods

}
```

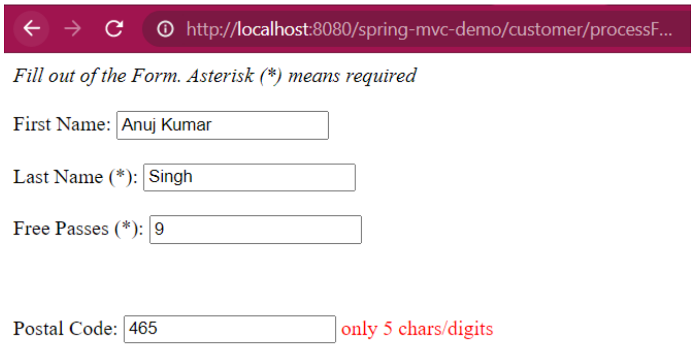
✓ <u>Display the Error message ono HTML Form</u>

```
<br><br>
Postal Code: <form:input path="postalCode"/>
<form:errors path="postalCode" cssClass="error"/>
<br><br>
```

✓ <u>Update the Confirmation Page</u>

```
<br><br>
Postal Code: ${customer.postalCode}
```

On running on server: So, Validation with Regular expression is working


http://localhost:8080/spring-mvc-demo/customer/processF...

Fill out of the Form. Asterisk (*) means required

First Name: Anuj Kumar

Last Name (*): Singh

Free Passes (*): 9

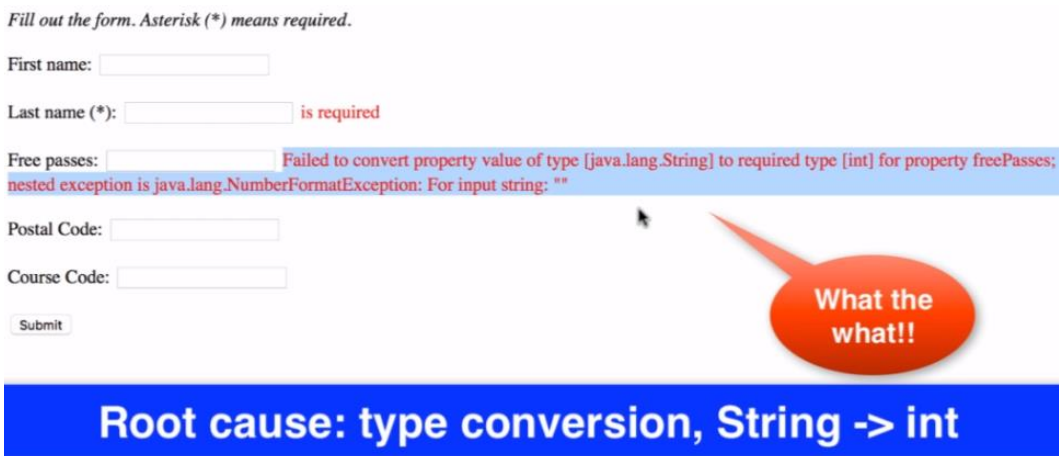
Postal Code: 465 only 5 chars/digits

➢ How to Integer Field Required?

Added @NotNull annotation will make sure that the ***freePasses*** field is required.

```
public class Customer {

        @NotNull(message="is required")
        @Min(value = 0,message = "must be greater than  or equal to 0")
        @Max(value = 10,message = "must be less than  or equal to 10")
        private int freePasses;
        .
        .
        .
        .  //getters and setters
        .

}
```

Output we get is: root cause is TypeCoversion error i.e., not able to convert String type into the primitive type of int. So, we need to make the Type Integer rather than int.


Fill out the form. Asterisk (*) means required.

First name:

Last name (*): is required

Free passes: Failed to convert property value of type [java.lang.String] to required type [int] for property freePasses; nested exception is java.lang.NumberFormatException: For input string: ""

Postal Code:

Course Code:

Submit

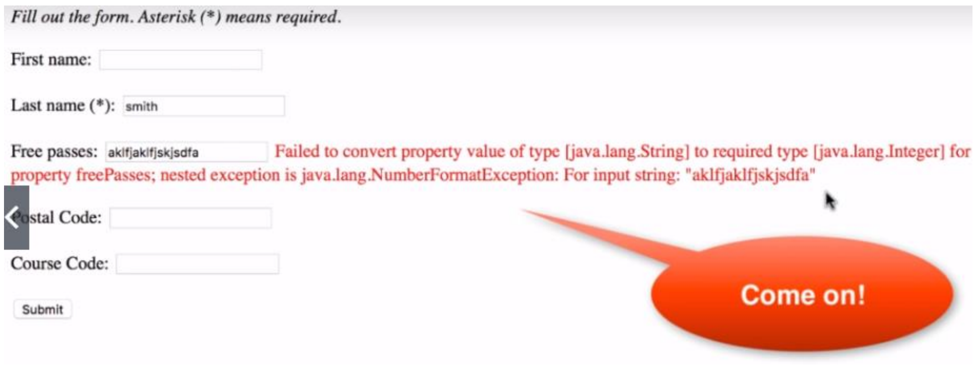
Fix:

```
        @NotNull(message="is required")
        @Min(value = 0,message = "must be greater than  or equal to 0")
        @Max(value = 10,message = "must be less than  or equal to 10")
        private Integer freePasses;
```

We are using Integer wrapper class; it works because our @InitBinder annotated method StrimTrimEditor will actually trim the input to null if input is null or whitespace. And also update getters and setter in the class.

So, our aim of making field required has been met.

- ***Submitting some String value for the Integer type field freePasses***, Let's see the output:



Fill out the form. Asterisk (*) means required.

First name:

Last name (*): smith

Free passes: aklfjaklfjskjsdfa Failed to convert property value of type [java.lang.String] to required type [java.lang.Integer] for property freePasses; nested exception is java.lang.NumberFormatException: For input string: "aklfjaklfjskjsdfa"

Postal Code:

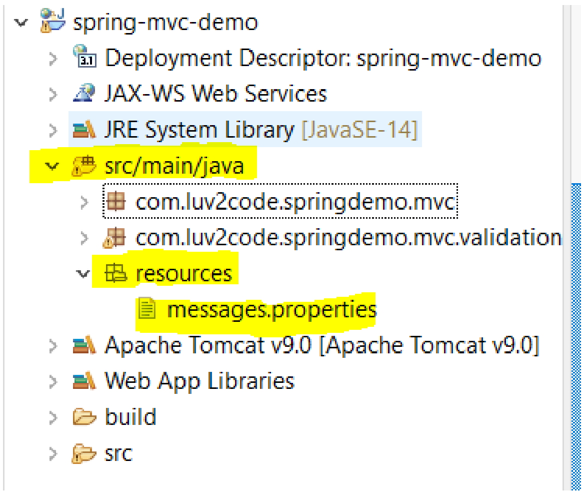
Course Code:

Submit

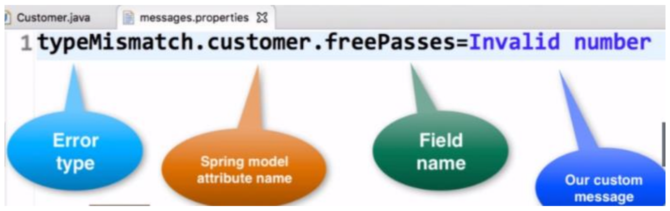
So, fix it, we will follow this Development Process:

1. Create custom error message ("Invalid Number")

   We will create "*resource*" folder and within that we will put our *messages.properties* file
   - *src/main/java/resources/messages.properties*


spring-mvc-demo
Deployment Descriptor: spring-mvc-demo
JAX-WS Web Services
JRE System Library [JavaSE-14]
src/main/java
com.luv2code.springdemo.mvc
com.luv2code.springdemo.mvc.validation
resources
messages.properties
Apache Tomcat v9.0 [Apache Tomcat v9.0]
Web App Libraries
build
src

into the *messages.properties* file:



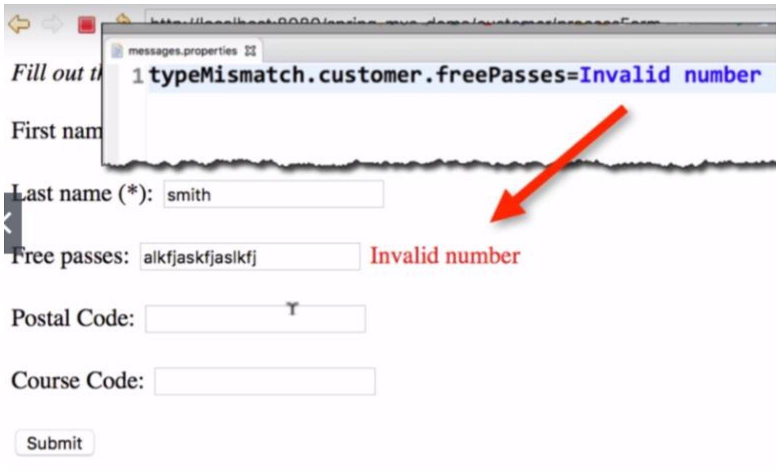
```
typeMismatch.customer.freePasses=Invalid number
```

2. Load Custom messages resource in Spring config file.

Now we need to tell spring about our custom error message i.e., we will load this custom error message inside our spring config file.
Open the ***spring-mvc-demo-servlet.xml*** file, and add:

```
<!-- Load the custom Message resources -->
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
<property name="basenames"  value="resources/messages"></property>
</bean>
```

3. Run on Server:



```
typeMismatch.customer.freePasses=Invalid number
```

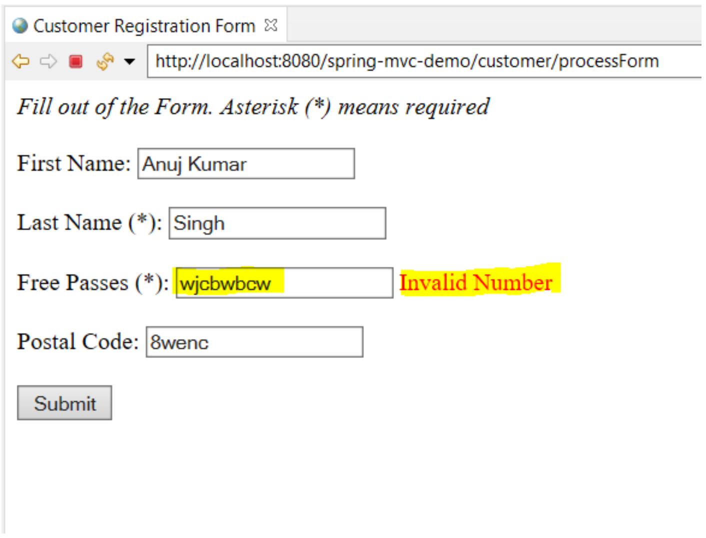
Fill out t
First nam
Last name (*): smith
Free passes: alkfjaskfjaslkfj Invalid number
Postal Code:
Course Code:

Now let's see how we came up with text that we have written in *messages.properties* file.

So, we will add some logging message in our Controller method, in which we'll inspect the ***bindingResult*** Object:

```
@RequestMapping("/processForm")
public String processForm(@Valid @ModelAttribute("customer") Customer theCustomer,
        BindingResult theBindingResult) {

        System.out.println("Last Name: |"+ theCustomer.getLastName()+"|");
        System.out.println("Binding Result: "+theBindingResult);
        System.out.println("\n\n\n\n");
        if(theBindingResult.hasErrors()) {
                return "customer-form";
        }
        else
        return "customer-confirmation";
}
```

Now on giving some random string into the freePasses field, check for the logging message that we got:


Customer Registration Form
http://localhost:8080/spring-mvc-demo/customer/processForm
Submit

Fill out of the Form. Asterisk (*) means required

First Name: Anuj Kumar

Last Name (*): Singh

Free Passes (*): wjcbwbcw Invalid Number

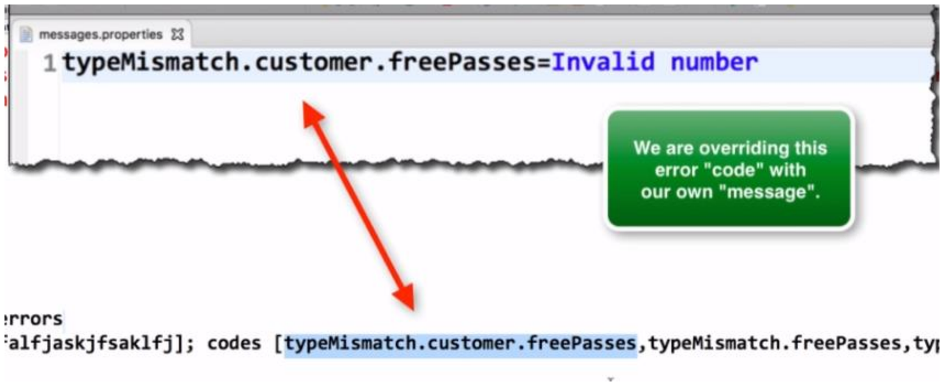
Postal Code: 8wenc

```
Logging Message:

Binding Result: org.springframework.validation.BeanPropertyBindingResult: 1 errors
Field error in object 'customer' on field 'freePasses': rejected value [wjcbwbcw]; codes
[typeMismatch.customer.freePasses,typeMismatch.freePasses,typeMismatch.java.lang.Integer,typeMisma
tch]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes
[customer.freePasses,freePasses]; arguments []; default message [freePasses]]; default message
[Failed to convert property value of type 'java.lang.String' to required type 'java.lang.Integer'
for property 'freePasses'; nested exception is java.lang.NumberFormatException: For input string:
"wjcbwbcw"]
```

Look at the highlighted part; where error codes are given:

Rejected Value: `[wjcbwbcw]`, It is the value that we have given in the input text field

Codes:
`[`***`typeMismatch.customer.freePasses`***`,typeMismatch.freePasses,typeMismatch.java.lang.Integer,typeMisma tch]`



```
typeMismatch.customer.freePasses=Invalid number
```

```
rrors
alfjaskjfsaklfj]; codes [typeMismatch.customer.freePasses,typeMismatch.freePasses,ty
```

So, we are overriding their default error code and we are providing our own custom message.

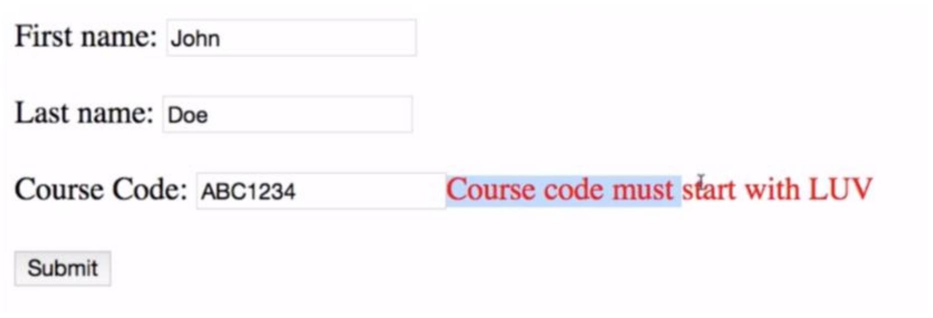
There is some other error codes mentioned in the array as well, basically error codes start with very specific error code from the left to very generic all the way to right in the array. Therefore, in this way we can use bindingResult to check the error codes and add our own custom message on unsuccessful validations.

➢ Custom Validation with Spring MVC:

Here we will add our custom annotation.

Ex: Custom Business Rule that we need to add here for the Course Code field

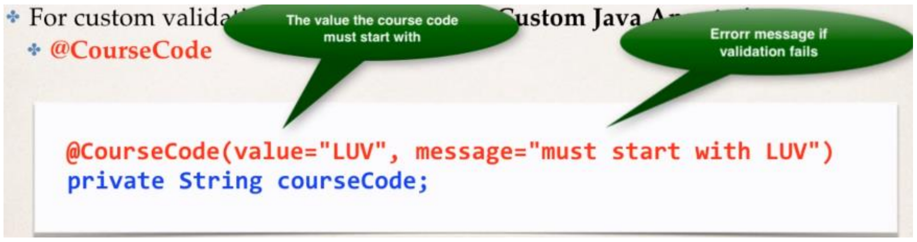
So, here we want to start the course code form LUV and if user has written any other text then validation will give error by putting the message "Course must start with LUV".



First name:	John
Last name:	Doe
Course Code:	ABC1234 Course code must start with LUV

Custom Validation

- Spring MVC calls our Custom Validation
- Custom Validation returns Boolean value for pass/fail (true/false) based on the code user entered on the field/
- We will create our own custom Java Annotation ***@CourseCode***

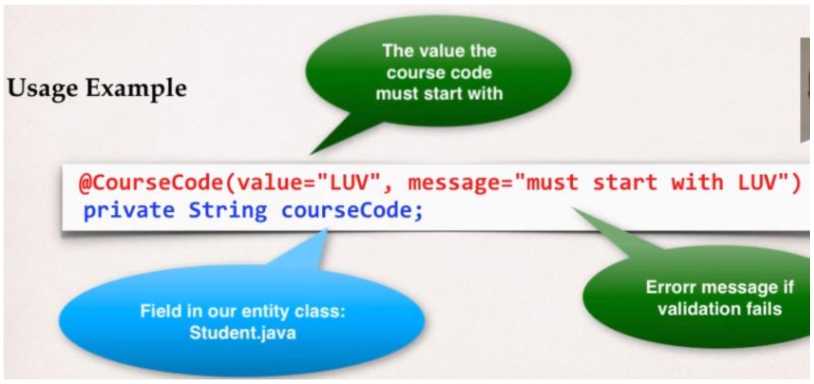

For custom valid
@CourseCode
The value the course code must start with
ustom Java A
Erorr message if validation fails

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

Development Process

1. Create custom Validation Rule

   - Create ***@CourseCode*** annotation
   - Create ***CourseCodeConstraintValidator Class***, this is where we put own custom Logic for Validation and determine true or false as if value passes our validation or not.


Usage Example
The value the course code must start with
Field in our entity class: Student.java
Errorr message if validation fails

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.METHOD })
public @interface CourseCode {
```

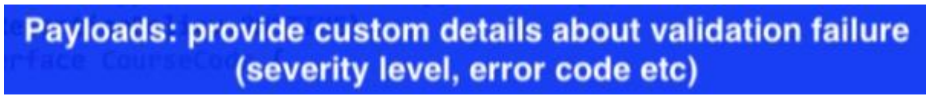
- @Constraint with `validatedBy` property will actually contain the Helper class that has the business rule for validating the process.
- @Target basically says where can we apply this annotation like to a method or a field and here, we decided to put this annotation on method or on field.
- @Retention means how long we should retain it and we want to keep the annotation in the compiled java byte code so we can use it and intercept on it during runtime.

This annotation has two parameters i.e., "value" and "message". So, we need to add some method declarations here inside the annotation.

```
//define the course code
public String value() default "LUV";
```

This means that the annotation has parameter called value and if user doesn't provide then we will use the default value of "LUV"

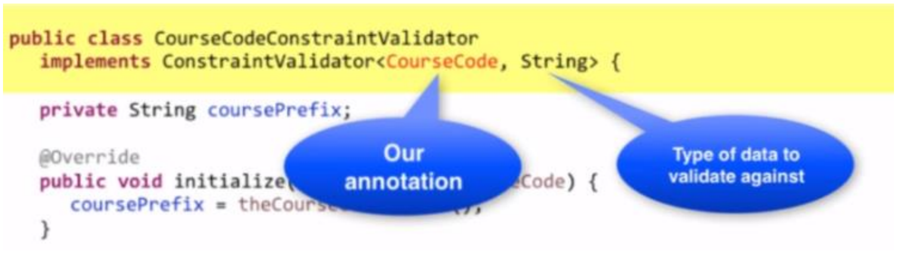
```
//define default error message
public String message() default "must start with LUV";

//define default groups
public Class<?>[] groups() default {};

//define default payloads
public Class<? extends Payload>[] payload() default {};
```


Payloads: provide custom details about validation failure (severity level, error code etc)

✓ *Now **CourseCodeConstraintValidator** Helper class*

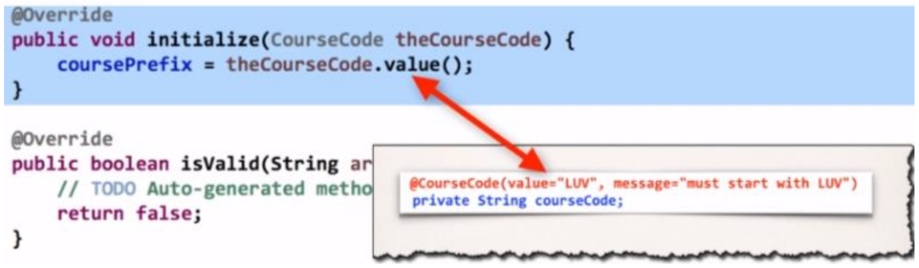
```
        public class CourseCodeConstraintValidator implements ConstraintValidator<CourseCode,
String>{

        private String coursePrefix;
        @Override
        public void initialize(CourseCode theCourseCode) {
        // TODO Auto-generated method stub
        coursePrefix = theCourseCode.value();
        }

        @Override
        public boolean isValid(String theCode, ConstraintValidatorContext
theConstraintValidatorContext) {
        // TODO Auto-generated method stub
        boolean result = theCode.startsWith(coursePrefix);

        return result;
        }
}
```

Code Explaination:



```
public class CourseCodeConstraintValidator
   implements ConstraintValidator<CourseCode, String> {

   private String coursePrefix;

   @Override
   public void initialize(...Code) {
      coursePrefix = theCours...
   }
```

- `public void initialize(CourseCode theCourseCode)`



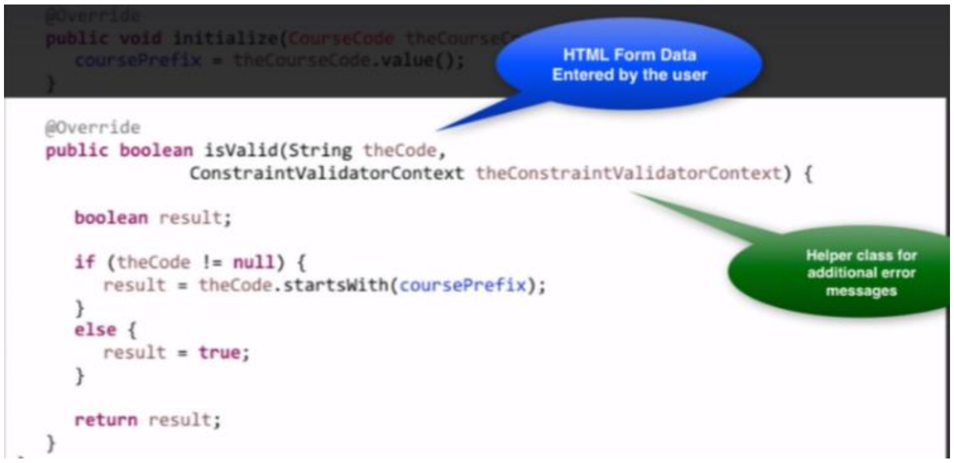
```
@Override
public void initialize(CourseCode theCourseCode) {
    coursePrefix = theCourseCode.value();
}

@Override
public boolean isValid(String ar
    // TODO Auto-generated metho
    return false;
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```

During initialization, it will get the actual value that passed in i.e., the prefix against which we want to validate in. This specially sets everything up, so our validator is up and running and now Spring MVC can actually use it for validation and then it takes us to another method isValid().

- `public boolean isValid(String theCode, ConstraintValidatorContext theConstraintValidatorContext)`

This method will be called by Spring MVC at runtime it will say here is the data submitted by the form and check if it's valid? Spring MVC will pass the Form Data entered by the user in String theCode parameter and also pass the ConstrainstValidatorContext which is just a helper class for additional error messages.


HTML Form Data Entered by the user
Helper class for additional error messages

```
@Override
public void initialize(CourseCode theCourseC
    coursePrefix = theCourseCode.value();
}

@Override
public boolean isValid(String theCode,
                    ConstraintValidatorContext theConstraintValidatorContext) {

    boolean result;

    if (theCode != null) {
        result = theCode.startsWith(coursePrefix);
    }
    else {
        result = true;
    }

    return result;
}
```

- **Business Rule:**

  If the data (`theCode`) is not equal to null then we will check whether "*`theCode`*" start with the "`coursePrefix`" that is given in annotation's **value** property and will return True/Flase based on that.

  And incase **`theCode`** is empty i.e., ***`null`*** then we are simply returning ***`True`***.

2. Add validation rule to Customer class
3. Display error message on HTML from

   File: customer-form.jsp

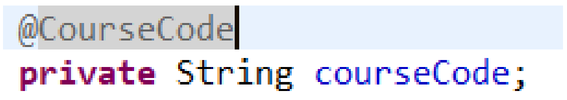
   ```
   <br><br>
   Course Code: <form:input path="courseCode"/>
   <form:errors path="courseCode" cssClass="error"/>
   <br><br>
   ```

4. Update confirmation Page

   File: customer-confirmation.jsp

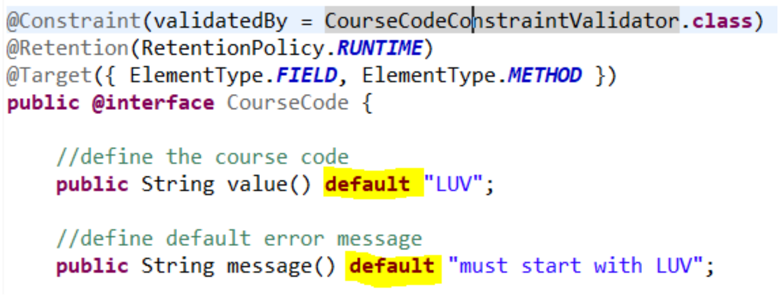
   ```
   <br><br>
   Course Code : ${customer.courseCode}
   ```

Now it will work as per our Business Rule:

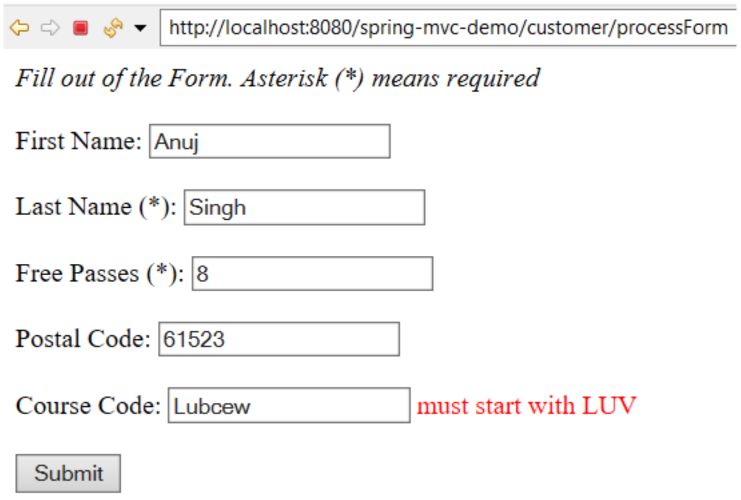


```
@CourseCode
private String courseCode;
```

Inside the Customer class



```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.METHOD })
public @interface CourseCode {

    //define the course code
    public String value() default "LUV";

    //define default error message
    public String message() default "must start with LUV";
```



http://localhost:8080/spring-mvc-demo/customer/processForm

Fill out of the Form. Asterisk (*) means required

First Name: Anuj

Last Name (*): Singh

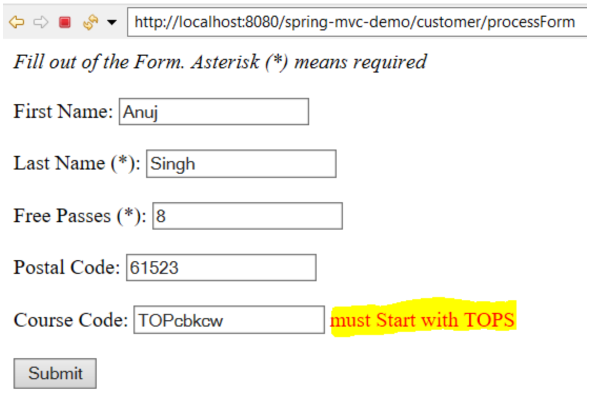
Free Passes (*): 8

Postal Code: 61523

Course Code: Lubcew must start with LUV

Submit

- ✓ Here *@CourseCode* will take `default value and message` that we have given while declaring the annotation.
- ✓ Below is the *customized version* of our *@CourseCode* annotation.



http://localhost:8080/spring-mvc-demo/customer/processForm

Fill out of the Form. Asterisk (*) means required

First Name: Anuj

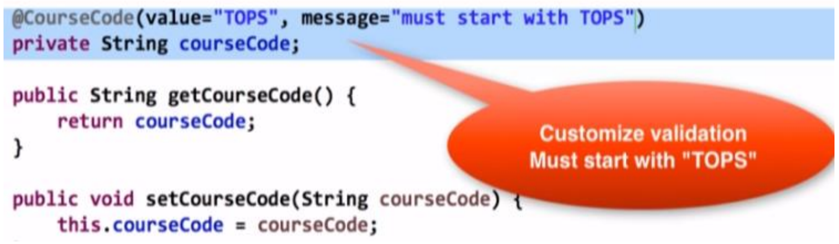
Last Name (*): Singh

Free Passes (*): 8

Postal Code: 61523

Course Code: TOPcbkcw must Start with TOPS

Submit



```
@CourseCode(value="TOPS", message="must start with TOPS")
private String courseCode;

public String getCourseCode() {
    return courseCode;
}

public void setCourseCode(String courseCode) {
    this.courseCode = courseCode;
```

FAQ: Spring MVC Custom Validation - Possible to validate with multiple strings?

Spring MVC Custom Validation - FAQ: Is it possible to integrate multiple validation string in one annotation?

Question:
Is it possible to integrate multiple validation string in one annotation? For example, validate against both LUV and TOPS.

Answer:
Yes, you can do this. In your validation, you will make use of an array of strings.

Here's an overview of the steps.

1. Update CourseCode.java to use an array of strings

2. Update CourseCodeConstraintValidator.java to validate against array of strings

3. Update Customer.java to validate using array of strings

Detailed Steps

1. Update CourseCode.java to use an array of strings

Change the value entry to an array of Strings:

```
    // define default course code
    public String[] value() default {"LUV"};
```

Note the use of square brackets for the array of Strings. Also, the initialized value uses curley-braces for array data.

2. Update CourseCodeConstraintValidator.java to validate against array of strings

Change the field for coursePrefixes to an array

```
private String[] coursePrefixes;
```

Update the isValid(...) method to loop through the course prefixes. In the loop, check to see if the code matches any of the course prefixes.

```
    @Override
    public boolean isValid(String theCode,
                        ConstraintValidatorContext theConstraintValidatorContext) {
        boolean result = false;

        if (theCode != null) {

            //
            // loop thru course prefixes
            //
            // check to see if code matches any of the course prefixes
            //
            for (String tempPrefix : coursePrefixes) {
                result = theCode.startsWith(tempPrefix);

                // if we found a match then break out of the loop
                if (result) {
                    break;
                }
            }
        }
        else {
            result = true;
        }

        return result;
  }
```

3. Update Customer.java to validate using array of strings

```
    @CourseCode(value={"TOPS", "LUV"}, message="must start with TOPS or LUV")
    private String courseCode;
```

Note the use of curley braces.

Complete Source Code:
https://gist.github.com/darbyluv2code/0275ddb6e70e085a10fd464e36a42739

That's it. This provides a solution to integrate multiple validation string in one annotation. In this example, the code validates against both LUV and TOPS.

Below GitHub Repository contains all the discussed exercises:

https://github.com/000GEV744/Spring-Git/tree/main/spring-mvc-demo