

Table of Contents

Overview	2
Setup Maven Project on Eclipse.....	3
Setting up Controllers to Handlers Web Request	9
Configuring Multiple Pages.....	16
Configure Multiple Controller	20
Designing 3-Tier Application	24
Extracting Dynamic URL Path	35
Accessing Request Parameter	39
Injecting Request Parameter.....	43
Configuring Request Parameters	46
Handling Exceptions using XML.....	48
Mapping Multiple and Default Exception	54
Handling Exceptions Using Annotation	58
Course Summary	61
Quiz.....	62

Overview

The Spring MVC web framework is a Java framework that includes all the standard features of a core Spring framework but utilizes the model-view-controller design pattern. Working with this framework, you can produce robust, flexible, loosely-coupled, three-tiered web applications.

In this course, you'll explore the basic tiers in a Spring MVC application. You'll configure applications with multiple controllers, multiple views, and simulate the classic 3-tier structure of a web application. You'll also learn how to deal with request parameters, dynamic URL paths, and exceptions thrown in your application.

Some of the tasks you will complete in this course include deploying WAR files, configuring multiple pages and controllers in your app, designing tiers and separating them into different packages, and extracting, accessing, injecting, and configuring request parameters.

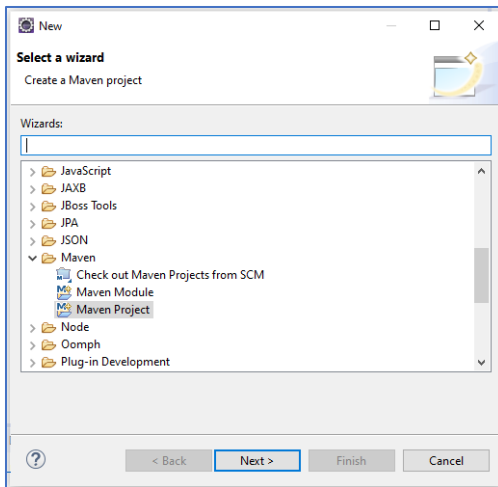
Setup Maven Project on Eclipse

1. Open Eclipse IDE



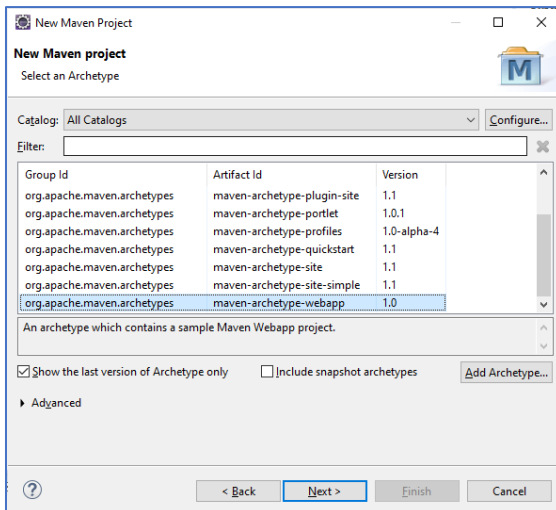
2. Create New Maven Project

File > New > Maven Project



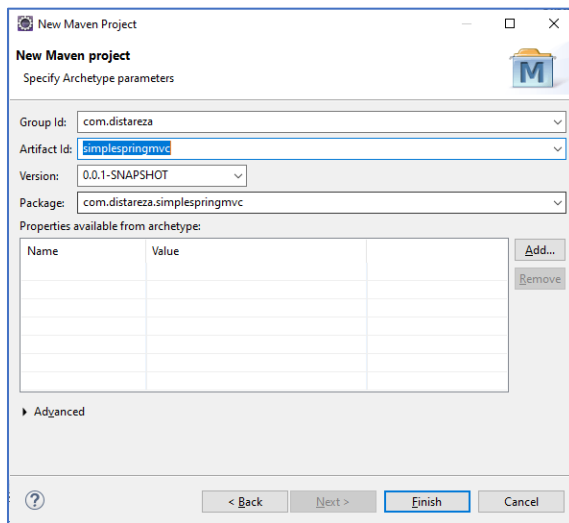
3. select or choose **Workspace** location

4. Choose **maven-archetype-webapp (ver 1.0)** as the archetype, click next



Now, in order to generate a project template for a Maven application, we need to choose an archetype. An archetype in Maven is a project templating toolkit. This will set up all of the boilerplate configuration for the kind of project that you want to build using Maven. Now, the kind of project that we want to set up is a web application. So choose the Maven archetype webapp version 1.0

5. Defined the artifact Id, and click finish



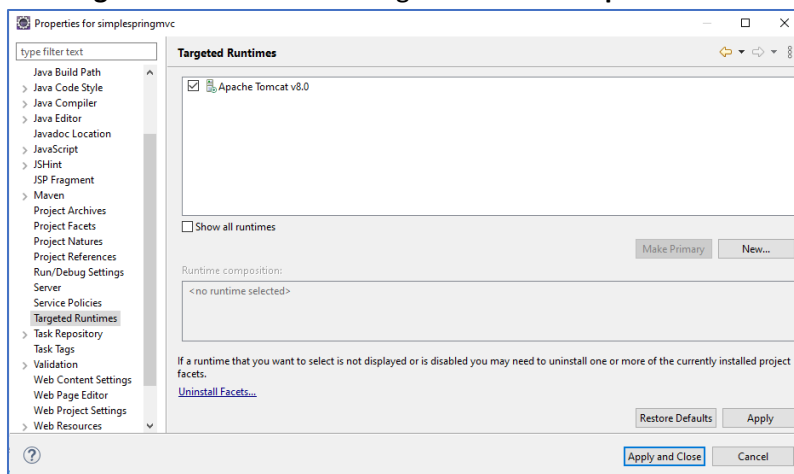
The Group id makes up part of the identifier that is used to uniquely identify this particular project. The Group id is typically the name of the company or the organization that's creating this project.

The Artifact id is the unique name of the project

6. Expand the Project and setup the Target Runtime Server to Tomcat

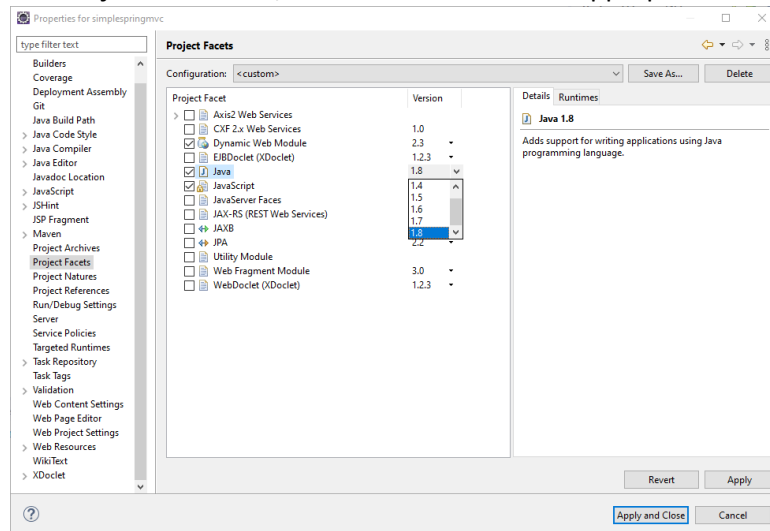
Right click on Project Name and select **Properties**

on **"Target Runtimes"** tab set target runtimes to **Apache Tomcat** and click **Apply**

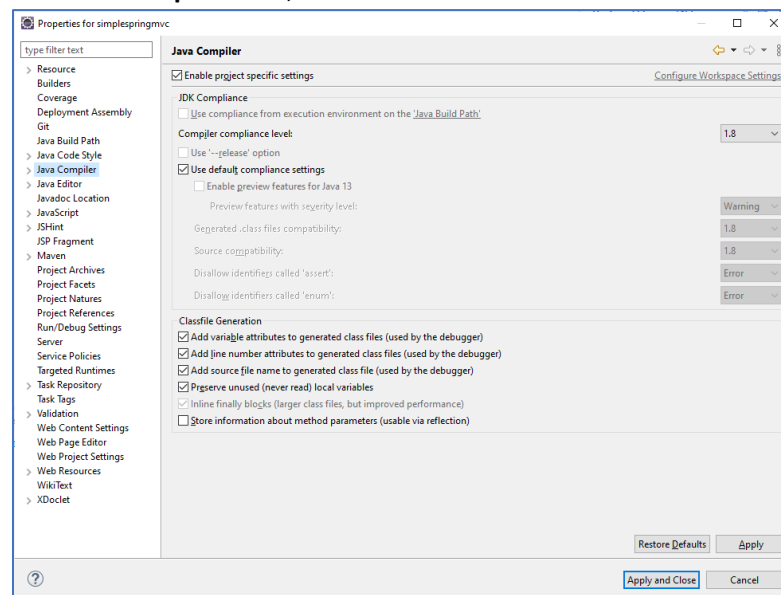


7. Setup Runtime Java Version

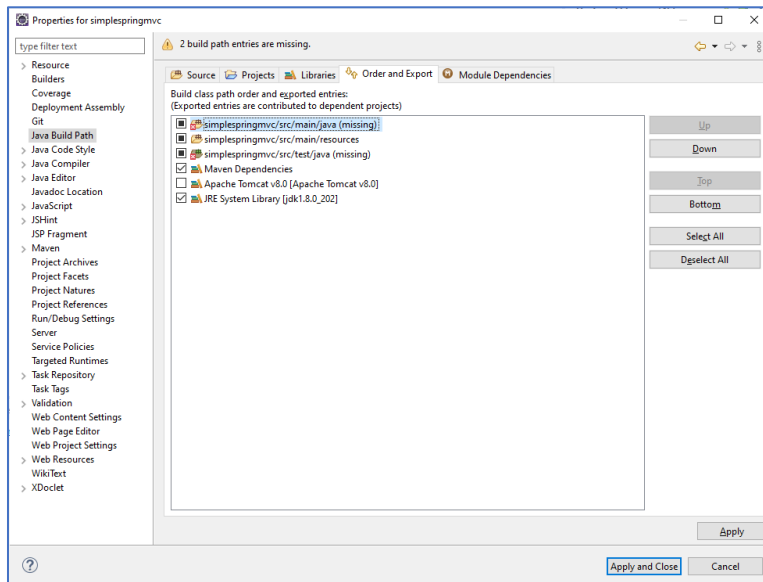
on **“Project Facet”** tab, select **“Java”** and set to appropriate JVM version and click Apply



on **“Java Compiler”** tab, make sure it set to same JVM Version

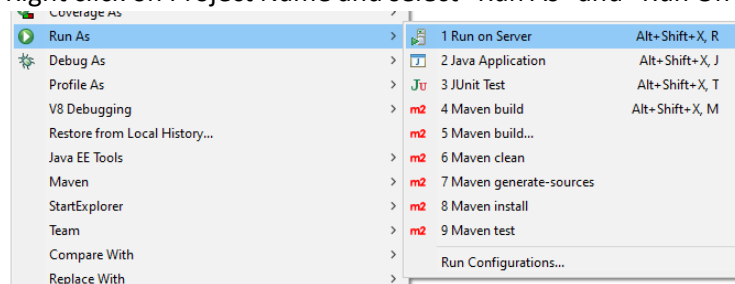


on “**Java Build Path**” tab, make sure that “**Maven Dependencies**” and “**JRE System Library**” is checked, click **Apply and Close**

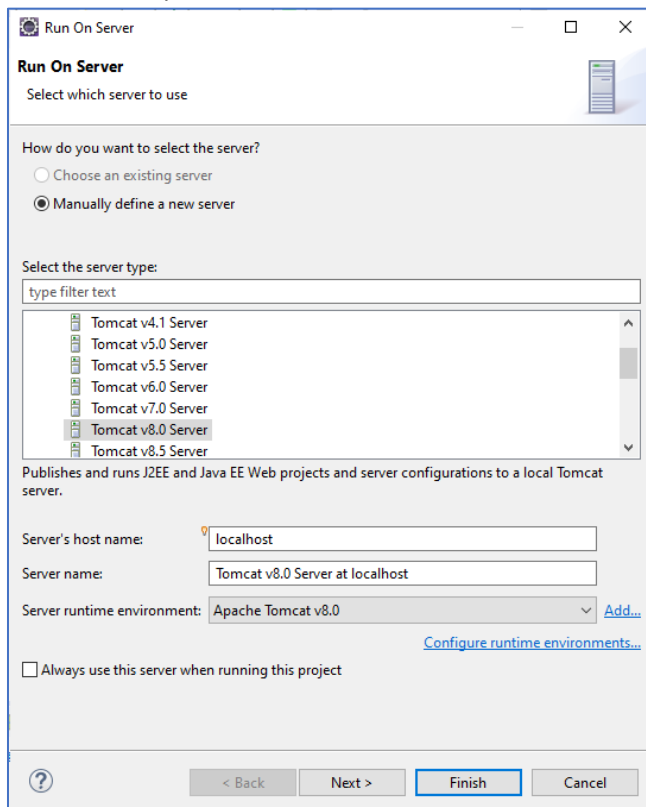


8. Test the default project

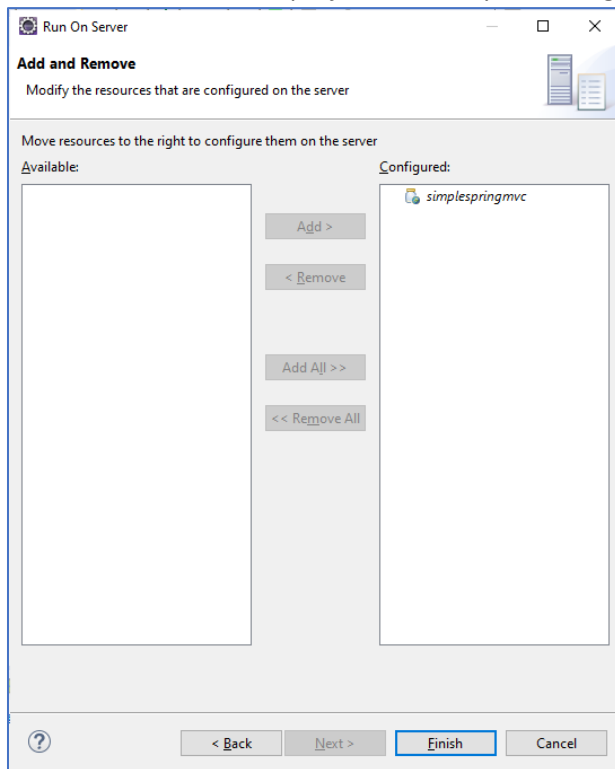
Right click on Project Name and select “Run As” and “Run On Server”



Choose the Apache Tomcat as the server

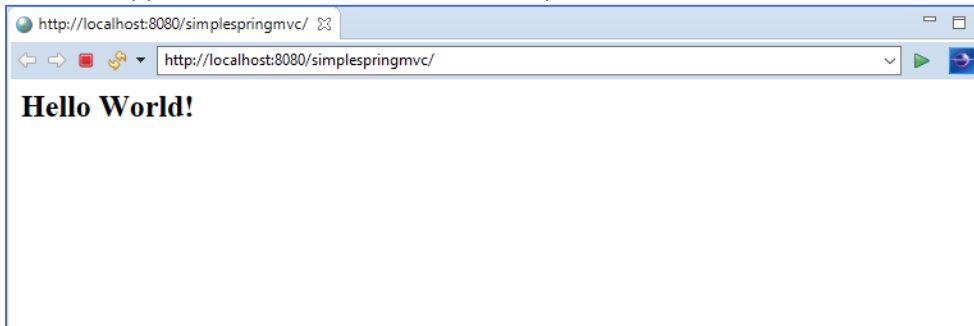


and make sure that our project is already on the right side of “**Configured**” column, click **Finish**



if you wait for a bit, you'll see a bunch of log messages on your console window and you'll see **Hello World!** printed out to screen. Your simple web application has been hosted and is running on your Tomcat Runtime Environment, your Tomcat server. Notice the structure of the URL here, **http://localhost:8080**. That is the port where your application is hosted. This is followed by our project named **simplespringmvc**. This is part of the path. And **Hello World!**, well, Hello World comes from some of the default files that have been set up as a part of your web app.

The Web application should be run inside Eclipse IDE as follows

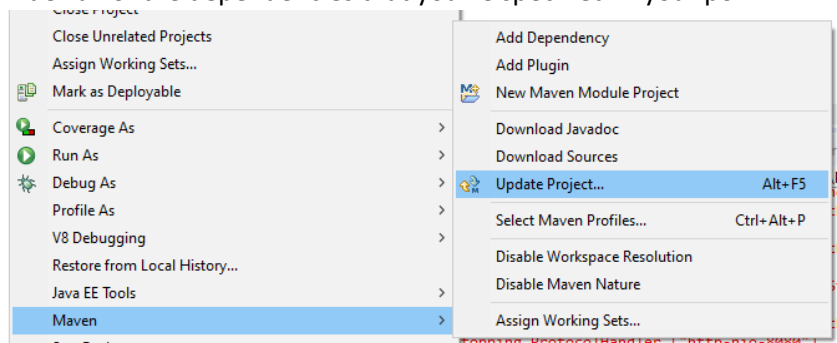


Setting up Controllers to Handlers Web Request

1. Update pom.xml files with adding the spring dependency jar as follows

```
simpleSpringmvc/pom.xml 53
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>com.distareza</groupId>
5   <artifactId>simpleSpringmvc</artifactId>
6   <packaging>war</packaging>
7   <version>0.0.1-SNAPSHOT</version>
8   <name>simpleSpringmvc Maven Webapp</name>
9   <url>http://maven.apache.org</url>
10  <dependencies>
11    <dependency>
12      <groupId>junit</groupId>
13      <artifactId>junit</artifactId>
14      <version>3.8.1</version>
15      <scope>test</scope>
16    </dependency>
17
18    <dependency>
19      <groupId>org.springframework</groupId>
20      <artifactId>spring-webmvc</artifactId>
21      <version>5.1.8.RELEASE</version>
22    </dependency>
23
24    <dependency>
25      <groupId>javax.servlet</groupId>
26      <artifactId>javax.servlet-api</artifactId>
27      <version>3.0.1</version>
28      <scope>provided</scope>
29    </dependency>
30
31    <dependency>
32      <groupId>javax.servlet</groupId>
33      <artifactId>jstl</artifactId>
34      <version>1.2</version>
35    </dependency>
36  </dependencies>
37  <build>
38    <finalName>simpleSpringmvc</finalName>
39    <sourceDirectory>src/main/java</sourceDirectory>
40    <plugins>
41      <plugin>
42        <artifactId>maven-compiler-plugin</artifactId>
43        <version>3.5.1</version>
44        <configuration>
45          <source>1.8</source>
46          <target>1.8</target>
47        </configuration>
48      </plugin>
49    </plugins>
50  </build>
51 </project>
52
```

Once you update the POM.xml , please update the project by do **Right Click** on project name and select **Maven > Update Project** to update or download jar dependencies into your local system and your project class path maven lib dependencies. This should allow Eclipse to find an index all of the dependencies that you've specified in your pom.xml file.



2. Update index.jsp file under webapps folder as follows

```
index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <html>
4 <head>
5   <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10   <h1>Welcome to Spring MVC!</h1>
11   <span>We are going to explore a brand new world</span> ${message}
12 </body>
13 </html>
14
```

3. Update META-INF/web.xml file as follows

```
web.xml
1 <!DOCTYPE web-app PUBLIC
2   "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
3   "http://java.sun.com/dtd/web-app_2_3.dtd" >
4
5 <web-app>
6   <display-name>Archetype Created Web Application</display-name>
7
8   <servlet>
9     <servlet-name>spring-mvc</servlet-name>
10    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
11    <init-param>
12      <param-name>contextConfigLocation</param-name>
13      <param-value>/WEB-INF/spring-mvc-servlet.xml</param-value>
14    </init-param>
15    <load-on-startup>1</load-on-startup>
16  </servlet>
17
18  <servlet-mapping>
19    <servlet-name>spring-mvc</servlet-name>
20    <url-pattern>/</url-pattern>
21  </servlet-mapping>
22
23 </web-app>
24
```

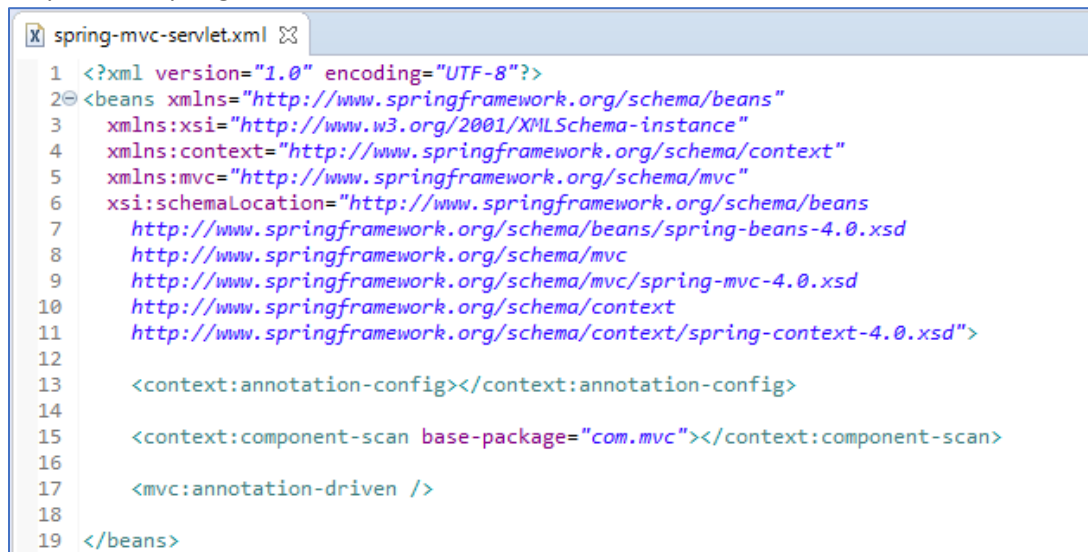
The contents of the web.xml file is used to specify application context.

- Servlet Tag: The servlet configuration is a part of the web application context to set up bean specification for the dispatcher servlet that will be injected into Spring application.
 - Servlet name
Any name that reference the Spring framework's dispatcher servlet
 - Servlet class
Specify the front controller for Spring MVC application, which is **Dispatcher Servlet** (*org.springframework.web.servlet.DispatcherServlet*)
 - Initialization parameters.
The **contextConfigLocation** parameter tells Spring MVC where exactly to look for and load configuration files. The location can be anywhere within our web application and in this example it specified that servlet configuration file **spring-mvc-servlet.xml** is under the **/WEB-INF/** folder. This is *relative* to the root directory of our application.
All files that contain servlets specific configuration, have to have the suffix **-servlet**.
If your configuration files are in the WEB-INF folder, this contextConfigLocation

initialization parameter is completely optional because Spring MVC will automatically look within the WEB-INF folder. However, if you want to place your XML files somewhere else, you'll need to explicitly specify the `contextConfigLocation`.

- The **load-on-startup** parameter indicating that the dispatcher servlet, which handles incoming requests in our MVC app should be loaded with the highest priority load-on-startup is one.
- Servlet Mapping Tag : specifies the URL patterns that'll be handled by the servlets that we have configured.
We have just the one front controller that is the dispatcher servlet named **spring-mvc** and the URL pattern that it handles is simply the `/` indicates that all requests to the root of our web application will be handled by the dispatcher servlet.

4. Prepare The spring-mvc-servlet.xml file



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
8         http://www.springframework.org/schema/mvc
9         http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
10        http://www.springframework.org/schema/context
11        http://www.springframework.org/schema/context/spring-context-4.0.xsd">
12
13     <context:annotation-config/>
14
15     <context:component-scan base-package="com.mvc"/>
16
17     <mvc:annotation-driven />
18
19 </beans>
```

File `spring-mvc-servlet.xml` contains our servlet configuration.

- Tag `<context:annotation-config/>`.
This tag tells Spring MVC that on all beans, controllers, components, repositories, and other objects that are injected into Spring should support **annotations**. So read in and interpret any annotation that is Java annotations that have been applied on these components. Annotation config recognizes general Spring annotations. These are annotations that are not specific to Spring MVC, even if you don't explicitly specify the annotation config tag, Spring will assume that it is present. The component-scan tag on line 15 is something that we're familiar with.
- Tag `<context:component-scan base-package="com.mvc" />`.
This tells Spring MVC to scan all packages starting from the base package that we have specified, looking for objects that are injectable into the Spring framework. As long as the components that you've annotated using Spring MVC annotations belong to

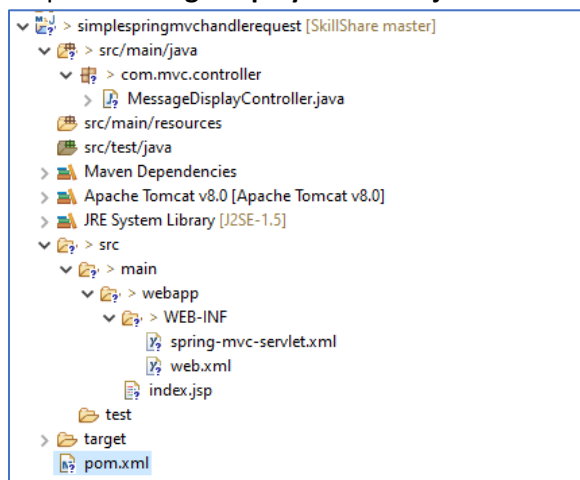
this base package, they will be scanned, their objects instantiated and injected into your application. It doesn't matter how many controllers you have or how many components you've set up, all of them will be instantiated and injected.

- Tag `<mvc:annotation-driven />`

Tag annotation-driven tells Spring to enable Spring MVC specific annotations such as `@Controller`.

The annotation-driven XML tag indicating classes with Spring MVC annotation should be recognized by Spring app. It does not explicitly have the annotation-config XML tag here.

5. Prepare **MessageDisplayController.java** class under **src/main/java/com/mvc/controller**



```
MessageDisplayController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.servlet.ModelAndView;
7
8 @Controller
9 public class MessageDisplayController {
10
11     @RequestMapping(value = "/hello")
12     public ModelAndView sayHello(Model model) {
13         model.addAttribute("message", "Hello there!");
14         return new ModelAndView("index.jsp");
15     }
16
17     @RequestMapping(value = "/start")
18     public ModelAndView startLearning(Model model) {
19         model.addAttribute("message", "Let's start learning");
20         return new ModelAndView("index.jsp");
21     }
22
23 }
```

The MessageDisplayController class is tag with the **@Controller** annotation. And thanks to the way we've specified our servlet configuration, this class will be scanned and injected into our Spring MVC app.

We have two handler mappings here for the **/hello** and the **/start** paths.

Now, if you remember in none of our configuration files have we specified a view resolver which means we cannot work with logical views in this particular application. And that is why both of these handler methods return physical views **index.jsp**.

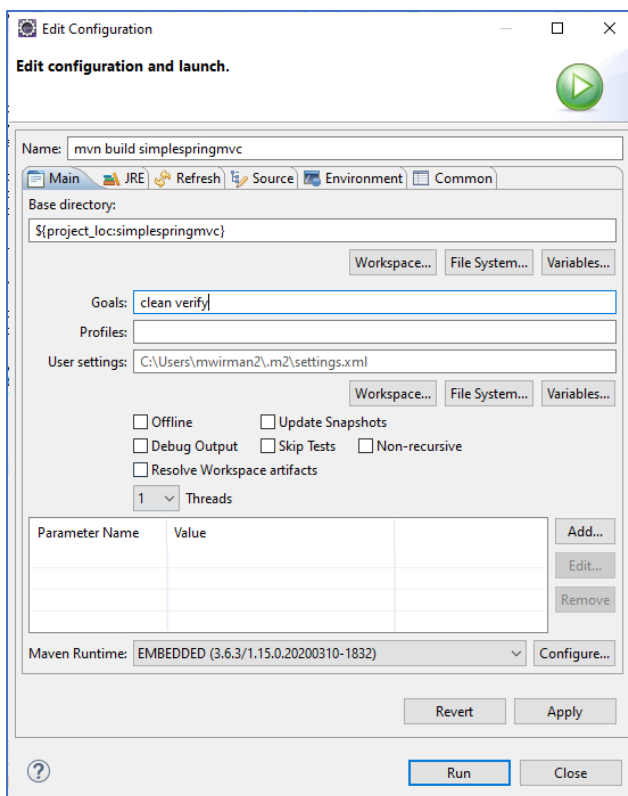
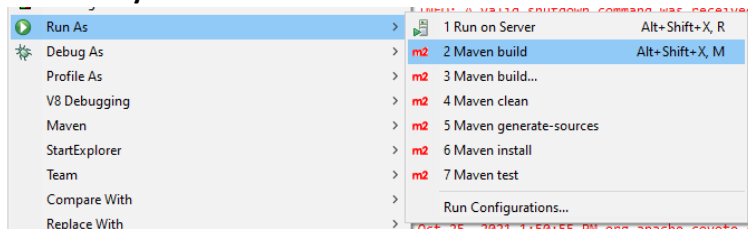
`model.addAttribute("message", "Hello there!");`

In both of these handler methods, after adding the message attribute, we map to a physical view, the ModelAndView objects in both cases return physical views.

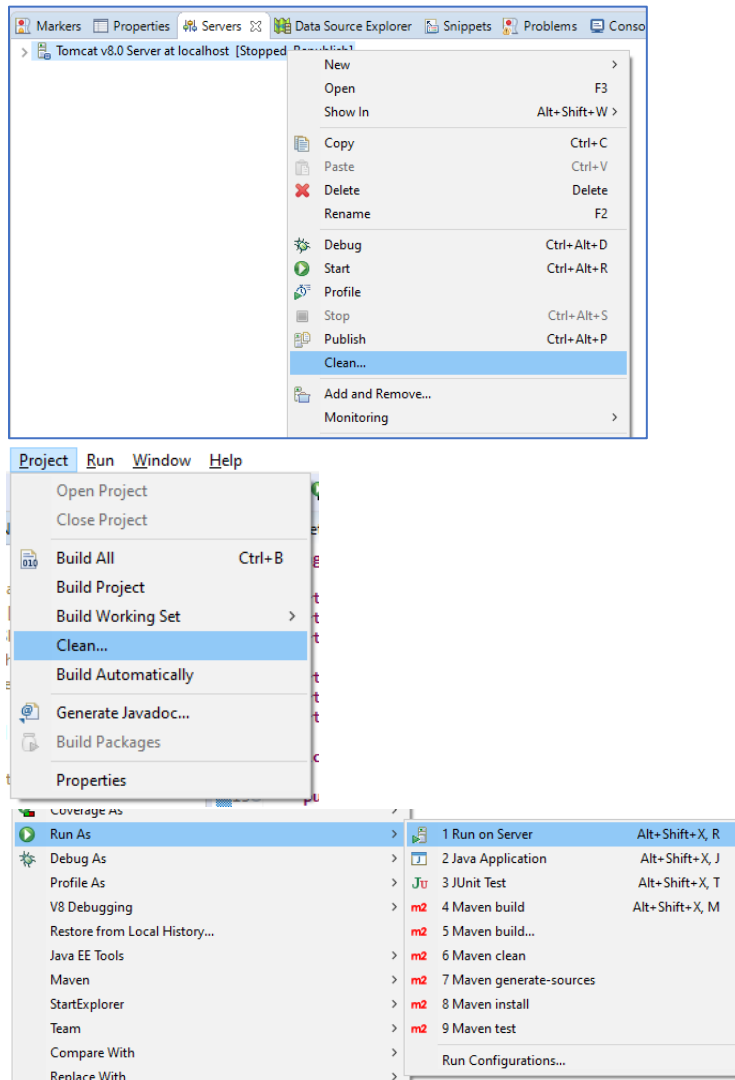
This is not a best practice, you should always work with logical views in your code. But here I wanted to show you that it's possible to map to physical views and it can be done. This is all the code there is in our app.

6. Build the Maven WAR file and run this on our Tomcat server.

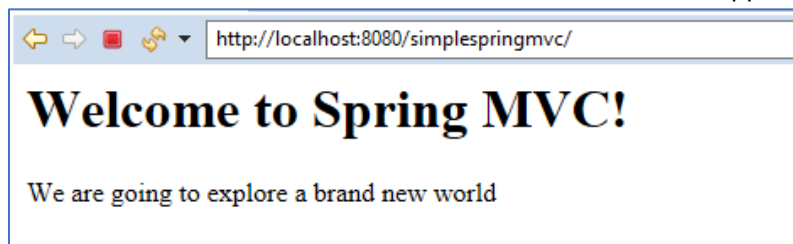
Right click and Run As and choose the Maven build option. Within the Goals of this dialog, specify **clean verify**.



Stop and Clean the Tomcat Cache



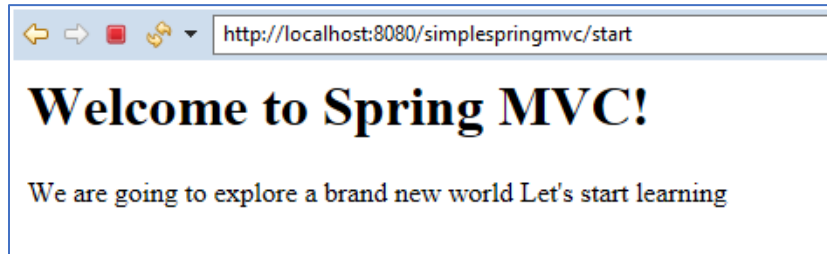
Let's take a look at the browser. Here's the root of our web application `simplespringmvc/`.



Let's go to the `/hello` path relative to the root and you can see the message says Hello there! Our message variable has been correctly picked up.

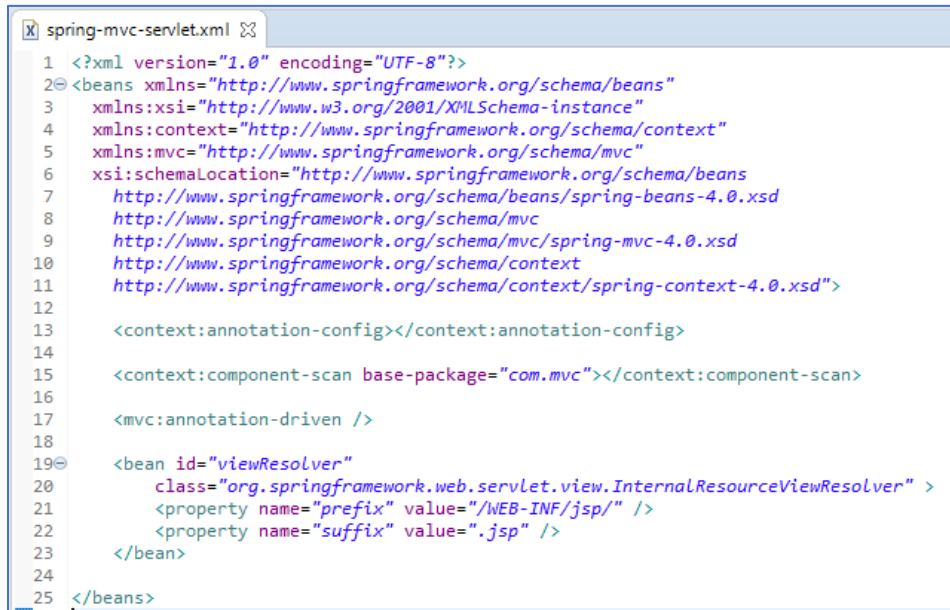


Let's go to another relative path the /start path and this maps to a different handler method and the message now says, Let's start learning.



Configuring Multiple Pages

1. Specified the viewResolver within servlet XML specification.



```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
8         http://www.springframework.org/schema/mvc
9         http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
10        http://www.springframework.org/schema/context
11        http://www.springframework.org/schema/context/spring-context-4.0.xsd">
12
13   <context:annotation-config/>
14
15   <context:component-scan base-package="com.mvc"/>
16
17   <mvc:annotation-driven />
18
19   <bean id="viewResolver"
20         class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
21     <property name="prefix" value="/WEB-INF/jsp/" />
22     <property name="suffix" value=".jsp" />
23   </bean>
24
25 </beans>

```

Adding new Bean Tag **viewResolver** bean in application context, which maps to the **InternalResourceViewResolver**. Takes care of *mapping the logical view names* that we'll use within our code, to *actual view* implementations.

Notice **prefix property** here, it's set to **/WEB-INF/jsp/**.

This property tells our Spring MVC application that all of our views, live within the WEB-INF folder, and the jsp subfolder under it. That's where it should look for, to find actual view implementations.

The **suffix property** is set to **.jsp** as before, all of views are .jsp pages.

2. Update the contain of index.jsp file under webapps as follows



```

1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <html>
4 <head>
5   <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10  <h1>Spring MVC!</h1>
11
12  <a href="hello">Hello Page</a><p></p>
13  <a href="start">Start Page</a>
14 </body>
15 </html>

```

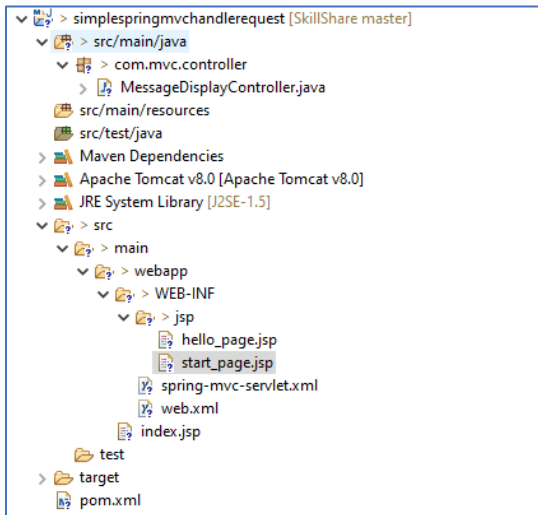
Notice that now there are two a href links, these are anchor tags.

On line 12, we can see that the text, `Hello Page<p></p>` is a link to a page which has the relative URL, Hello. The href here, refers to a relative URL, relative to the root of the application.

On line 13, we have the text Start Page, `Start Page` which is a reference to the relative URL, Start, and this maps to the handler/start that we will set up in our controller.

We'll wire up our controller, such that when users click on the hello page link, they will be taken to hello_page.jsp. Which will simply say Hello there, we are chugging along with Spring MVC.

3. Prepare **home_page.jsp** and **start_page.jsp** files under folder /WEB-INF/jsp



```
hello_page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>Hello Page</title>
8 </head>
9 <body>
10 <h2>Hello there! We're chugging along with Spring MVC</h2>
11 </body>
12 </html>
```

```
start_page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>Start Page</title>
8 </head>
9 <body>
10 <h2>Spring MVC, yippee! Let's start learning</h2>
11 </body>
12 </html>
```

Both the start_page, and hello_page jsp files are under the WEB-INF folder /jsp. This is the prefix that we had specified for all of our view implementations, for the internal view resolver.

4. Update MessageDisplayController.java file

Mapping an incoming request to a path, to a particular view that needs to be rendered is done by the controller, the **MessageDisplayController** is in the **com.mvc.controller** package as before.

```
MessageDisplayController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5
6 @Controller
7 public class MessageDisplayController {
8
9     @RequestMapping("/hello")
10    public String sayHello() {
11        return "hello_page";
12    }
13
14    @RequestMapping("/start")
15    public String startLearning() {
16        return "start_page";
17    }
18
19 }
```

The controller class is tag with the **@Controller** annotation, so that it's injected into Spring app. There are two handler mapping specifications within this controller.

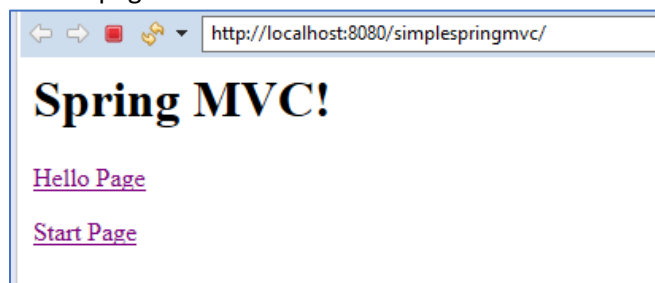
1. RequestMapping to the **/hello** path. **@RequestMapping("/hello")**.
This is where the Hello page will go to, notice that the method simply returns a string "hello_page", which is the *logical name of the view* **hello_page.jsp** that needs to be rendered, in a response to a request made to this path.
2. RequestMapping to the **/start** path. **@RequestMapping("/start")**,
when a request comes in for this path, we will render the Start page logical view, which maps to **start_page.jsp**.

5. Test and Run the App

This completes the setup of simple spring mvc application, which has multiple pages that are rendered, in response to different requests.

follows are the output:

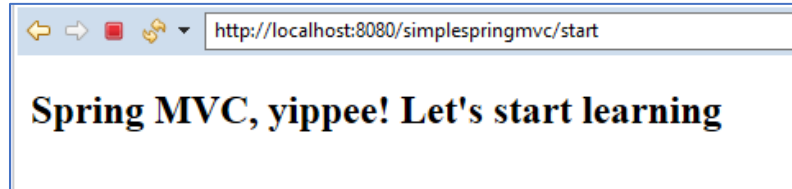
- default page:



- hello page:



- start page :

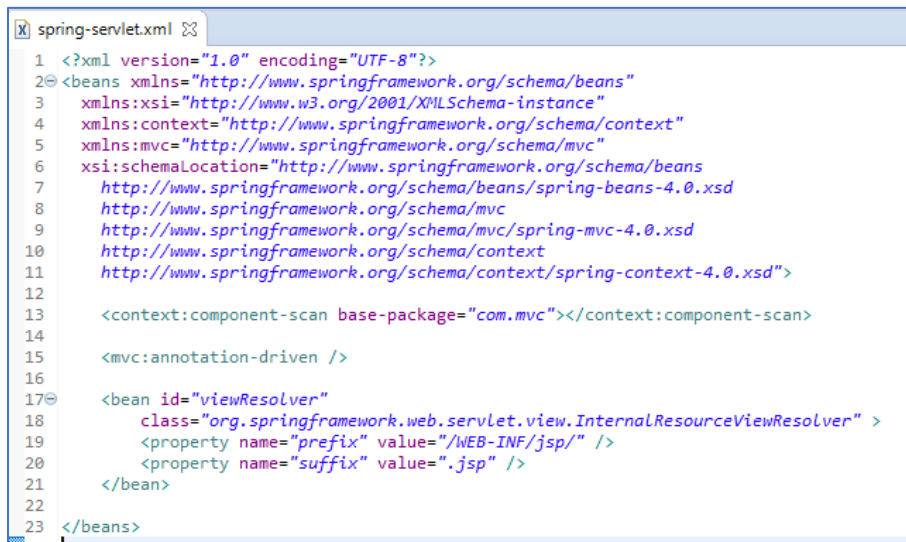
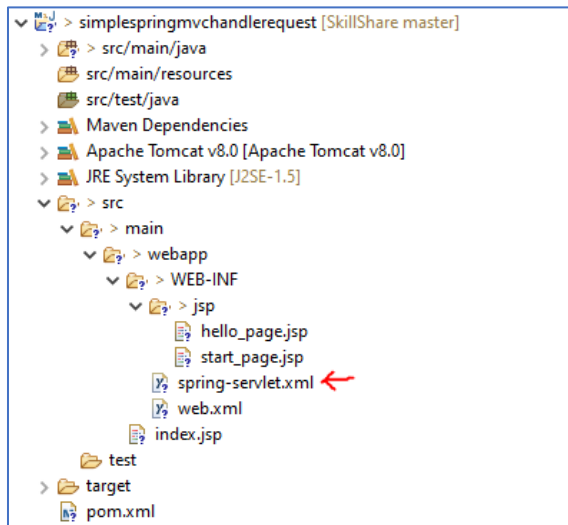


Configure Multiple Controller

Demonstrate how to have multiple controllers to handle multiple request in Spring MVC application

1. Renamed *spring-mvc-servlet.xml* to *spring-servlet.xml*

It doesn't matter how you name your servlet's specification file, so long as you use the **suffix servlet** (e.g : [any_file_name-]servlet.xml) and you place it in under the WEB-INF folder. This is the default location where Spring MVC will look for your servlet specification XML based files.



The content of this spring-servlet.xml have

- The context **component-scan** tag base-package set to **com.mvc**.

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

As long as the components that annotated using Spring MVC annotations belong to this base package, they will be scanned, their objects instantiated and injected into Spring

application. It doesn't matter how many controllers you have or how many components you've set up, all of them will be instantiated and injected.

- It also have the **annotation-driven XML tag** indicating classes with Spring MVC annotation should be recognized by our app.

```
<mvc:annotation-driven/>
```

- It is **NOT** explicitly have the **annotation-config** XML tag.

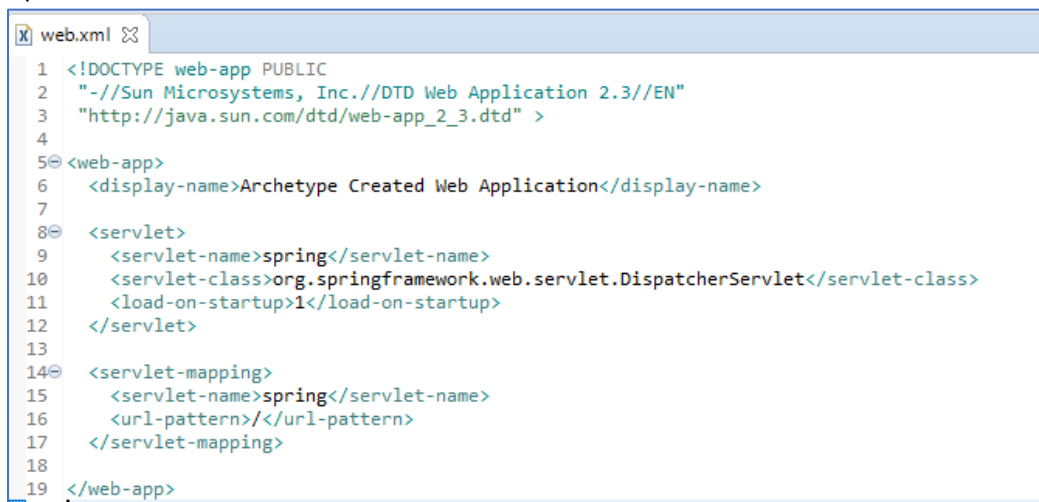
```
<context:annotation-config></context:annotation-config>
```

The annotation-driven XML tag kind of subsumes the annotation-config tag.

- The bean tag **viewResolver** to map my logical view names to physical JSP view implementations.

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

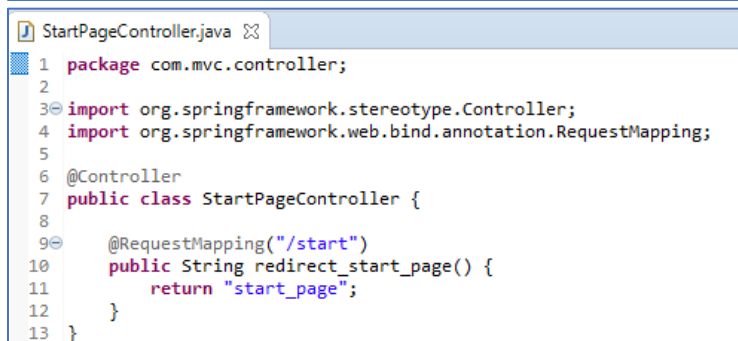
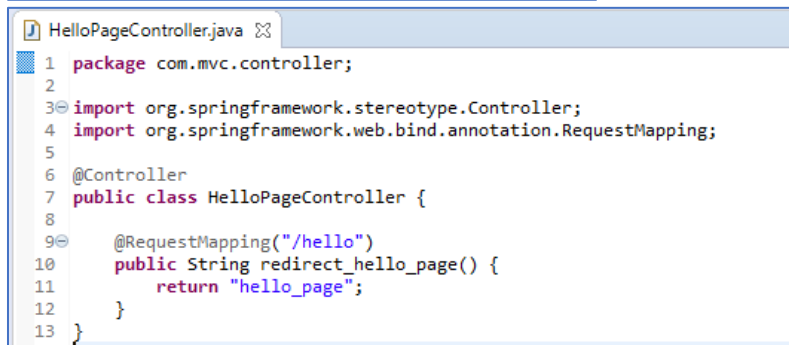
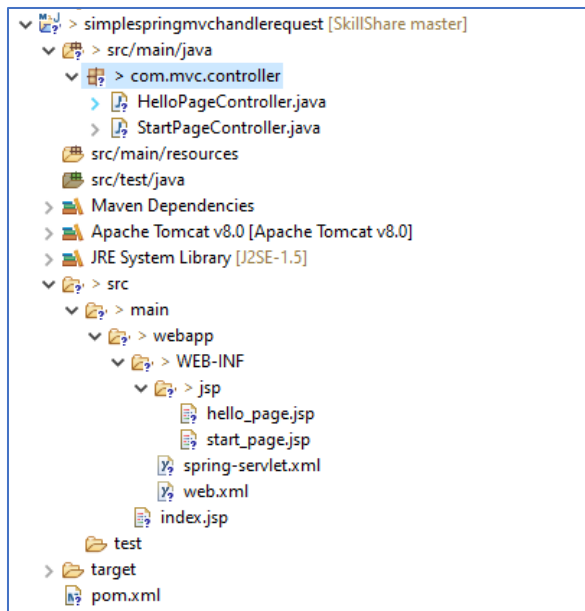
2. Update the Contains of web.xml file as follows



- Specify the DispatcherServlet that is the front controller for Spring MVC application. and set the servlet loaded on startup with the highest priority.
- The contextConfigLocation initialization parameter is not declared, it is because it completely optional, Spring MVC will automatically look within the WEB-INF folder all files that contain servlets specific configuration, that have to have the suffix **-servlet**.
- Make sure that this servlet maps to the url-pattern **/**.

3. Prepare 2 New Controller HelloPageController.java and StartPageController.java Existing MessageDisplayController.java need to be removed and replace with these added 2 new files instead.

Spring MVC Framework : Handling Request & Errors

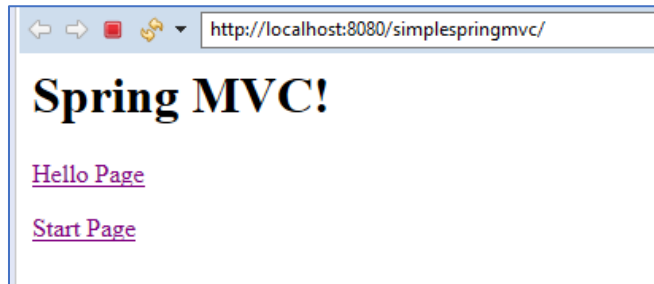


4. Test and Run the App

This completes the setup of simple spring mvc application, which has multiple pages that are rendered from different controller, in response to different requests.

follows are the output:

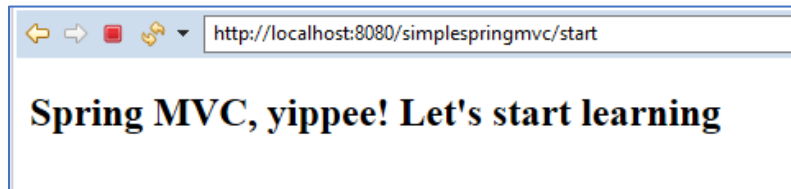
- default page:



- hello page:



- start page :



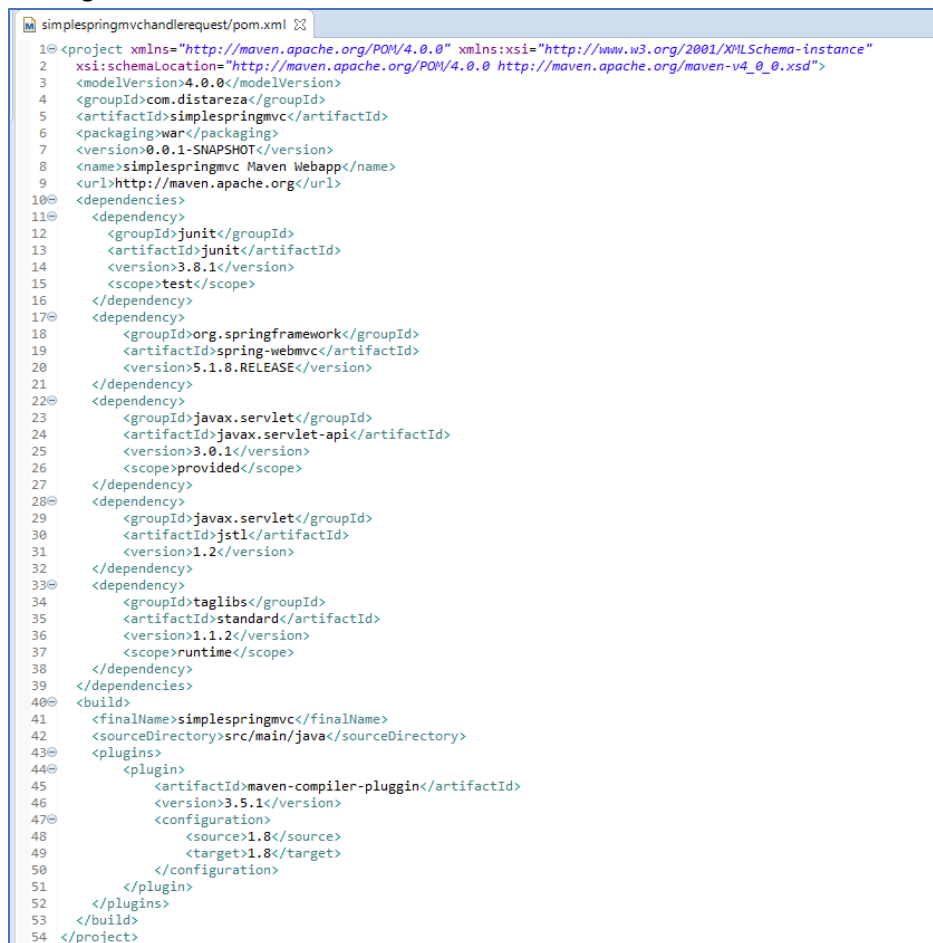
Designing 3-Tier Application

Simulate the 3-tier architecture to build an application using Spring MVC

Spring MVC 3-tier application :

1. Data Access Layer : interacts with database and data objects.
2. Service Layer or Business Logic layer : interacts with the data access layer and adds an additional business logic.
3. Presentation Layer : interacts with the business logic layer, and then renders the view to user.

1. Prepare the Pom with the required dependency : spring webmvc framework, java servlet api, jstl and tag libs.



```
1<?xml version="1.0" encoding="UTF-8"?>
2<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4<modelVersion>4.0.0</modelVersion>
5<groupId>com.distareza</groupId>
6<artifactId>simplespringmvc</artifactId>
7<packaging>war</packaging>
8<version>0.0.1-SNAPSHOT</version>
9<name>simplespringmvc Maven Webapp</name>
10<url>http://maven.apache.org</url>
11<dependencies>
12<dependency>
13<groupId>junit</groupId>
14<artifactId>junit</artifactId>
15<version>3.8.1</version>
16<scope>test</scope>
17</dependency>
18<dependency>
19<groupId>org.springframework</groupId>
20<artifactId>spring-webmvc</artifactId>
21<version>5.1.8.RELEASE</version>
22</dependency>
23<dependency>
24<groupId>javax.servlet</groupId>
25<artifactId>javax.servlet-api</artifactId>
26<version>3.0.1</version>
27<scope>provided</scope>
28</dependency>
29<dependency>
30<groupId>javax.servlet</groupId>
31<artifactId>jstl</artifactId>
32<version>1.2</version>
33</dependency>
34<dependency>
35<groupId>taglibs</groupId>
36<artifactId>standard</artifactId>
37<version>1.1.2</version>
38<scope>runtime</scope>
39</dependency>
40</dependencies>
41<build>
42<finalName>simplespringmvc</finalName>
43<sourceDirectory>src/main/java</sourceDirectory>
44<plugins>
45<plugin>
46<artifactId>maven-compiler-plugin</artifactId>
47<version>3.5.1</version>
48<configuration>
49<source>1.8</source>
50<target>1.8</target>
51</configuration>
52</plugin>
53</plugins>
54</build>
55</project>
```

note *) please update project dependencies on eclipse by *right click* on pom.xml and choose **"Maven > Update Project"** every time after modify the pom.xml.

2. Prepare web.xml version 3.0

Specify web.xml app configuration with version 3.0 as follows

```
web.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://java.sun.com/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
5     id="WebApp_ID"
6     version="3.0" >
7
8     <display-name>Spring MVC</display-name>
9
10    <servlet>
11        <servlet-name>spring</servlet-name>
12        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
13        <load-on-startup>1</load-on-startup>
14    </servlet>
15
16    <servlet-mapping>
17        <servlet-name>spring</servlet-name>
18        <url-pattern>/</url-pattern>
19    </servlet-mapping>
20
21 </web-app>
```

With Java EE annotations, the standard web.xml deployment descriptor is optional. With the servlet 3.0 specification, annotations can be defined on certain Web components, such as servlets, filters, listeners, and tag handlers. The annotations are used to declare dependencies on external resources.

note : JSTL would not work on Servlet 2.3 or older, You should also make sure that your web.xml is declared conform at least Servlet 2.4 or above. Otherwise EL expressions inside JSTL tags would in turn fail to work. Pick the highest version matching your target container.

Tomcat version 7 and above are compatible with Servlet 3.0 specification.

3. Prepare the spring-servlet.xml

```
spring-servlet.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xmlns:mvc="http://www.springframework.org/schema/mvc"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
8         http://www.springframework.org/schema/mvc
9         http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
10        http://www.springframework.org/schema/context
11        http://www.springframework.org/schema/context/spring-context-4.0.xsd">
12
13    <context:component-scan base-package="com.mvc"></context:component-scan>
14
15    <mvc:annotation-driven />
16
17    <bean id="viewResolver"
18        class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
19        <property name="prefix" value="/WEB-INF/jsp/" />
20        <property name="suffix" value=".jsp" />
21    </bean>
22
23 </beans>
```

4. Prepare Model Layer, Student Class

```
Student.java
1 package com.mvc.model;
2
3 import java.io.Serializable;
4
5 public class Student implements Serializable {
6
7     private static final long serialVersionUID = 1L;
8
9     private Long id;
10    private String firstName;
11    private String lastName;
12    private String major;
13
14    public Student(String firstName, String lastName, String major) {
15        this.id = (long) Math.floor(Math.random() * 1000000);
16        this.firstName = firstName;
17        this.lastName = lastName;
18        this.major = major;
19    }
20
21    public Long getId() {
22        return id;
23    }
24
25    public void setId(Long id) {
26        this.id = id;
27    }
28
29    public String getFirstName() {
30        return firstName;
31    }
32
33    public void setFirstName(String firstName) {
34        this.firstName = firstName;
35    }
36
37    public String getLastName() {
38        return lastName;
39    }
40
41    public void setLastName(String lastName) {
42        this.lastName = lastName;
43    }
44
45    public String getMajor() {
46        return major;
47    }
48
49    public void setMajor(String major) {
50        this.major = major;
51    }
52
53    @Override
54    public String toString() {
55        return String.format("{ firstName = %s, lastName= %s, major= %s }", this.firstName, this.lastName, this.major);
56    }
57 }
58
```

5. Prepare DAO Layer and its Interface, StudentDAO and StudentDAOImpl Class

```
StudentDAO.java
1 package com.mvc.dao;
2
3 import java.util.List;
4
5
6 public interface StudentDAO {
7
8     public List<Student> getStudents();
9
10 }
11 }
```

```
StudentDAOImpl.java
1 package com.mvc.dao;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 import org.springframework.stereotype.Repository;
7
8 import com.mvc.model.Student;
9
10 @Repository
11 public class StudentDAOImpl implements StudentDAO {
12
13     public List<Student> getStudents() {
14         List<Student> studentList = new ArrayList<Student>();
15
16         studentList.add(new Student("Greg", "Johnson", "Computer Science"));
17         studentList.add(new Student("Alice", "Kruger", "Philosophy"));
18         studentList.add(new Student("Peter", "Smith", "Math"));
19         studentList.add(new Student("Claudia", "Wallace", "Chemistry"));
20         studentList.add(new Student("Sinead", "O' Connor", "Operations"));
21
22         return studentList;
23     }
24 }
25 }
```

Let's take a look at the implementation of this class. We've overridden the `getStudents` method from the interface and provided an implementation here.

Notice that the DAO Implementation Class is tag with **@Repository** tag. This tag is a Spring MVC specific tag. And it indicates to Spring that this object is an injectable component, which references a repository that is an underlying database. Using this `@Repository` tag not only injects this object into the right places where you need a data access implementation. Any exception thrown by this object is translatable to a generic data access exception.

So if you have objects that query a database, make sure you use the specific `@Repository` tag.

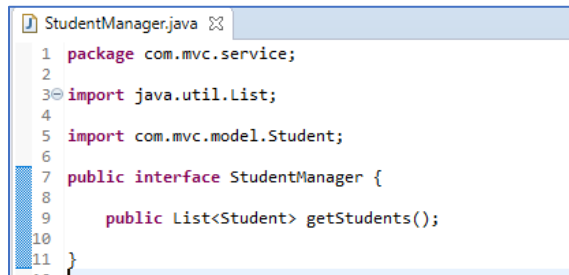
We haven't actually accessed a database, we'll set that up in a later demo. What I've done here is instantiated a list of students and added five students to this list, each of these students belong to different majors.

For our example here, you can assume that this list of students has been retrieved from an underlying database.

6. Prepare Service Logic Layer and its Interface, StudentManager and StudentManagerImpl Class

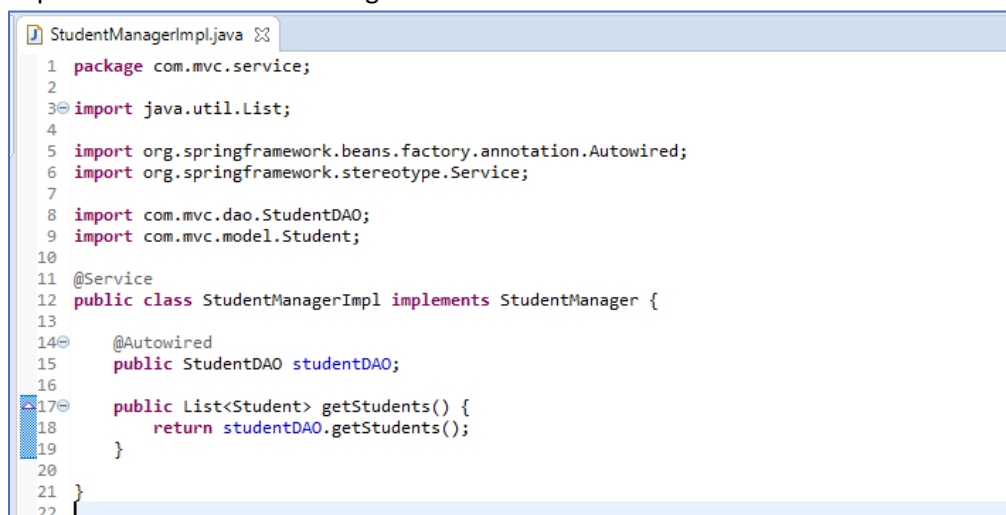
Service Logic Layer is the middle layer in 3-tier architecture, it lies above the data access layer, and this is the layer that contains the business logic. This interface here simply retrieves and returns a list of students. We haven't added any additional details. But you can imagine that the business logic layer may contain other calls to retrieve other information about students from other database objects.

This is the interface

A screenshot of an IDE showing the code for StudentManager.java. The code defines a package, imports List and Student, and declares a public interface StudentManager with a single method getStudents() returning a List of Student objects.

```
1 package com.mvc.service;
2
3 import java.util.List;
4
5 import com.mvc.model.Student;
6
7 public interface StudentManager {
8
9     public List<Student> getStudents();
10
11 }
```

let's take a look at the implementation of this interface called StudentManagerImpl which implements the student manager interface.

A screenshot of an IDE showing the code for StudentManagerImpl.java. The code defines a package, imports List, Autowired, Service, StudentDAO, and Student, and declares a public class StudentManagerImpl that implements StudentManager. It uses @Autowired to inject StudentDAO and implements the getStudents() method by delegating the call to studentDAO.getStudents().

```
1 package com.mvc.service;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 import com.mvc.dao.StudentDAO;
9 import com.mvc.model.Student;
10
11 @Service
12 public class StudentManagerImpl implements StudentManager {
13
14     @Autowired
15     public StudentDAO studentDAO;
16
17     public List<Student> getStudents() {
18         return studentDAO.getStudents();
19     }
20
21 }
22 }
```

Observe the annotation here, it's the **@Service** annotation. The @Service annotation indicates to Spring that this is an *injectable component*. This is a bean whose object should be instantiated and injected into other objects that depend on this object.

The **@Service** annotation doesn't provide any additional functionality for this object besides injecting it as a component. But it specifies the intent of the object better, indicating that it's part of the service layer, which should include your business logic. This service layer that forms the middle tier of a 3-tier architecture accesses the data access layer and then retrieves the data, performs additional business logic and passes the data along to the presentation layer.

So it stands to reason that service layer objects will depend on objects from the data access layer. And you wire up these dependencies automatically using dependency injection. This is the inversion of control that Spring offers.

The **@Autowired** annotation does autowires dependencies together. Inject the data access dependency into this current object that is part of the middle tier, that is the service layer or the business logic layer.

Notice that we've applied this **@Autowired** annotation to the *interface StudentDAO*. Spring will take care of finding the right implementation for this interface and performing the injection. The only bit of code left to explain is this overridden `getStudents` method.

All we do is access the object with the `studentDAO` interface, and call `getStudents`, and the students will be returned to any layer that works with this service layer. This is where you can add additional business logic if you want to.

Maybe you will perform a check to see that the students haven't already graduated, these are all current students. Maybe you will have business logic checks to see that a student is not on a leave of absence. All of that will be included here in the service layer or the business logic layer.

7. Prepare the Controller class as part of Presentation Layer, StudentController Class

```
StudentController.java
1 package com.mvc.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.ui.Model;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RequestMethod;
8
9 import com.mvc.service.StudentManager;
10
11 @Controller
12 public class StudentController {
13
14     @Autowired
15     private StudentManager studentManager;
16
17     @RequestMapping(value = "/university/students", method = RequestMethod.GET)
18     public String getStudentsMapping(Model model) {
19         model.addAttribute("students", studentManager.getStudents());
20         return "students_view";
21     }
22 }
23 }
```

This forms part of our presentation layer which renders the view. As we've seen before, controller classes are tagged using the **@Controller** annotation indicating that this is an injectable component that is part of the presentation layer, the controller.

It has an autowire dependency to the StudentManager, which is our service that is the business logic layer.

```
@Autowired
private StudentManager studentManager;
```

We've specified exactly one handler mapping here within this controller, the value is /university/students.

```
@RequestMapping(value = "/university/students", method = RequestMethod.GET)
```

This is the relative path from the root of our application that will retrieve the list of students. In addition, we've also specified that this handler responds to get request from the user, method is =RequestMethod.Get. Now within this method, we add the list of student objects as the student's attribute.

```
model.addAttribute("students", studentManager.getStudents());
```

And the logical view that we render is called students_view, which is responsible for displaying this list of students.

```
return "students_view";
```

8. Prepare the JSP Pages as part of Presentation Layer, index.jsp and WEB-INF/jsp/students_view.jsp

```

index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <html>
4 <head>
5 <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10 <h1>Spring MVC!</h1>
11
12 <a href="hello">Hello Page</a><p></p>
13 <a href="start">Start Page</a>
14
15 <h1>Fantabulous University Database</h1>
16 <a href="university/students">View Students</a>
17 </body>
18 </html>
19
students_view.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
5 <!DOCTYPE html>
6 <html>
7 <head>
8 <meta charset="ISO-8859-1">
9 <title>Fantabulous University - Students</title>
10 <style type="text/css">
11   table, th, td {
12     border: 2px solid green;
13     border-collapse: collapse;
14     padding: 8px;
15     color: red;
16   }
17 </style>
18
19 </head>
20 <body>
21 <h2>Students of Fantabulous University</h2>
22
23 <table>
24 <tr>
25 <th>First Name</th>
26 <th>Last Name</th>
27 <th>Major</th>
28 </tr>
29 <c:forEach items="${students}" var="student">
30 <tr>
31 <td>${student.firstName}</td>
32 <td>${student.lastName}</td>
33 <td>${student.major}</td>
34 </tr>
35 </c:forEach>
36 </table>
37
38 </body>
39 </html>

```

Notice at the very top, we have an import for the taglib library, which gives us useful tags, allowing us to add control structures to our jsp code.

```

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

```

We've imported the jstl core and the jstl format libraries with the prefix c and fmt respectively. If you scroll down below, I've set up a simple table to display the students that we've retrieved using our app. These are the Students of Fantabulous University, the header contains the First Name, Last Name and Major.

```
<table>
  <tr>
    <th>First Name</th>
    <th>Last Name</th>
    <th>Major</th>
  </tr>
```

And notice that we have a `forEach` loop here, c: `forEach`.

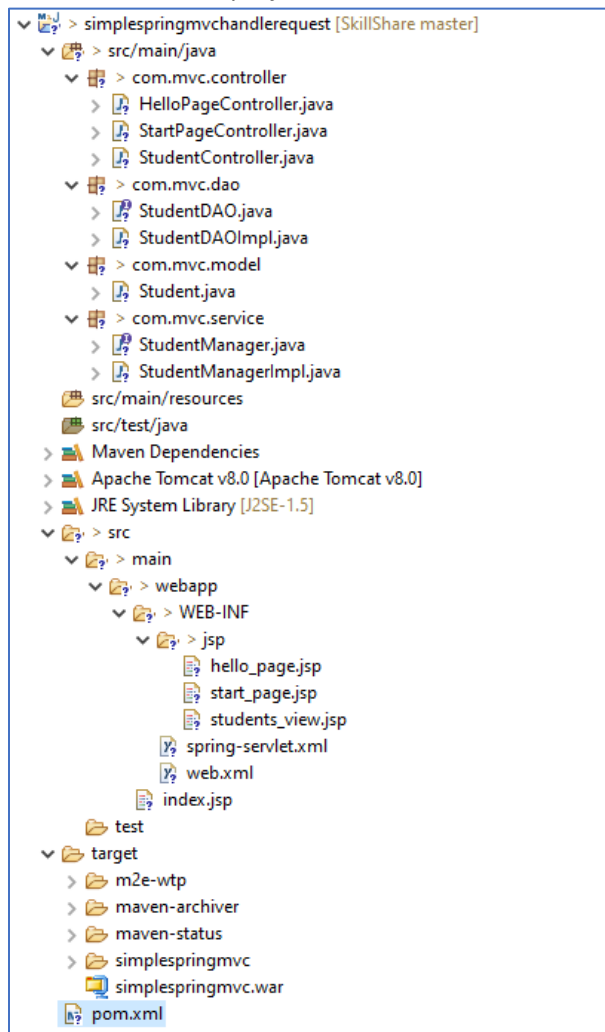
```
<c:forEach items="${students}" var="student">
```

We iterate over all of the student objects that the controller added while rendering this view. We represent each student using the variable `student`, that is `var= "student"`. And for each student, we display the `firstName`, `lastName` and `major`.

```
  <tr>
    <td>${student.firstName}</td>
    <td>${student.lastName}</td>
    <td>${student.major}</td>
  </tr>
```

Notice how we can access the member variables of the student object even though the member variables are private, this is done using reflection. And thus, using JSP and the `taglibs` library, we've referenced the list of students that make up our model and have displayed them to screen.

Here is the overall project structures are look like :

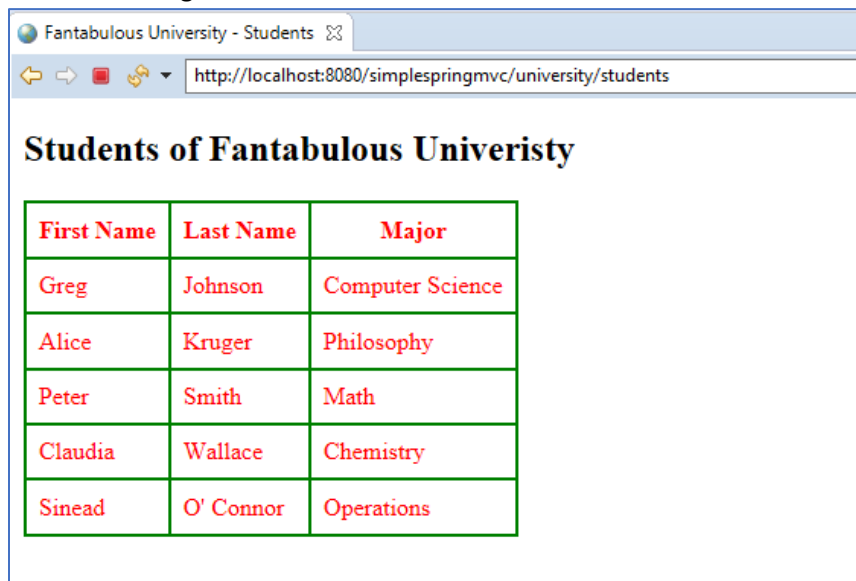


- Let's build our WAR file and display this application using the Tomcat server. When you run this application, this is what you will see first, the Fantabulous University Database. Click through to View Students and the list of students that is retrieved from our data access layer is printed out to screen.

Initial Page :



List student Page :



Extracting Dynamic URL Path

10. Modify “student_view.jsp” to include dynamic anchor tag link as follows

```

students_view.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
4 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
5 <!DOCTYPE html>
6 <html>
7 <head>
8 <meta charset="ISO-8859-1">
9 <title>Fantabulous University - Students</title>
10 <style type="text/css">
11     table, th, td {
12         border: 2px solid green;
13         border-collapse: collapse;
14         padding: 8px;
15         color: red;
16     }
17 </style>
18
19 </head>
20 <body>
21 <h2>Students of Fantabulous Univeristy</h2>
22
23 <table>
24 <tr>
25 <th>First Name</th>
26 <th>Last Name</th>
27 <th>Major</th>
28 </tr>
29 <c:forEach items="${students}" var="student">
30 <tr>
31 <td><a href=".."students/${student.firstName}">${student.firstName}</a></td>
32 <td>${student.lastName}</td>
33 <td>${student.major}</td>
34 </tr>
35 </c:forEach>
36 </table>
37
38 </body>
39 </html>

```

Notice The name of the student is a part of the path.

```
<a href="students/${student.firstName}">${student.firstName}</a>
```

We want to set things up so that the student's name can be extracted from the path in our controller method.

11. Create new Controller and JSP Page to handle the new request for dynamic link

```

StudentController.java
1 package com.mvc.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.ui.Model;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RequestMethod;
9
10 import com.mvc.service.StudentManager;
11
12 @Controller
13 public class StudentController {
14
15     @Autowired
16     private StudentManager studentManager;
17
18     @RequestMapping(value = "/university/students", method = RequestMethod.GET)
19     public String getStudentsMapping(Model model) {
20         model.addAttribute("students", studentManager.getStudents());
21         return "students_view";
22     }
23
24     @RequestMapping(value = "/students/{name}")
25     public String displayStudent(@PathVariable("name") String name, Model model) {
26         String welcomeMessage = String.format("Welcome to %s's home Page", name);
27         model.addAttribute("welcomeMessage", welcomeMessage);
28         return "student_home_page";
29     }
30
31 }

```

The content inside of this the `@Controller` tag class having new `@RequestMapping` tag method as follows

```
@RequestMapping(value = "/students/{name}")
```

The value that we've passed into the `RequestMapping` that is the path that corresponds to this handler is `/students/{name}`. This tells spring mvc that this path is dynamically generated. So any request to `/students/` and then just a single element, the second element is actually the name of a student. Now this name will be available as an input argument to this `display` method.

```
public String displayStudent(@PathVariable("name") String name, Model model)
```

Notice that we've used an `@PathVariable` annotation for the string name input argument. This tells spring mvc that the value for this input argument is available from the dynamically created path. It should be extracted from the path and passed in as an input argument.

Now, within the `@PathVariable` annotation, we've specified the name of the property that is name. This should map to the property name that we have defined in the value mapping. In our mapping, we have `/students/{name}`. This name should be same as the property that we have specified to the `PathVariable`.

In addition to the name that is extracted from the dynamically created path, there is an additional input argument to this `display` method and that is the `Model`. The `Model` here is a reference to the model of our spring mvc application, which we can use to access state.

This `Model` in the mvc application can also be used to specify attributes that will be rendered by our presentation view, that is our view pages. Now within this handler method, I'm going to construct a welcome message that includes the name of the student whose page this particular view is.

Welcome to name's home page. The name variable will get its value from the dynamically created path.

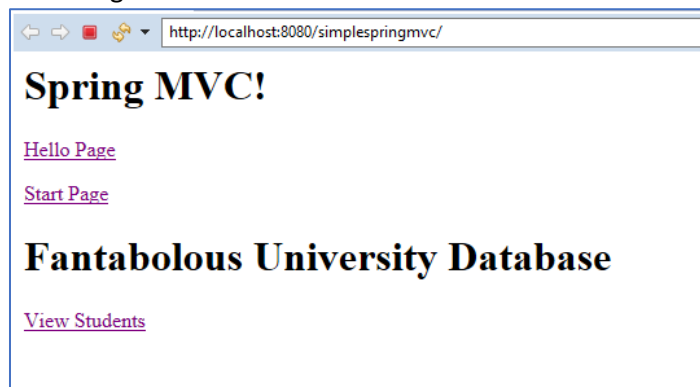
```
String welcomeMessage = String.format("Welcome to %s's home Page", name);
```

I'll add this welcome message to our model and then render the student_home_page.

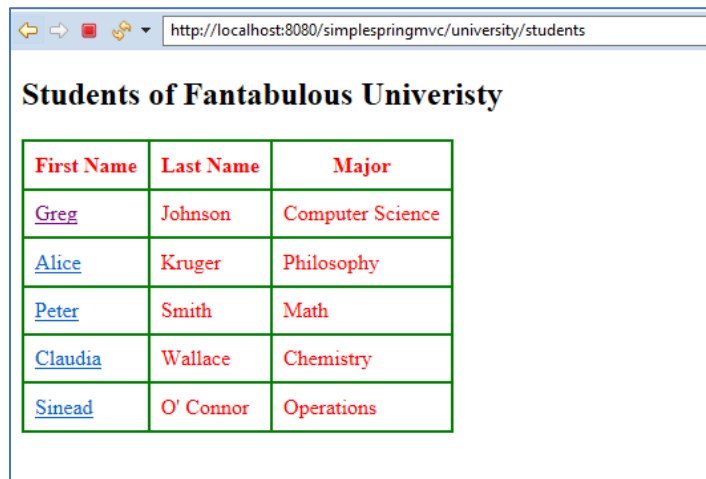
```
student_home_page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Students Home Page</title>
8   <style>
9     h2 {
10       font-size: 44px;
11       padding: 8px;
12       color: lilac;
13     }
14   </style>
15 </head>
16 <body>
17   <h2>${welcomeMessage}</h2>
18 </body>
19 </html>
```

12. Run and Test the code

- Initial Page :

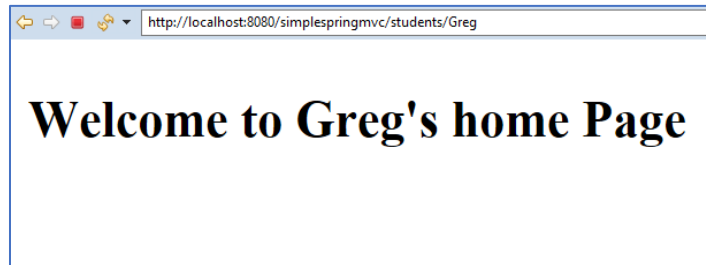


- Click View Students



First Name	Last Name	Major
Greg	Johnson	Computer Science
Alice	Kruger	Philosophy
Peter	Smith	Math
Claudia	Wallace	Chemistry
Sinead	O' Connor	Operations

- Click one of students link (example Greg)



Take a look at the URL here **/students/Greg** is the path to Greg's homepage. The name of the student, Greg, is also extracted into the name variable in our handler method. and we've used this name variable to construct this welcome message, *Welcome to Greg's home page*.

Accessing Request Parameter

Follows is to demonstrate how Spring MVC to extract parameters from the input request.

1. Prepare New JSP file, registration.jsp

```
index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <html>
4 <head>
5   <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10   <h1>Spring MVC!</h1>
11
12   <a href="hello">Hello Page</a><p></p>
13   <a href="start">Start Page</a>
14
15   <h1>Fantabulous University Database</h1>
16   <a href="university/students">View Students</a>
17
18   <h1>Greeting</h1>
19   <a href="registration.jsp">Greeting</a>
20
21 </body>
22 </html>
```

```
registration.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Registration</title>
8 </head>
9 <body>
10   <form action="greeting">
11     Enter your name : <input type="text" name="name"/><br><br>
12     <input type="submit" name="submit">
13   </form>
14
15 </body>
16 </html>
```

Notice the form tag, The action attribute that we have specified for this form is, **hello**, the action attribute tells our view page where to send the form data when the Submit button of the form is clicked.

```
<form action="hello">
```

This form data, will be sent as a **POST** request to the handler method that handles the relative path **/hello**. The contents of our form that we'll use to pass data onto the server is very simple. It consists of a single input text box, specifying the name of an individual.

```
<input type="text" name="name"/>
```

The input type = "text", the name of this text box is "**name**".

```
<input type="submit" name="submit">
```

We then have the form, submit button, the type of input that is, is "submit" and name = "submit".

2. Prepare the Controller Class, GreetingController.java

```
GreetingController.java
1 package com.mvc.controller;
2
3 import javax.servlet.http.HttpServletRequest;
4
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.Model;
7 import org.springframework.web.bind.annotation.RequestMapping;
8
9 @Controller
10 public class GreetingController {
11
12     @RequestMapping("/greeting")
13     public String greeting(HttpServletRequest request, Model model) {
14         String name = request.getParameter("name");
15         String msg = String.format("Hello %s", name);
16         model.addAttribute("message", msg);
17         return "greeting_page";
18     }
19
20 }
```

The GreetingsController, where we specified a handler method, to handle form submission. The GreetingsController is tagged using the **@Controller** annotation

We have the greeting method, tagged using **@RequestMapping**. The path is **/greeting**.

The submission of a form, makes an HTTP POST request to a server. Now, when we don't specify an explicit method for our RequestMapping, it accepts all HTTP requests, whether it's a GET request, a POST request, PUT request or a DELETE request. The input arguments that will be injected when this greeting handler method is invoked on our application server, is the HttpServletRequest object, and the model for our application.

```
String name = request.getParameter("name");
```

The HttpServletRequest object will contain the request parameters from the client. Once the request object is available within our code, we can access the values corresponding to the request parameters using **request.getParameter()**.

```
String msg = String.format("Hello %s", name);
model.addAttribute("message", msg);
```

This message, constructed using a parameter specified by the user, we then add to the model corresponding to the message key.

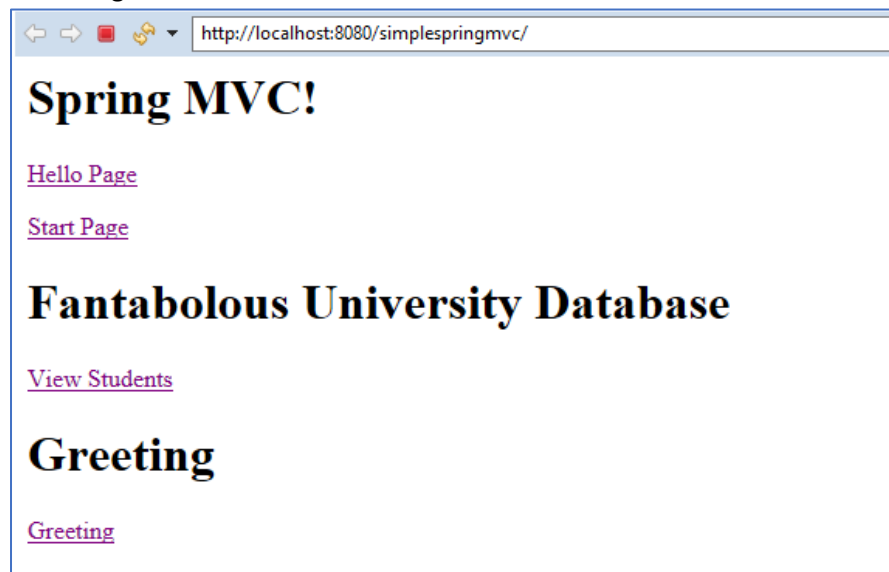
3. Prepare the JSP Class, greeting_page.jsp

```
greeting_page.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>Greeting Page</title>
8 </head>
9 <body>
10 <h2>${message}</h2>
11 </body>
12 </html>
```

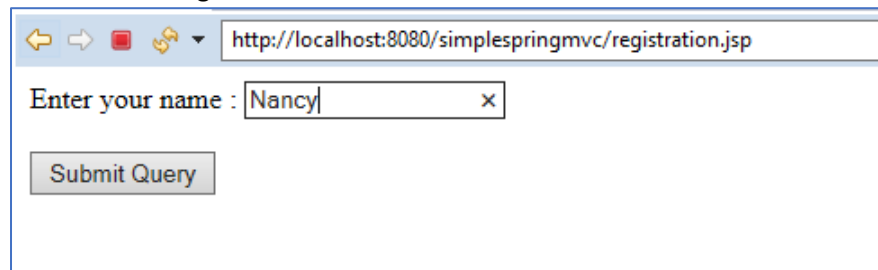
This greetingPage, is a logical view that will render a jsp page, with this greeting on to screen. This is what greetingPage.jsp looks like. We just have an h2 header, rendering out the message sent down from the server.

4. Run and Test the Code

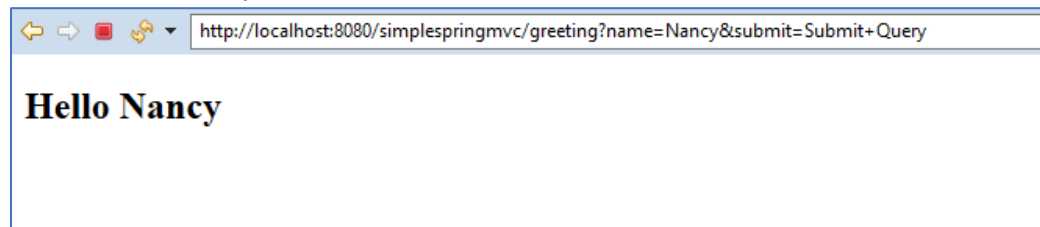
- Initial Page



- Click on Greeting



- Enter text on Enter your name Textbox and Click Submit



the relative path to the URL of this page includes **/greeting**, this was specified by the form action attribute, which was set to hello. And this mapped to our handler method specification in our controller. After the path to the page, we have a question mark, and then we have two parameters that have been passed to the server, **name = "Nancy"** and **submit = "Submit+Query"**.

These are the two parameters which are specified as inputs in our form. The input type text which is equal to "name" and the input type "submit". Now, this request URL was mapped to our GreetingsController, the **RequestMapping("/greeting")**. The `HttpServletRequest` injected, contain the value of the parameter name, that was extracted from the URL. Our controller then constructed a message using this request parameter, and that message, Hello Nancy, is what is displayed here to screen.

Injecting Request Parameter

Demonstrate how we can access request parameters from the incoming request using the `@RequestParam` annotation.

5. Update Greeting Controller

```
GreetingController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestParam;
7
8 @Controller
9 public class GreetingController {
10
11     @RequestMapping("/greeting")
12     public String greeting(@RequestParam String name, Model model) {
13         String msg = String.format("Hello %s", name);
14         model.addAttribute("message", msg);
15         return "greeting_page";
16     }
17
18 }
```

Previous Code :

```
@RequestMapping("/greeting")
public String greeting(HttpServletRequest request, Model model) {
    String name = request.getParameter("name");
    String msg = String.format("Hello %s", name);
    return "greeting_page";
}
```

The method is called `greeting`, and the input argument no longer contains an HTTP request. Instead, we inject the parameters specified in the HTTP request directly as input arguments to this `greeting` method.

```
public String greeting(@RequestParam String name, Model model)
```

Notice the `@RequestParam` annotation that we've applied to the `name` input argument. The fact that this variable is called `name` is important here. We haven't specified what request parameter should be extracted and what value should be present in `name`. Now, the key is picked up using the name of the variable.

The key here is **name**, the value associated with the `name` key is what will be injected into the `String name` variable. We'll construct a new message which says `Hello + whatever is in the name variable`. We'll add this message to the model, and render the `greetingPage`, which will reference this message and print it out to screen.

6. Update registration.jsp file to accept multi input as follows

```

1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2  pageEncoding="ISO-8859-1"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="ISO-8859-1">
7  <title>Registration</title>
8  </head>
9  <body>
10 <form action="greeting">
11     Enter your first name : <input type="text" name="firstName"/><br><br>
12     Enter your last name : <input type="text" name="lastName"/><br><br>
13     Enter your major : <input type="text" name="major"/><br><br>
14     <input type="submit" name="submit">
15 </form>
16
17 </body>
18 </html>

```

Notice this form here, there are multiple input boxes, all of type *text*. The name of the first input box is **firstName**, the second one is **lastName**, and last one is the **major**. I also have a submit button that allows me to submit this form.

The form action is mapped to the request path **greeting**, which is where the form data will be submitted.

7. Update Greeting Controller again as follows

```

1  package com.mvc.controller;
2
3  import org.springframework.stereotype.Controller;
4  import org.springframework.ui.Model;
5  import org.springframework.web.bind.annotation.RequestMapping;
6  import org.springframework.web.bind.annotation.RequestParam;
7
8  @Controller
9  public class GreetingController {
10
11     @RequestMapping("/greeting")
12     public String greeting(
13         @RequestParam("firstName") String name,
14         @RequestParam("lastName") String lastName,
15         @RequestParam String major,
16         Model model) {
17         String msg = String.format("Hello %s %s, welcome to your major :%s ", name, lastName, major);
18         model.addAttribute("message", msg);
19         return "greeting_page";
20     }
21
22 }

```

In order to be able to accept multi request parameters, we need to update our GreetingsController as well. We have the greeting method as we did before, tagged using the **@RequestMapping** annotation to **/greeting**.

Notice that there are now Four input arguments to this greeting method.

The first two are request parameters. The first input argument name extracts the value associated with the **firstName** request parameter. We've specified **firstName** as a property of the **@RequestParam** annotation.

The last input argument, **major**, is also tagged with the **@RequestParam** annotation, but there is no property specification.

The request parameter that'll be extracted and placed into this variable will be the same as the name of the variable, the major request parameter. This is the value specified in the input text box with the name *major*. The **@RequestParam** annotation will automatically extract the values of the request parameters, and pass them as input to our handler method, which will then construct a message, hello name, welcome to your major, that is the major.

8. Run and Test the Code

The image shows two browser windows. The top window displays a registration form at `http://localhost:8080/simplespringmvc/registration.jsp`. The form contains three text input fields: "Enter your first name :" with the value "Nancy", "Enter your last name :" with the value "Adams", and "Enter your major :" with the value "Economics". Below these fields is a "Submit Query" button. The bottom window shows the result of the submission at `http://localhost:8080/simplespringmvc/greeting?firstName=Nancy&lastName=Adams&major=Economics&`. The page displays the message "Hello Nancy Adams, welcome to your major :Economics" in a bold, black, serif font.

Configuring Request Parameters

By default, all parameters that pass to spring controller method are mandatory, follows code demonstrate how to make some request Parameters as Optional

```
GreetingController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.ui.Model;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestParam;
7
8 @Controller
9 public class GreetingController {
10
11     @RequestMapping("/greeting")
12     public String greeting(
13         @RequestParam("firstName") String name,
14         @RequestParam(name = "lastName", required = false) String lastName,
15         @RequestParam String major,
16         Model model) {
17         String msg = String.format("Hello %s %s, welcome to your major :%s ", name, lastName, major);
18         model.addAttribute("message", msg);
19         return "greeting_page";
20     }
21 }
22 }
```

`@RequestParam(name = "lastName", required = false) String lastName`

This tells Spring MVC that this parameter need not be specified when this particular handler method is being accessed. It's not a required parameter

To setup a default value of incoming parameters you can use as follows

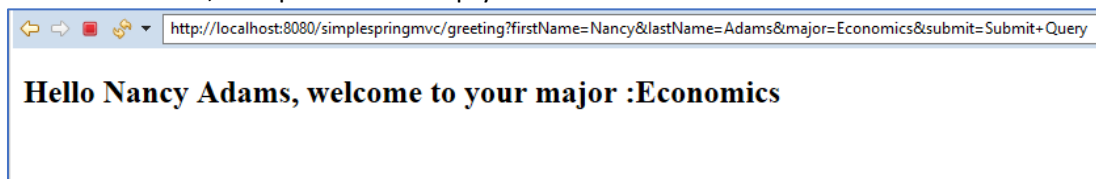
```
GreetingController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class GreetingController {
7
8     @RequestMapping("/greeting")
9     public String greeting(
10         @RequestParam("firstName") String name,
11         @RequestParam String lastName,
12         @RequestParam(defaultValue = "English") String major,
13         Model model) {
14         String msg = String.format("Hello %s %s, welcome to your major :%s ", name, lastName, major);
15         model.addAttribute("message", msg);
16         return "greeting_page";
17     }
18 }
19 }
```

`@RequestParam(defaultValue = "English") String major`

It will take on the default value if the user hasn't specified an explicit value. Within our handler method here we can expect that a value for major is always present. It's either English if nothing has been specified, or whatever value has been explicitly specified.

Test and Run the App

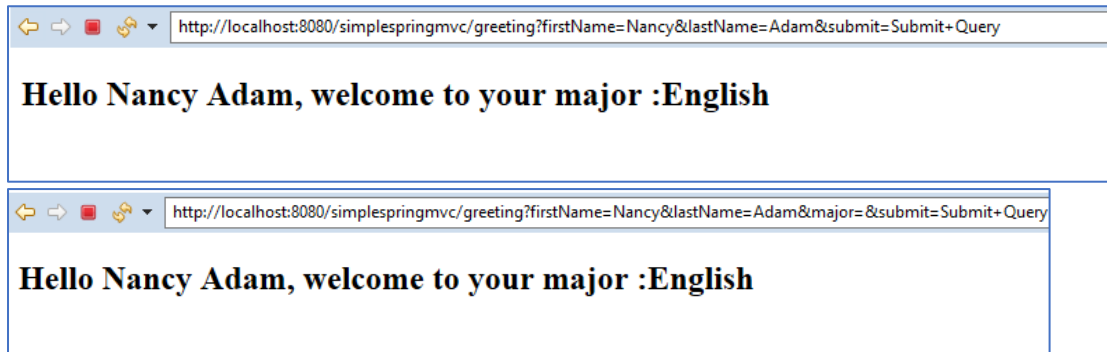
- Normal scenario , all input are not empty



- One of parameter is missing



- One of parameter is having default value when not declared



Another way to define Input Parameters in Controller Class, as follows

```
GreetingController.java
1 package com.mvc.controller;
2
3 import java.util.Map;
4
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.Model;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RequestParam;
9
10 @Controller
11 public class GreetingController {
12
13     @RequestMapping("/greeting")
14     public String greeting(
15         @RequestParam Map<String, String> allParametersMap,
16         Model model) {
17         String firstName = allParametersMap.get("firstName");
18         String lastName = allParametersMap.get("lastName");
19         String major = allParametersMap.get("major");
20         String msg = String.format("Hello %s %s, welcome to your major :%s ", firstName, lastName, major);
21         model.addAttribute("message", msg);
22         return "greeting_page";
23     }
24 }
25 }
```

Instead of accepting separate input arguments for our handler method corresponding to each of the request parameters, Spring can accept just a single parameter map. All of the request parameters that we pass to this handler will be available within this map.

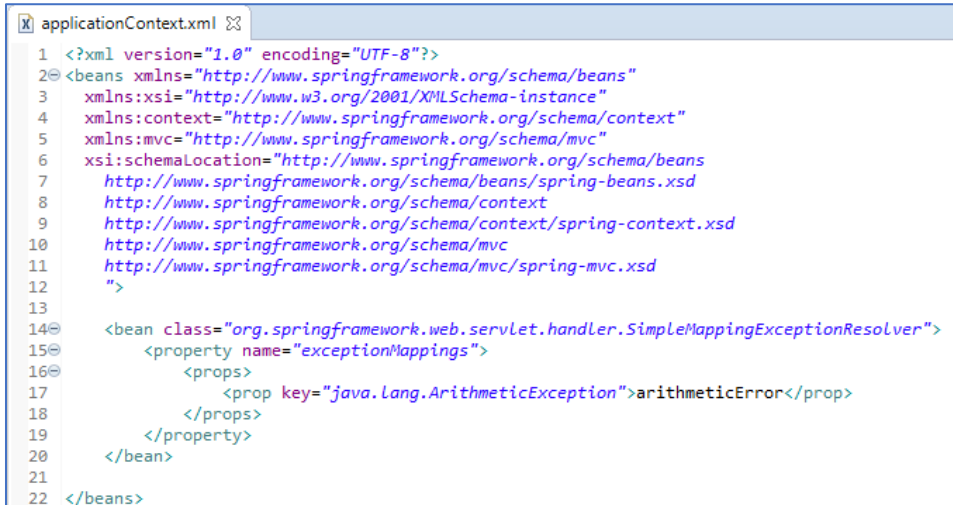
`@RequestParam Map<String, String> allParametersMap`

The advantages using this approach is that none of the request parameters are required.

Handling Exceptions using XML

When building Spring MVC application, it's quite possible that the code in one of controllers throws an exception. You want your user interface to handle this exception gracefully, maybe take you to a special page. And that's what we'll configure here in this demo. We'll see how we can use the XML configuration for our application to handle exceptions.

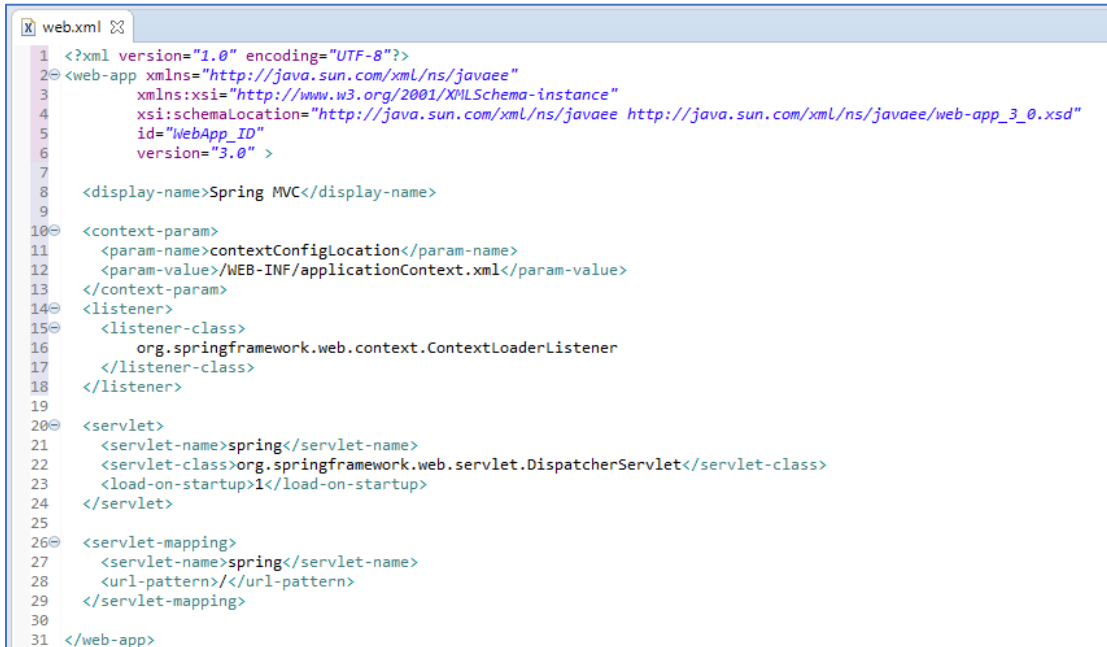
1. Specify the default exception configuration in applicationContext.xml under webapp/WEB-INF

A screenshot of an XML editor showing the configuration of applicationContext.xml. The file is titled 'applicationContext.xml' with a small icon. The XML content is as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7                           http://www.springframework.org/schema/beans/spring-beans.xsd
8                           http://www.springframework.org/schema/context
9                           http://www.springframework.org/schema/context/spring-context.xsd
10                          http://www.springframework.org/schema/mvc
11                          http://www.springframework.org/schema/mvc/spring-mvc.xsd"
12 >
13
14 <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
15   <property name="exceptionMappings">
16     <props>
17       <prop key="java.lang.ArithmeticException">arithmeticError</prop>
18     </props>
19   </property>
20 </bean>
21
22 </beans>
```

Whenever any controller within the spring application throws this particular exception that is the `ArithmeticException`, our app will take us to a specifically configured error page. This error page logical name is **arithmeticError**, it corresponds to a JSP file. The **SimpleMappingExceptionResolver** in Spring MVC is responsible for mapping all `ArithmeticExceptions`, across all of our controllers to this particular page.

2. Update Web.xml to accept applicationContext.xml file as configuration file for Spring Application

A screenshot of an XML editor showing the configuration of web.xml. The file is titled 'web.xml' with a small icon. The XML content is as follows:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://java.sun.com/xml/ns/javaee"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
5          id="WebApp_ID"
6          version="3.0" >
7
8   <display-name>Spring MVC</display-name>
9
10  <context-param>
11    <param-name>contextConfigLocation</param-name>
12    <param-value>/WEB-INF/applicationContext.xml</param-value>
13  </context-param>
14  <listener>
15    <listener-class>
16      org.springframework.web.context.ContextLoaderListener
17    </listener-class>
18  </listener>
19
20  <servlet>
21    <servlet-name>spring</servlet-name>
22    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
23    <load-on-startup>1</load-on-startup>
24  </servlet>
25
26  <servlet-mapping>
27    <servlet-name>spring</servlet-name>
28    <url-pattern>/</url-pattern>
29  </servlet-mapping>
30
31 </web-app>
```


Because It uses a separate XML file to specify the applicationContext of this SpringMVC application. Within web.xml file indicates the configLocation for the applicationContext.xml. The contextConfigLocation points to /WEB-INF/applicationContext.xml.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

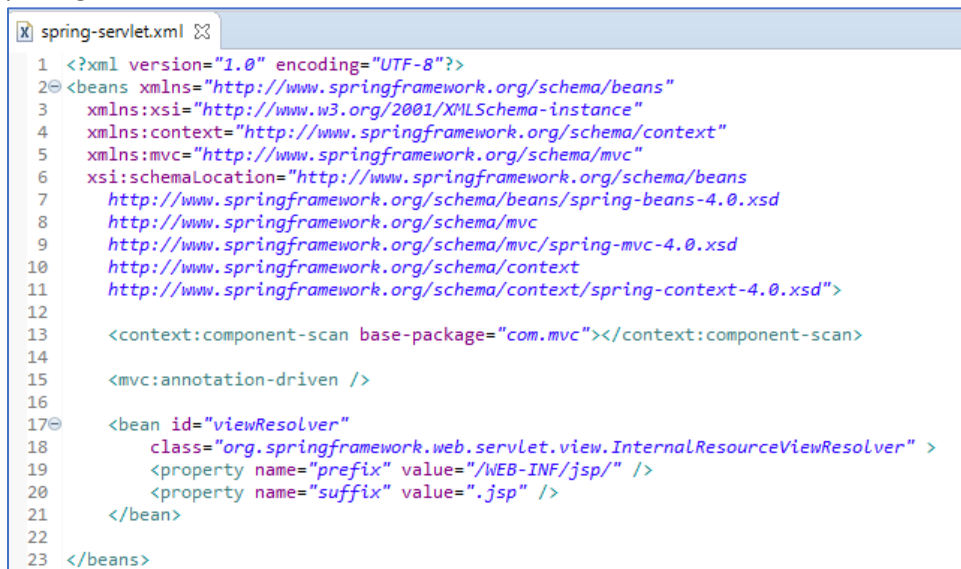
When you split up your configuration in this way, it's useful to use the ContextLoaderListener, which I've specified here as a listener-class.

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

This listener-class is a servlet-listener that loads all of the different configuration files across all layers, into a single spring applicationContext.

This is what helps us split our configuration across multiple files. The rest of the configuration here in web.xml is the same. We have the DispatcherServlet, and the url-pattern that it's mapped to.

3. There is no change in our spring-servlet.xml file, we've specified the component-scan, base package is com.mvc, we have a viewResolver.



```
spring-servlet.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xmlns:mvc="http://www.springframework.org/schema/mvc"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
8         http://www.springframework.org/schema/mvc
9         http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd
10        http://www.springframework.org/schema/context
11        http://www.springframework.org/schema/context/spring-context-4.0.xsd">
12
13     <context:component-scan base-package="com.mvc"></context:component-scan>
14
15     <mvc:annotation-driven />
16
17     <bean id="viewResolver"
18         class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
19         <property name="prefix" value="/WEB-INF/jsp/" />
20         <property name="suffix" value=".jsp" />
21     </bean>
22
23 </beans>
```

4. Prepare JSP Presentation Layer

```

index.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1" isELIgnored="false" %>
3 <html>
4 <head>
5   <meta charset="ISO-8859-1" content="text/html" http-equiv="Content-Type">
6 </head>
7 <title>Spring MVC Page</title>
8 </head>
9 <body>
10   <h1>Spring MVC!</h1>
11
12   <a href="hello">Hello Page</a><p></p>
13   <a href="start">Start Page</a>
14
15   <h1>Fantabulous University Database</h1>
16   <a href="university/students">View Students</a>
17
18   <h1>Greeting</h1>
19   <a href="registration.jsp">Greeting</a>
20
21   <h1>Calculate</h1>
22   <a href="calculate.jsp">calculation</a>
23
24 </body>
25 </html>

```

```

calculate.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Calculate</title>
8 </head>
9 <body>
10   <form action="calculate">
11     Number 1 : <input type="text" name="number1"><br/><br/>
12     Number 2 : <input type="text" name="number2"><br/><br/>
13     <input type="submit" name="submit">
14   </form>
15
16 </body>
17 </html>

```

The /calculate path is where we'll submit our form data. Now, there are two input text boxes corresponding to Number 1 and Number 2, where we'll input two numbers. And when we hit Submit, we'll get the result of arithmetic operations performed on these numbers. The name of the first input text box is number1, the second one is number2, these will correspond to the request parameter names.

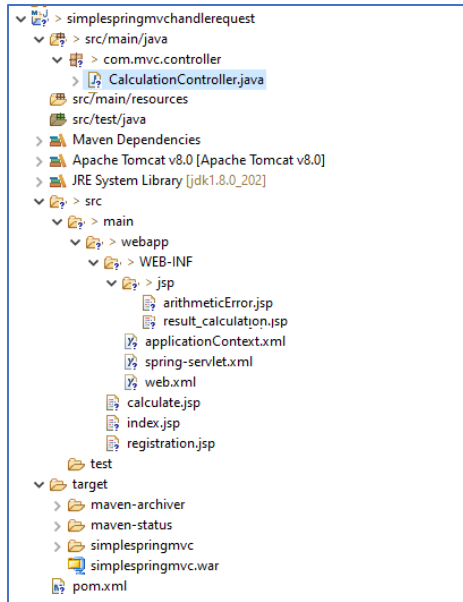
```
result_calculation.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Result Calculation</title>
8   <style type="text/css">
9     div {
10       color: blue;
11       font-size: 64px;
12       padding-left: 24px;
13       padding-left: 48px;
14     }
15   </style>
16 </head>
17 <body>
18   <div>Sum : ${sum}</div>
19   <div>Subtract : ${subtract}</div>
20   <div>Multiply : ${multiply}</div>
21   <div>Divide : ${divide}</div>
22 </body>
23 </html>
```

At the resultsPage, which will display the results of the arithmetic operations that we perform on these numbers. We have the title Calculation Results. This page displays the sum, difference, multiplication, that is the product and the division result on the two numbers that we had input using our form.

```
arithmeticError.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Arithmetic Error</title>
8 </head>
9 <body>
10   <h2>Mathematical error when performing calculation : ${exception}</h2>
11 </body>
12 </html>
```

In the **arithmeticError.jsp** file that we've specified within the WEB-INF/jsp folder. This is the page that will be displayed whenever an **arithmeticException** is encountered in our code. In our demo here, we'll print out a simple message, Mathematical error while performing calculation:. And we'll print out the message within the exception.

Spring MVC Framework : Handling Request & Errors



5. Prepare the calculation controller class as follows

```
CalculationController.java
1 package com.mvc.controller;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class CalculationController {
7
8     @RequestMapping("/calculate")
9     public String calculate(
10         @RequestParam("number1") int number1,
11         @RequestParam("number2") int number2,
12         Model model
13     ) {
14
15         String message = String.format("Here are your calculation result for : %d and %d ", number1, number2);
16
17         model.addAttribute("message", message);
18         model.addAttribute("sum", number1 + number2);
19         model.addAttribute("subtract", number1 - number2);
20         model.addAttribute("multiply", number1 * number2);
21         model.addAttribute("divide", number1 / number2);
22
23         return "result_calculation";
24     }
25 }
```

In the CalculationController, tagged using the @Controller annotation. There's exactly one request that is mapped to a handler method here, to the /calculate path. This is the handler mapping for our form data submission. Now this calculate method takes in three input arguments, two of which are request parameters that are injected in. The first request parameter is an integer number1, the name of the request parameter is number1, and it comes from our first input text box. The second request parameter is number2, which comes from our second input text box.

When you look at these input arguments, you'll see an implicit assumption here, we assume that number1 and number2 will both be integers. Even though the input text box can accept text values, we inject the model as well as the third input argument.

Within the handler method, we'll construct a message which says Here are your calculation results for number1 and number2. We'll then calculate the sum, difference, product, and quotient for number1 and number2. And we'll add each of these calculation results to our model, so that it's accessible using the resultsPage. The model now contains values for the keys message, sum, subtract, multiply, divide.

The resultsPage will use all of these to print out to screen.

6. Build and Test run the app

- Normal Expected Scenario

http://localhost:8080/simplespringmvc/calculate.jsp

Number 1 :

Number 2 :

Sum : 8
Subtract : 4
Multiply : 8
Divide : 3

- Failed Scenario

http://localhost:8080/simplespringmvc/calculate.jsp

Number 1 :

Number 2 :

Mathematical error when performing calculation : java.lang.ArithmeticException: / by zero

Mapping Multiple and Default Exception

It's possible in Spring MVC to set up multiple mappings from exceptions to error pages. It's also possible to configure a default error page for those exceptions that haven't been explicitly mapped. This is what we'll see here in this demo.

Notice within my applicationContext.xml file in my WEB-INF folder, I have a bean specification for the SimpleMappingExceptionHandler. We'll continue to use this.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xmlns:mvc="http://www.springframework.org/schema/mvc"
6     xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-context.xsd
10        http://www.springframework.org/schema/mvc
11        http://www.springframework.org/schema/mvc/spring-mvc.xsd
12    ">
13
14    <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">
15        <property name="exceptionMappings">
16            <props>
17                <prop key="java.lang.ArithmeticException">arithmeticError</prop>
18                <prop key="java.sql.SQLException">sqlError</prop>
19            </props>
20        </property>
21
22        <property name="defaultErrorView" value="error" />
23    </bean>
24
25 </beans>
```

Within the exceptionMappings property, I now have two entries. The key for the first entry is a java.lang.ArithmeticException that maps to the arithmeticError logical view. So if any of our controllers throw this ArithmeticException, that is the view that will be displayed. I have another key here for the java.sql.SQLException. If this is the exception that is encountered in our controllers, the page that will be displayed is the view corresponding to the sqlError logical view.

And notice at the bottom, I have a defaultErrorView property. The value is set to error. error.jsp is the page that will be displayed for all other exceptions thrown by our controllers. So any exception other than the ArithmeticException and the SQLException, both of which have been explicitly mapped will be handled by this error view. Our application will perform the same arithmetic calculation as before on two numbers and display the sum, difference, product, and quotient for these two numbers.

7. Prepare Presentation Layer JSP file

```
sqlError.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>SQL Database Error</title>
8 </head>
9 <body>
10 <h2>There was an error accessing the database : ${exception}</h2>
11 </body>
12 </html>
```

In the case of a SQLException, the sqlError.jsp file will be rendered to screen. That will say there was an error accessing the database and exception details will be printed out.

```
error.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>General Error</title>
8 </head>
9 <body>
10 <h2>General error while performing calculation : ${exception}</h2>
11 </body>
12 </html>
```

And finally, error.jsp will be the view rendered for all other exceptions.

That will say General error while performing calculation and we'll print out exception details.

This is the mapping specified in our default error view property. This is the error page rendered for all errors that have not been explicitly mapped.

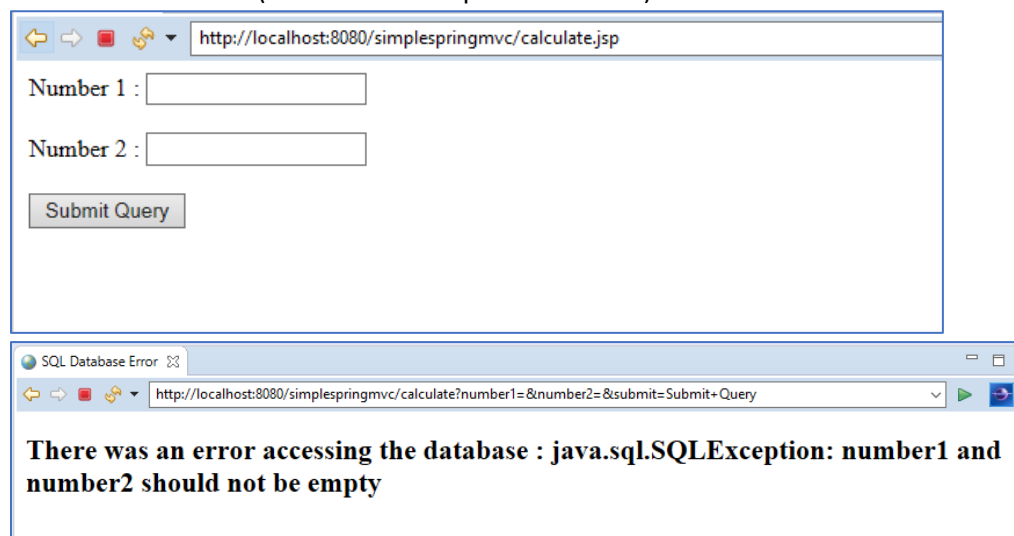
8. Update Controller Class as follows

```
CalculationController.java
1 package com.mvc.controller;
2
3 import java.sql.SQLException;
4
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.Model;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RequestParam;
9
10 @Controller
11 public class CalculationController {
12
13     @RequestMapping("/calculate")
14     public String calculate (
15         @RequestParam(name = "number1", required = false) String strNumber1,
16         @RequestParam(name = "number2", required = false) String strNumber2,
17         Model model
18     ) throws Exception {
19
20         int number1 = 0;
21         int number2 = 0;
22
23         if (strNumber1 == null || strNumber2 == null || strNumber1.trim().length() == 0 || strNumber2.trim().length() == 0)
24             throw new SQLException("number1 and number2 should not be empty");
25
26         number1 = (int) new Integer(strNumber1.trim());
27         number2 = (int) new Integer(strNumber2.trim());
28
29
30         String message = String.format("Here are your calculation result for : %d and %d ", number1, number2);
31
32         model.addAttribute("message", message);
33         model.addAttribute("sum", number1 + number2);
34         model.addAttribute("subtract", number1 - number2);
35         model.addAttribute("multiply", number1 * number2);
36         model.addAttribute("divide", number1 / number2);
37
38         return "result_calculation";
39     }
40 }
```

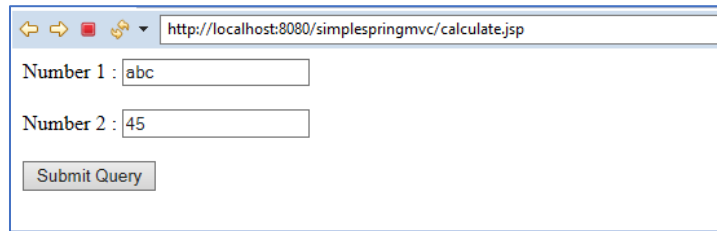
adding an extra code to handle some exception

9. Build and Run the code

- Failed Scenario No 1 (No Parameter input are filled in)

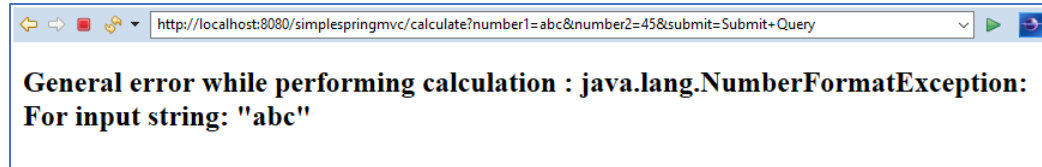


- Failed Scenario No 2 (invalid numeric input to one or all inputs parameter)



Number 1 :

Number 2 :



**General error while performing calculation : java.lang.NumberFormatException:
For input string: "abc"**

Handling Exceptions Using Annotation

In this demo, we'll see how we can use a combination of XML based configuration as well as code base configuration for exception handling within your Spring MVC application. Here is our applicationContext.xml file. There is nothing much here that you don't already know. I've specified the bean for the SimpleMappingExceptionHandler.

10. Modify ApplicationContext xml file by removing "default Error View" handler

```
applicationContext.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xmlns:context="http://www.springframework.org/schema/context"
5       xmlns:mvc="http://www.springframework.org/schema/mvc"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7         http://www.springframework.org/schema/beans/spring-beans.xsd
8         http://www.springframework.org/schema/context
9         http://www.springframework.org/schema/context/spring-context.xsd
10        http://www.springframework.org/schema/mvc
11        http://www.springframework.org/schema/mvc/spring-mvc.xsd
12    ">
13
14    <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
15        <property name="exceptionMappings">
16            <props>
17                <prop key="java.lang.ArithmeticException">arithmeticError</prop>
18                <!-- <prop key="java.sql.SQLException">sqlError</prop> -->
19            </props>
20        </property>
21
22        <!-- <property name="defaultErrorView" value="error" /> -->
23    </bean>
24
25 </beans>
```

I've gotten rid of the other mappings as well as the default error page specification in this applicationContext.xml. Our other XML-based configuration files remain the same.

11. Update error.jsp

```
error.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6   <meta charset="ISO-8859-1">
7   <title>Unknown Error</title>
8 </head>
9 <body>
10   <h2>Unknown error while performing calculation : ${exceptionMessage}</h2>
11 </body>
12 </html>
```

This page will simply say Unknown error occurred while performing calculation:. And it'll print out the message present in the **exceptionMessage** variable.

We'll explicitly set a value for exceptionMessage from our controller code.

12. Add new method to handle error with annotation in the Controller as follows

```

1 package com.mvc.controller;
2
3 import java.sql.SQLException;
4
5 import org.springframework.stereotype.Controller;
6 import org.springframework.ui.Model;
7 import org.springframework.web.bind.annotation.ExceptionHandler;
8 import org.springframework.web.bind.annotation.RequestMapping;
9 import org.springframework.web.bind.annotation.RequestParam;
10 import org.springframework.web.servlet.ModelAndView;
11
12 @Controller
13 public class CalculationController {
14
15     @RequestMapping("/calculate")
16     public String calculate (
17         @RequestParam(name = "number1", required = false) String strNumber1,
18         @RequestParam(name = "number2", required = false) String strNumber2,
19         Model model
20     ) throws Exception {
21
22         int number1 = 0;
23         int number2 = 0;
24
25         if (strNumber1 == null || strNumber2 == null || strNumber1.trim().length() == 0 || strNumber2.trim().length() == 0)
26             throw new SQLException("number1 and number2 should not be empty");
27
28         number1 = (int) new Integer(strNumber1.trim());
29         number2 = (int) new Integer(strNumber2.trim());
30
31         String message = String.format("Here are your calculation result for : %d and %d ", number1, number2);
32
33         model.addAttribute("message", message);
34         model.addAttribute("sum", number1 + number2);
35         model.addAttribute("subtract", number1 - number2);
36         model.addAttribute("multiply", number1 * number2);
37         model.addAttribute("divide", number1 / number2);
38
39         return "result_calculation";
40     }
41
42     @ExceptionHandler({Exception.class, SQLException.class})
43     public ModelAndView handleException(Exception ex) {
44         ModelAndView model = new ModelAndView("error");
45         model.addObject("exceptionMessage", ex.getMessage());
46         return model;
47     }
48 }

```

We've defined a second method here in our controller class, and we have annotated it using the `@ExceptionHandler` annotation.

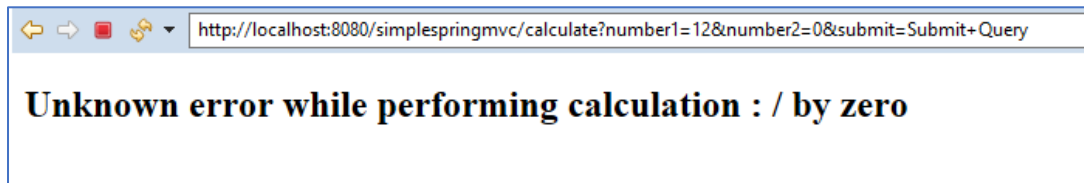
This is a way to programmatically configure the view that you want to render when exceptions are encountered in your code. I've explicitly specified that if we encounter the `Exception` or the `SQLException`, then we should render the error view. The `ExceptionHandler` annotation allows you to specify one or more exceptions that you want mapped. Within the method I instantiate a `ModelAndView` for the error view.

Add an `exceptionMessage` to be printed out to screen and return.

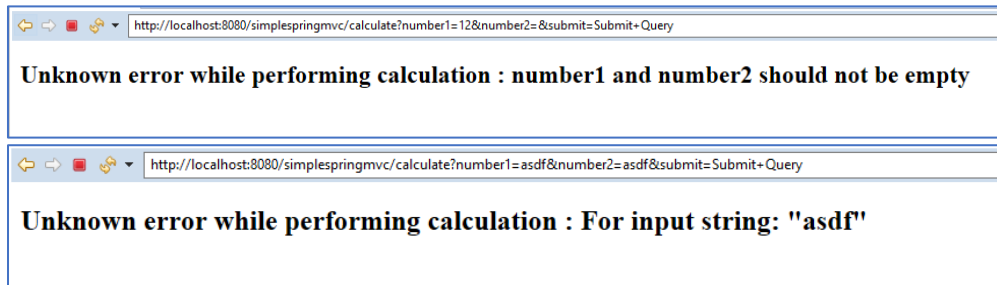
We have access to the exception that triggered this `ExceptionHandler`. It will be injected as an input argument to this method invocation. Time for us to take a look at how this programmatic mapping of exceptions work.

13. Build and Run the app

Failed scenario due to arithmetic exception will returning as follows



Other Failed scenario is always returning as follows:



Course Summary

In this course, we saw how we can deal with multiple controllers and views, request parameters, dynamic URL paths, and exceptions in Spring MVC. We started off by building a WAR file using Maven, and deploying this WAR file directly to a Tomcat server. We created applications that used multiple controller specifications, and rendered different views for different handler mappings.

We then set up our application using the classic 3-tier architecture, the data access layer, the business logic layer, and the presentation layer. We then explored how we could access request parameters sent by the user using the `@RequestParam` annotation. And how dynamic URL path elements could be extracted using the `@PathVariable` annotation. Finally, we saw how we could customize the error pages to handle exceptions in our code using the built-in Spring MVC exception resolver.

We configured exception handling using XML file specifications as well as programmatically.

Quiz

1. How should you name the XML file with the servlet specification for your Spring MVC application?

XML file with the "spring-" prefix

✓ XML file with the "-servlet" suffix

XML file with the ".mvc" extension

There are no naming conventions for this XML file

2. Which folder under the Tomcat server installation should you deploy your .war file to?

src/

tomcat/

✓ webapps/

META-INF

3. What is the advantage of deploying your .war file to the Tomcat webapps/ folder directly, rather than using Eclipse to run on Tomcat?

Eclipse does not allow you to run the Tomcat server

Eclipse runs the server but does not deploy the .war files

✓ Eclipse adds in a layer of caching for builds that could make it hard to work with

Eclipse adds an additional interceptor to your requests

4. Given the controller below, what is the href link to request a page from the index.jsp entry point in your application?

```
@RequestMapping("/welcome")

public String welcome_page() {

    return "welcome_page";

}
```

/welcome

✓ welcome

app_root/welcome

some_page/welcome

5. What XML tags in the server configuration allows Spring MVC to detect multiple controller objects that might be present in your app?

`<mvc:controller/>`

`<mvc:resources/>`

✓ `<context:component-scan base-package="com.skillsoft" />`

✓ `<mvc:annotation-driven/>`

6. What tiers make up the three tiers of a Spring MVC application?

✓ Presentation tier

Configuration tier

✓ Data access tier

✓ Business logic tier

Indexing tier

7. Which statements are true about the `@RequestMapping` annotation?

Can be applied to input arguments of a method

✓ Can be applied to methods of a class

✓ Can be applied to the class itself

Can be applied to member variables of a class

8. What is the annotation used to access the dynamic elements of a URL mapping?

✓ `@PathVariable`

@RequestMapping

@Url

@RequestParam

9. What is the class of the request object that can be an input to our handler method?

✓ `HttpServletRequest`

Request

WebRequest

RequestParam

10. Which statements about accessing request parameters is true?

✓ Request parameters are required unless otherwise specified

✓ Can be injected into a method using `@RequestParam` annotation

We can have only a single request parameter input to a method

Cannot be injected, have to be explicitly accessed using the Request object

11. In which cases do you NOT need to specify a value for the "greeting" request parameter?

✓ `printGreeting(@RequestParam(required=false) String greeting)`

`printGreeting(@RequestParam(required=true) String greeting)`

`printGreeting(@RequestParam String greeting)`

✓ `printGreeting(@RequestParam(defaultValue = "Hello") String greeting)`

12. What is the role of the ContextConfigListener?

To specify that the application context needs to be injected using Java annotations

To specify that the application context may change at runtime and will need to be updated

✓ To load the configuration properties from multiple XML files to a single application context configuration

To configure the application context programmatically

13. What is the Spring MVC class that allows you to map different exceptions to different views?

ExceptionHandler

ExceptionHandler

✓ `SimpleMappingExceptionHandler`

ExceptionHandler

14. What annotation can we use to map exceptions to specific views?

@ExceptionHandler

@ExceptionHandler

@ExceptionHandler

✓ `@ExceptionHandler`