**Spring MVC Example for User Registration and Login**

This article is a step by step guide for User Registration and Login using the below tools and technologies.

1. Spring Framework (Core, MVC & JBDC)

2. Java 1.8

3. Maven 3.3.9

4. Eclipse IDE (Mars2)

5. MySQL 5.1

## Step 1: Create a Maven Project

Using Eclipse IDE, create a Maven Project by selecting Web Archetype.

## Step 2: Update Pom.xml

Update your maven dependencies.

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>jbr</groupId>

<artifactId>SpringMvcUser</artifactId>

<packaging>war</packaging>

<version>0.1</version>

<name>SpringMvcUser Maven Webapp</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>4.3.2.RELEASE</spring.version>

<junit.version>4.12</junit.version>

<servlet.version>3.1.0</servlet.version>

<servlet.version>3.0.1</servlet.version>

<mysql.connector.version>5.1.9</mysql.connector.version>

</properties>

<dependencies>

<!-- Junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>${junit.version}</version>

<scope>test</scope>

</dependency>

<!--Servlet-Api -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>${servlet.version}</version>

</dependency>

<!-- Spring Framework -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${spring.version}</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<!-- Servlet API -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>${servlet.version}</version>

</dependency>

<!-- MySQL database driver -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>${mysql.connector.version}</version>

</dependency>

</dependencies>

<build>

<finalName>SpringMvcUser</finalName>

</build>

</project>

## Step 3: Update web.xml

Update the web.xml under src/main/webapp/WEB-INF as shown below:

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" version="3.0">

<display-name>Archetype Created Web Application</display-name>

<welcome-file-list>

<welcome-file>home.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>spring-mvc</servlet-name>

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

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

</servlet>

<servlet-mapping>

<servlet-name>spring-mvc</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

1. DispatcherServlet  is the root Servlet for any Servlet defined in your application. It will redirect the request made to the application to the appropriate controller based on the URL.
2. spring-mvc  is the name of the root servlet and Spring container will look for the configuration with this name. So next step is to create a spring configuration with spring-mvc-servlet.xml.

## Step 4: Create spring-mvc-servlet.xml

Create  spring-mvc-servlet.xml  under the src/main/webapp/WEB-INF folder and define the beans.

<?xml version="1.0" encoding="UTF-8"?>

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

<import resource="classpath:jbr/config/user-beans.xml" />

<context:component-scan base-package="jbr.springmvc" />

<context:annotation-config />

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/jsp/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

## Step 5: Create Controllers

Create Separate Controllers for User Registration and Login as below.

Create a package  jbr.springmvc.controller

#### RegistrationController.java

package jbr.springmvc.controller;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import jbr.springmvc.model.User;

import jbr.springmvc.service.UserService;

@Controller

public class RegistrationController {

@Autowired

public UserService userService;

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

public ModelAndView showRegister(HttpServletRequest request, HttpServletResponse response) {

ModelAndView mav = new ModelAndView("register");

mav.addObject("user", new User());

return mav;

}

@RequestMapping(value = "/registerProcess", method = RequestMethod.POST)

public ModelAndView addUser(HttpServletRequest request, HttpServletResponse response,

@ModelAttribute("user") User user) {

userService.register(user);

return new ModelAndView("welcome", "firstname", user.getFirstname());

}

}

#### LoginController.java

package jbr.springmvc.controller;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

import jbr.springmvc.model.Login;

import jbr.springmvc.model.User;

import jbr.springmvc.service.UserService;

@Controller

public class LoginController {

@Autowired

UserService userService;

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

public ModelAndView showLogin(HttpServletRequest request, HttpServletResponse response) {

ModelAndView mav = new ModelAndView("login");

mav.addObject("login", new Login());

return mav;

}

@RequestMapping(value = "/loginProcess", method = RequestMethod.POST)

public ModelAndView loginProcess(HttpServletRequest request, HttpServletResponse response,

@ModelAttribute("login") Login login) {

ModelAndView mav = null;

User user = userService.validateUser(login);

if (null != user) {

mav = new ModelAndView("welcome");

mav.addObject("firstname", user.getFirstname());

} else {

mav = new ModelAndView("login");

mav.addObject("message", "Username or Password is wrong!!");

}

return mav;

}

}

## Step 5: Create POJO Classes

Create a package:  jbr.springmvc.model

#### Login.java

package jbr.springmvc.model;

public class Login {

private String username;

private String password;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

#### User.java

package jbr.springmvc.model;

public class User {

private String username;

private String password;

private String firstname;

private String lastname;

private String email;

private String address;

private int phone;

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

public String getFirstname() {

return firstname;

}

public void setFirstname(String firstname) {

this.firstname = firstname;

}

public String getLastname() {

return lastname;

}

public void setLastname(String lastname) {

this.lastname = lastname;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

public int getPhone() {

return phone;

}

public void setPhone(int phone) {

this.phone = phone;

}

}

Note: If you wanted to add any new logic, you can add another layer called service and access the DAO layer.

## Step 6: Create DAO Classes

Create an interface/classes for accessing the MySQL Database.

Create a package:  jbr.springmvc.dao

#### UserDao.java

package jbr.springmvc.dao;

import jbr.springmvc.model.Login;

import jbr.springmvc.model.User;

public interface UserDao {

void register(User user);

User validateUser(Login login);

}

#### UserDaoImpl.java

package jbr.springmvc.dao;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.RowMapper;

import jbr.springmvc.model.Login;

import jbr.springmvc.model.User;

public class UserDaoImpl implements UserDao {

@Autowired

DataSource datasource;

@Autowired

JdbcTemplate jdbcTemplate;

public void register(User user) {

String sql = "insert into users values(?,?,?,?,?,?,?)";

jdbcTemplate.update(sql, new Object[] { user.getUsername(), user.getPassword(), user.getFirstname(),

user.getLastname(), user.getEmail(), user.getAddress(), user.getPhone() });

}

public User validateUser(Login login) {

String sql = "select \* from users where username='" + login.getUsername() + "' and password='" + login.getPassword()

+ "'";

List<User> users = jdbcTemplate.query(sql, new UserMapper());

return users.size() > 0 ? users.get(0) : null;

}

}

class UserMapper implements RowMapper<User> {

public User mapRow(ResultSet rs, int arg1) throws SQLException {

User user = new User();

user.setUsername(rs.getString("username"));

user.setPassword(rs.getString("password"));

user.setFirstname(rs.getString("firstname"));

user.setLastname(rs.getString("lastname"));

user.setEmail(rs.getString("email"));

user.setAddress(rs.getString("address"));

user.setPhone(rs.getInt("phone"));

return user;

}

}

## Step 7: Create Spring Beans Configuration

Under  resources , create a new xml bean configuration  user-beans.xml to define the beans we created.

<?xml version="1.0" encoding="UTF-8"?>

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

<context:component-scan base-package="jbr.springmvc" />

<context:annotation-config />

<bean id="userService" class="jbr.springmvc.service.UserServiceImpl" />

<bean id="userDao" class="jbr.springmvc.dao.UserDaoImpl" />

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="datasource" />

</bean>

<bean id="datasource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/myusers" />

<property name="username" value="root" />

<property name="password" value="root" />

</bean>

</beans>

## Step 8: Create Tables in MySQL

CREATE DATABASE

IF NOT EXISTS myusers;USE

DROP TABLE

DROP TABLE IF EXISTS `myusers`.`users`;CREATE TABLE `myusers`.`users`

(

`username` VARCHAR(45) NOT NULL,

`password` VARCHAR(45) NULL,

`firstname` VARCHAR(45) NOT NULL,

`lastname` VARCHAR(45) NULL,

`email` VARCHAR(45) NULL,

`address` VARCHAR(45) NULL,

`phone` INT NULL,

PRIMARY KEY (`username`)

)

## Step 9: Create Views

Under the /src/main/webapp folder, create a home.jsp which we defined as 'welcome-file' in web.xml.

#### home.jsp

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Welcome</title>

</head>

<body>

<table align="center">

<tr>

<td><a href="login">Login</a>

</td>

<td><a href="register">Register</a>

</td>

</tr>

</table>

</body>

</html>

Under /src/main/webapp/jsp create:

#### register.jsp

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

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Registration</title>

</head>

<body>

<form:form id="regForm" modelAttribute="user" action="registerProcess" method="post">

<table align="center">

<tr>

<td>

<form:label path="username">Username</form:label>

</td>

<td>

<form:input path="username" name="username" id="username" />

</td>

</tr>

<tr>

<td>

<form:label path="password">Password</form:label>

</td>

<td>

<form:password path="password" name="password" id="password" />

</td>

</tr>

<tr>

<td>

<form:label path="firstname">FirstName</form:label>

</td>

<td>

<form:input path="firstname" name="firstname" id="firstname" />

</td>

</tr>

<tr>

<td>

<form:label path="lastname">LastName</form:label>

</td>

<td>

<form:input path="lastname" name="lastname" id="lastname" />

</td>

</tr>

<tr>

<td>

<form:label path="email">Email</form:label>

</td>

<td>

<form:input path="email" name="email" id="email" />

</td>

</tr>

<tr>

<td>

<form:label path="address">Address</form:label>

</td>

<td>

<form:input path="address" name="address" id="address" />

</td>

</tr>

<tr>

<td>

<form:label path="phone">Phone</form:label>

</td>

<td>

<form:input path="phone" name="phone" id="phone" />

</td>

</tr>

<tr>

<td></td>

<td>

<form:button id="register" name="register">Register</form:button>

</td>

</tr>

<tr></tr>

<tr>

<td></td>

<td><a href="home.jsp">Home</a>

</td>

</tr>

</table>

</form:form>

</body>

</html>

#### ****login.jsp****

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

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Login</title>

</head>

<body>

<form:form id="loginForm" modelAttribute="login" action="loginProcess" method="post">

<table align="center">

<tr>

<td>

<form:label path="username">Username: </form:label>

</td>

<td>

<form:input path="username" name="username" id="username" />

</td>

</tr>

<tr>

<td>

<form:label path="password">Password:</form:label>

</td>

<td>

<form:password path="password" name="password" id="password" />

</td>

</tr>

<tr>

<td></td>

<td align="left">

<form:button id="login" name="login">Login</form:button>

</td>

</tr>

<tr></tr>

<tr>

<td></td>

<td><a href="home.jsp">Home</a>

</td>

</tr>

</table>

</form:form>

<table align="center">

<tr>

<td style="font-style: italic; color: red;">${message}</td>

</tr>

</table>

</body>

</html>

#### welcome.jsp

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Welcome</title>

</head>

<body>

<table>

<tr>

<td>Welcome ${firstname}</td>

</tr>

<tr>

</tr>

<tr>

</tr>

<tr>

<td><a href="home.jsp">Home</a>

</td>

</tr>

</table>

</body>

</html>

## Step 10: Run the Application

1. Right-click the application and Maven->clean.
2. Right-click the application and Maven->install.
3. Right-click the application and Run As->Run on Server->Run using Tomcat.

You will see the below outputs.

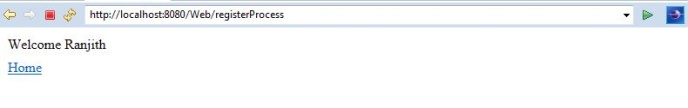
Home Page:



User Registration:



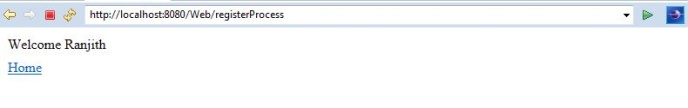
User Registration Success:



Login:



Login Success:



Spring 4 MVC HelloWorld Tutorial – Annotation/JavaConfig Example

VISIT

http://websystique.com/springmvc/spring-4-mvc-helloworld-tutorial-annotation-javaconfig-full-example/

# Spring MVC 4 HelloWorld Annotation/JavaConfig Example, step-by-step learning Spring MVC 4 annotations, project setup, code, deploy & Run, in simple way. Let’s get going.

In the previous [Spring MVC 4 Hello World tutorial-XML example](http://websystique.com/springmvc/spring-4-mvc-helloworld-tutorial-full-example/), we have developed a Hello world web application using XML configuration. However, XML is not the only way to configure spring application. Alternatively, we can configure the application using Java configuration.

# If you look back on previous post, you will find that there are mainly two places where we have used XML configuration. First, in **spring-servlet.xml** where we have defined a view-resolver for identifying the real view , and location to search for beans via component-scanning. Second, in **web.xml**, we have defined the front-controller configuration and the url pattern it will be looking on.

# In this tutorial, we will again create a Hello world example but using Java configuration this time. We will **REMOVE** both of above mentioned xml files and replace these xml configurations via their java counterparts.

**Following technologies stack being used:**

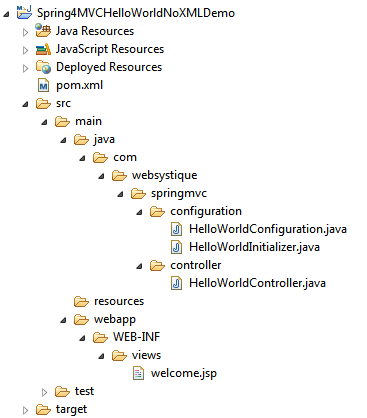
* Spring 4.0.6.RELEASE
* Maven 3
* JDK 1.6
* Tomcat 8.0.21
* Eclipse JUNO Service Release 2

Let’s begin.

#### Step 1: Create a project with required directory structure

Post [Creating a maven web project with eclipse](http://websystique.com/maven/create-a-maven-web-project-with-eclipse/) contains step-by-step instruction to create a maven project with eclipse.

Following will be the final Project structure.



Now let’s add/update the content mentioned in above project structure discussing each in detail.

#### Step 2: Update pom.xml with Spring and Servlet dependency

The Spring java-based configuration we are going to discuss depends on Servlet 3.0 api, so we need to include that as a dependency in pom.xml

|  |
| --- |
| <?xml version="1.0"?>  <project xsi:schemaLocation="<a class="vglnk" href="<http://maven.apache.org/POM/4.0.0>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>POM</span><span>/</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span></a> <a class="vglnk" href="<http://maven.apache.org/xsd/maven-4.0.0.xsd>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>xsd</span><span>/</span><span>maven</span><span>-</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span><span>.</span><span>xsd</span></a>"      xmlns="<a class="vglnk" href="<http://maven.apache.org/POM/4.0.0>" rel="nofollow"><span>http</span><span>://</span><span>maven</span><span>.</span><span>apache</span><span>.</span><span>org</span><span>/</span><span>POM</span><span>/</span><span>4</span><span>.</span><span>0</span><span>.</span><span>0</span></a>" xmlns:xsi="<a class="vglnk" href="<http://www.w3.org/2001/XMLSchema-instance>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>w3</span><span>.</span><span>org</span><span>/</span><span>2001</span><span>/</span><span>XMLSchema</span><span>-</span><span>instance</span></a>">        <modelVersion>4.0.0</modelVersion>      <groupId>com.websystique.springmvc</groupId>      <artifactId>Spring4MVCHelloWorldNoXMLDemo</artifactId>      <packaging>war</packaging>      <version>1.0.0</version>      <name>Spring4MVCHelloWorldNoXMLDemo</name>        <properties>          <springframework.version>4.0.6.RELEASE</springframework.version>      </properties>        <dependencies>          <dependency>              <groupId>org.springframework</groupId>              <artifactId>spring-webmvc</artifactId>              <version>${springframework.version}</version>          </dependency>            <dependency>              <groupId>javax.servlet</groupId>              <artifactId>javax.servlet-api</artifactId>              <version>3.1.0</version>          </dependency>          <dependency>              <groupId>javax.servlet.jsp</groupId>              <artifactId>javax.servlet.jsp-api</artifactId>              <version>2.3.1</version>          </dependency>          <dependency>              <groupId>javax.servlet</groupId>              <artifactId>jstl</artifactId>              <version>1.2</version>          </dependency>      </dependencies>        <build>          <pluginManagement>              <plugins>                  <plugin>                      <groupId>org.apache.maven.plugins</groupId>                      <artifactId>maven-compiler-plugin</artifactId>                      <version>3.2</version>                      <configuration>                          <source>1.6</source>                          <target>1.6</target>                      </configuration>                  </plugin>                  <plugin>                      <groupId>org.apache.maven.plugins</groupId>                      <artifactId>maven-war-plugin</artifactId>                      <version>2.4</version>                      <configuration>                          <warSourceDirectory>src/main/webapp</warSourceDirectory>                          <warName>Spring4MVCHelloWorldNoXMLDemo</warName>                          <failOnMissingWebXml>false</failOnMissingWebXml>                      </configuration>                  </plugin>              </plugins>          </pluginManagement>          <finalName>Spring4MVCHelloWorldNoXMLDemo</finalName>      </build>  </project> |

First thing to notice here is the maven-war-plugin declaration. As we will be completely removing web.xml, we will need to configure this plugin in order to avoid maven failure to build war package. Second change is the inclusion of JSP/Servlet/Jstl dependencies which we might be needing as we are going to use servlet api’s and jstl view in our code.In general, containers already contains these libraries, so we can set the scope as provided for them in pom.xml.

Additionally, maven-compiler-plugin has been added here to explicitly specify the jdk-version we are going to use. Do note that it also forces eclipse to respect the jdk-version being used for the project. if it is not present, and you perform mvn-update from within your eclipse, eclipse switches jdk-version back to default jdk-version [1.5] which is annoying. So do add it in your project pom as well.

#### Step 3: Add Controller

Add a controller class under src/main/java as shown below:

com.websystique.springmvc.controller.HelloWorldController

|  |
| --- |
| package com.websystique.springmvc.controller;    import org.springframework.stereotype.Controller;  import org.springframework.ui.ModelMap;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;    @Controller  @RequestMapping("/")  public class HelloWorldController {        @RequestMapping(method = RequestMethod.GET)      public String sayHello(ModelMap model) {          model.addAttribute("greeting", "Hello World from Spring 4 MVC");          return "welcome";      }        @RequestMapping(value = "/helloagain", method = RequestMethod.GET)      public String sayHelloAgain(ModelMap model) {          model.addAttribute("greeting", "Hello World Again, from Spring 4 MVC");          return "welcome";      }    } |

@Controller annotation on class name declares this class as spring bean and @RequestMapping annotation declares that this class is default handler for all requests of type ‘/’. First method does not have any mapping declared so, it will inherit the mapping from mapping declared on class level, acting as default handler for GET requests. Second method (due to additional mapping declaration with value attribute) will serve the request of form /helloagain. Attribute method says which type of HTTP request this method can serve. ModelMap is a Map implementation, which here acting as replacement of [request.getAttribute()/request.setAttribute()]setting values as request attribute. Note that we are returning “welcome” string form this method. This string will be suffixed and prefixed with suffix and prefix defined in view resolver(see spring-servlet.xml above) to form the real view file name.

#### Step 4: Add View

Create a new folder named views under WEB-INF and add in a Simple JSP page welcome.jsp (WEB-INF/views/welcome.jsp) to simply access the model value sent from controller.

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=ISO-8859-1"      pageEncoding="ISO-8859-1"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "<a class="vglnk" href="<http://www.w3.org/TR/html4/loose.dtd>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>w3</span><span>.</span><span>org</span><span>/</span><span>TR</span><span>/</span><span>html4</span><span>/</span><span>loose</span><span>.</span><span>dtd</span></a>">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">  <title>HelloWorld page</title>  </head>  <body>      Greeting : ${greeting}  </body>  </html> |

#### Step 5: Add Configuration Class

Add the below mentioned class under src/main/java with specified package as shown below. This configuration class can be treated as a replacement of spring-servlet.xml as it contains all the information required for component-scanning and view resolver.

com.websystique.springmvc.configuration.HelloWorldConfiguration

|  |
| --- |
| package com.websystique.springmvc.configuration;    import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.ComponentScan;  import org.springframework.context.annotation.Configuration;  import org.springframework.web.servlet.ViewResolver;  import org.springframework.web.servlet.config.annotation.EnableWebMvc;  import org.springframework.web.servlet.view.InternalResourceViewResolver;  import org.springframework.web.servlet.view.JstlView;    @Configuration  @EnableWebMvc  @ComponentScan(basePackages = "com.websystique.springmvc")  public class HelloWorldConfiguration {      @Bean      public ViewResolver viewResolver() {          InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();          viewResolver.setViewClass(JstlView.class);          viewResolver.setPrefix("/WEB-INF/views/");          viewResolver.setSuffix(".jsp");            return viewResolver;      }    } |

@Configuration indicates that this class contains one or more bean methods annotated with @Bean producing bean manageable by spring container. Above Configuration class is equivalent to following XML counterpart:

|  |
| --- |
| <beans xmlns="<a class="vglnk" href="<http://www.springframework.org/schema/beans>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>beans</span></a>"      xmlns:context="<a class="vglnk" href="<http://www.springframework.org/schema/context>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>context</span></a>"      xmlns:mvc="<a class="vglnk" href="<http://www.springframework.org/schema/mvc>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>mvc</span></a>"      xmlns:xsi="<a class="vglnk" href="<http://www.w3.org/2001/XMLSchema-instance>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>w3</span><span>.</span><span>org</span><span>/</span><span>2001</span><span>/</span><span>XMLSchema</span><span>-</span><span>instance</span></a>"      xsi:schemaLocation="<a class="vglnk" href="<http://www.springframework.org/schema/beans>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>beans</span></a> <a class="vglnk" href="<http://www.springframework.org/schema/beans/spring-beans-4.0.xsd>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>beans</span><span>/</span><span>spring</span><span>-</span><span>beans</span><span>-</span><span>4</span><span>.</span><span>0</span><span>.</span><span>xsd</span></a>      <a class="vglnk" href="<http://www.springframework.org/schema/mvc>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>mvc</span></a> <a class="vglnk" href="<http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>mvc</span><span>/</span><span>spring</span><span>-</span><span>mvc</span><span>-</span><span>4</span><span>.</span><span>0</span><span>.</span><span>xsd</span></a>      <a class="vglnk" href="<http://www.springframework.org/schema/context>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>context</span></a> <a class="vglnk" href="<http://www.springframework.org/schema/context/spring-context-4.0.xsd>" rel="nofollow"><span>http</span><span>://</span><span>www</span><span>.</span><span>springframework</span><span>.</span><span>org</span><span>/</span><span>schema</span><span>/</span><span>context</span><span>/</span><span>spring</span><span>-</span><span>context</span><span>-</span><span>4</span><span>.</span><span>0</span><span>.</span><span>xsd</span></a>">        <context:component-scan base-package="com.websystique.springmvc" />        <mvc:annotation-driven />        <bean          class="org.springframework.web.servlet.view.InternalResourceViewResolver">          <property name="prefix">              <value>/WEB-INF/views/</value>          </property>          <property name="suffix">              <value>.jsp</value>          </property>      </bean>    </beans> |

@EnableWebMvc is equivalent to mvc:annotation-driven in XML. It enables support for @Controller-annotated classes that use @RequestMapping to map incoming requests to specific method.

@ComponentScan is equivalent to context:component-scan base-package="..." providing with where to look for spring managed beans/classes.

#### Step 6: Add Initialization class

Add an initializer class implementing WebApplicationInitializer under src/main/java with specified package as shown below(which in this case acts as replacement of any spring configuration defined in web.xml). During Servlet 3.0 Container startup, this class will be loaded and instantiated and its onStartup method will be called by servlet container.

com.websystique.springmvc.configuration.HelloWorldInitializer

|  |
| --- |
| package com.websystique.springmvc.configuration;    import javax.servlet.ServletContext;  import javax.servlet.ServletException;  import javax.servlet.ServletRegistration;    import org.springframework.web.WebApplicationInitializer;  import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;  import org.springframework.web.servlet.DispatcherServlet;    public class HelloWorldInitializer implements WebApplicationInitializer {        public void onStartup(ServletContext container) throws ServletException {            AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();          ctx.register(HelloWorldConfiguration.class);          ctx.setServletContext(container);            ServletRegistration.Dynamic servlet = container.addServlet("dispatcher", new DispatcherServlet(ctx));            servlet.setLoadOnStartup(1);          servlet.addMapping("/");      }    } |

The content above resembles the content of web.xml from previous tutorial as we are using the front-controller DispatherServler, assigning the mapping (url-pattern in xml) and instead of providing the path to spring configuration file(spring-servlet.xml) , here we are registering the Configuration Class. Overall, we are doing the same thing, just the approach is different.

**UPDATE:** Note that now you can write the above class even more concisely [**and it’s the preferred way**], by extendingAbstractAnnotationConfigDispatcherServletInitializer base class, as shown below:

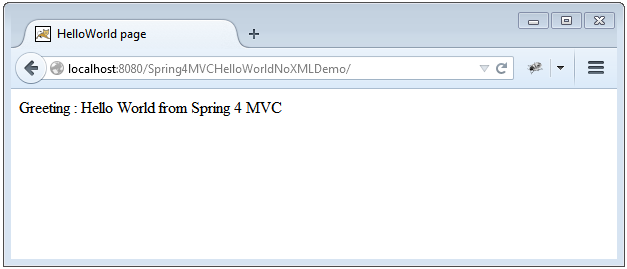
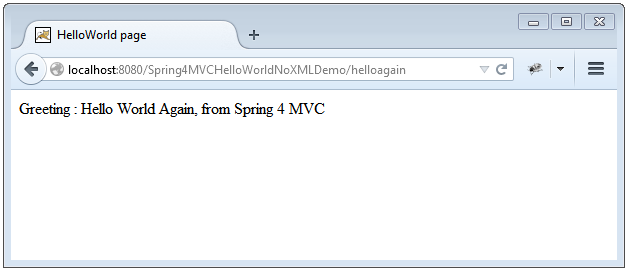
|  |
| --- |
| package com.websystique.springmvc.configuration;    import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;    public class HelloWorldInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {        @Override      protected Class<?>[] getRootConfigClasses() {          return new Class[] { HelloWorldConfiguration.class };      }        @Override      protected Class<?>[] getServletConfigClasses() {          return null;      }        @Override      protected String[] getServletMappings() {          return new String[] { "/" };      }    } |

#### Step 7: Build and Deploy the application

One thing to keep in mind that the Spring java based configuration api’s like WebApplicationInitializer depends on Servlet 3.0 containers.So make sure you don’t have any web.xml with servlet declaration less than 3.0. For our case, we have removed web.xml file from our application.

Now build the war (either by eclipse as was mentioned in last tutorial) or via maven command line( mvn clean install). Deploy the war to a Servlet 3.0 container . Since here i am using Tomcat, i will simply put this war file into tomcat webapps folder and click on start.bat inside tomcat bin directory.

Run the application

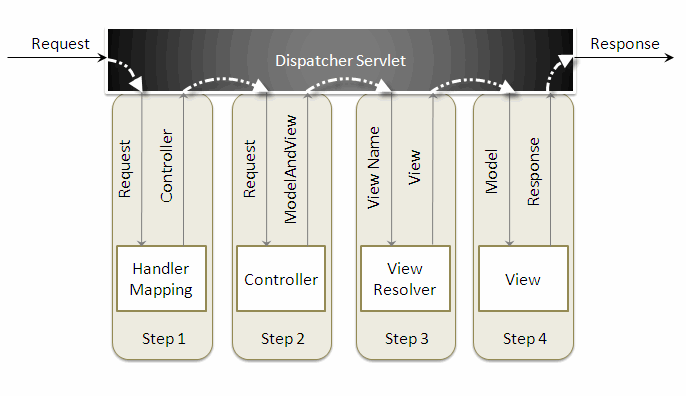
  


That’s it.

# Spring MVC Framework Tutorial

Spring MVC helps in building flexible and loosely coupled web applications. The Model-view-controller design pattern helps in seperating the business logic, presentation logic and navigation logic. Models are responsible for encapsulating the application data. The Views render response to the user with the help of the model object . Controllers are responsible for receiving the request from the user and calling the back-end services.

The figure below shows the flow of request in the Spring MVC Framework.

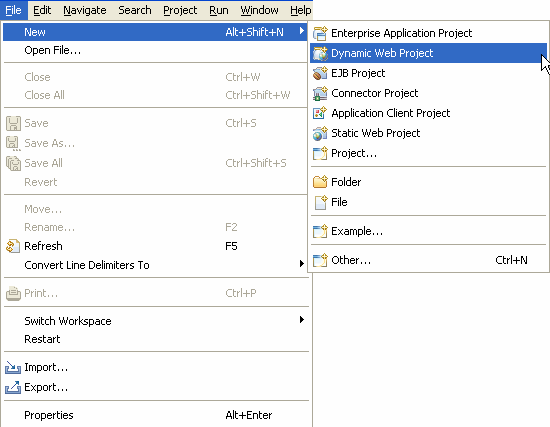


When a request is sent to the Spring MVC Framework the following sequence of events happen.

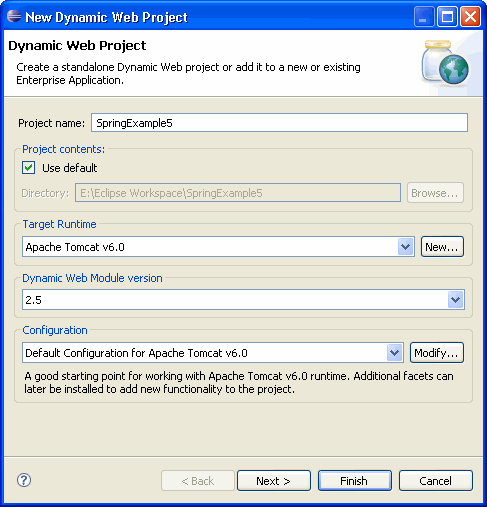
* The *DispatcherServlet* first receives the request.
* The *DispatcherServlet* consults the *HandlerMapping* and invokes the *Controller* associated with the request.
* The *Controller* process the request by calling the appropriate service methods and returns a *ModeAndView* object to the *DispatcherServlet*. The *ModeAndView* object contains the model data and the view name.
* The *DispatcherServlet* sends the view name to a *ViewResolver* to find the actual *View* to invoke.
* Now the *DispatcherServlet* will pass the model object to the *View* to render the result.
* The *View* with the help of the model data will render the result back to the user.

To understand the Spring MVC Framework we will now create a simple hello world example using the Eclipse IDE. I am using Exclipse IDE 3.4 , Spring IDE plugin, Tomcat 6.0 and Spring 3.0 to demonstrate this example.

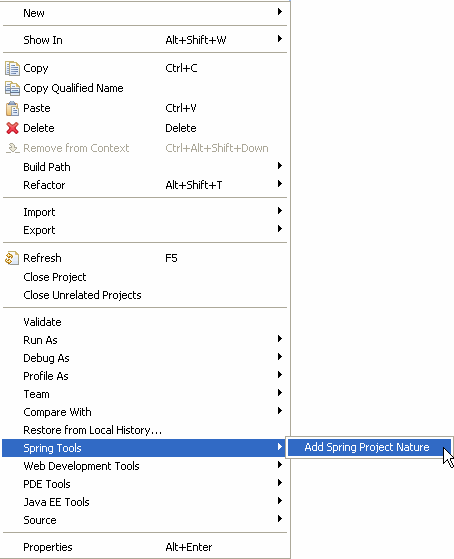
Go to*File -> New -> Dynamic Web Project*, to create a web project.



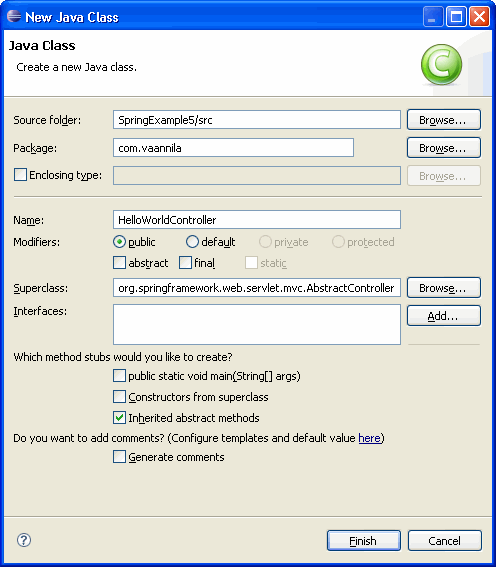
 Enter the project name and click the *Finish* button.



 Right click the project folder, and select *Spring Tools -> Add Spring Project Nature*, to add Spring capabilities to the web project. This feature will be available once you install the Spring IDE.



Create a new package *com.vaannila* inside the *src* directory. The Spring controller class extends *org.springframework.web.servlet.mvc.AbstractController* class. To create a new controller class right click the *src* directory and create a new java class, enter the controller class name and super class name and the *Finish* button.



 Copy the following code inside the *HelloWorldController* class.

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.mvc.AbstractController;

public class HelloWorldController extends AbstractController {

private String message;

@Override

protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response) throws Exception {

return new ModelAndView("welcomePage","welcomeMessage", message);

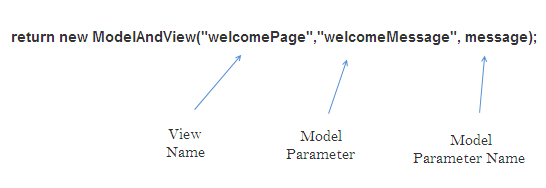
}

public void setMessage(String message) {

this.message = message;

}

}



 The *DispatcherSevlet*, as the name indicates, is a single servlet that manages the entire request-handling process. When a request is sent to the *DispatcherServlet* it delegates the job by invoking the appropriate controllers to process the request. Like any other servlet the *DispatcherServlet* need to be configured in the web deployment descriptor as shown.

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd" id="WebApp\_ID" version="2.5">

<servlet>

<servlet-name>dispatcher</servlet-name>

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

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>\*.htm</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>redirect.jsp</welcome-file>

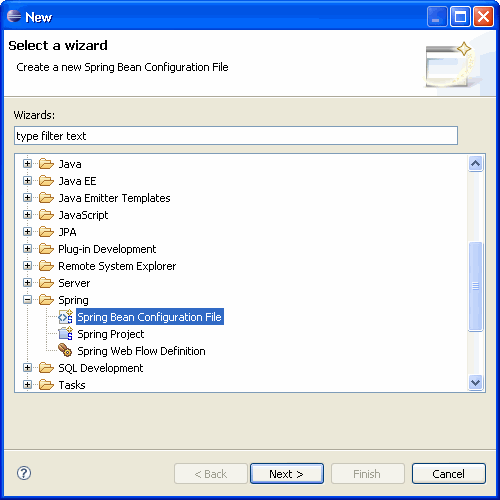
</welcome-file-list>

</web-app>

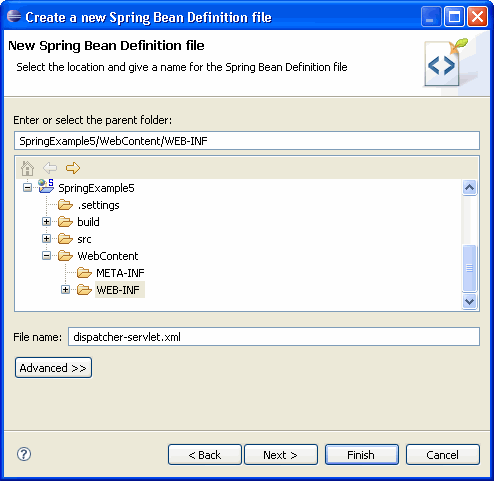
Here the servlet name is *dispatcher*. By default the *DispatcherServlet* will look for a file name *dispatcher-servlet.xml* to load the Spring MVC configuration. This file name is formed by concatenating the servlet name ("*dispatcher*") with "*-servlet.xml*". Here we user the the *url-pattern*as "*.htm*" inorder to hide the implementations technology to the users.

The*redirect.jsp* will be invoked first when we execute the Spring web application. This is the only *jsp*file outside the *WEB-INF* directory and it is here to provide a redirect to the *DispatcherServlet*. All the other views should be stored under the *WEB-INF* directory so that they can be invoked only through the controller process.

To create a bean configuration file right click the *WebContent* folder and select*New -> Other*. The following dialog box appears.



Select the Spring Bean Configuration file and click Next.



Enter the file name as "*dispatcher-servlet.xml*" and click the *Finish* button.

Now the Spring bean configuration file is created, we need to configure the *Controller* and the *ViewResolver* classes. The following code shows how to do this.

<bean id="viewResolver"

class=" org.springframework.web.servlet.view. InternalResourceViewResolver" >

<property name="prefix">

<value>/WEB-INF/jsp/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

<bean name="/welcome.htm" class="com.vaannila.HelloWorldController" >

<property name="message" value="Hello World!" />

</bean>

</beans>

First let's understand how to configure the controller.

<bean name="/welcome.htm" class="com.vaannila.HelloWorldController" >

<property name="message" value="Hello World!" />

</bean>

Here the *name* attribute of the *bean* element indicates the URL pattern to map the request. Since the *id* attribute can't contain special characters like "*/*" , we specify the URL pattern using the *name*attribute of the *bean* element. By default the *DispatcherServlet* uses the *BeanNameUrlHandlerMapping* to map the incoming request. The *BeanNameUrlHandlerMapping*uses the bean name as the URL pattern. Since *BeanNameUrlHandlerMapping* is used by default, you need not do any seperate configuration for this.

We set the message attribute of the *HelloWorldController* class thru setter injection. The *HelloWorldController* class is configured just like an another JavaBean class in the Spring application context, so like any other JavaBean we can set values to it through Dependency Injection(DI).

The*redirect.jsp* will redirect the request to the *DispatcherServlet*, which inturn consults with the *BeanNameUrlHandlerMapping* and invokes the *HelloWorldController*. The *handleRequestInternal()* method in the HelloWorldController class will be invoked. Here we return the *message* property under the name *welcomeMessage* and the view name *welcomePage*to the *DispatcherServlet*. As of now we only know the view name, and to find the actual view to invoke we need a *ViewResolver*.

The *ViewResolver* is configured using the following code.

<bean id="viewResolver"

class=" org.springframework.web.servlet.view.InternalResourceViewResolver" >

<property name="prefix">

<value>/WEB-INF/jsp/</value>

</property>

<property name="suffix">

<value>.jsp</value>

</property>

</bean>

Here the *InternalResourceViewResolver* is used to resolve the view name to the actual view. The *prefix value + view name + suffix value* will give the actual view location. Here the actual view location is */WEB-INF/jsp/welcomePage.jsp*

The following library files are needed to run the example.

antlr-runtime-3.0

commons-logging-1.0.4

org.springframework.asm-3.0.0.M3

org.springframework.beans-3.0.0.M3

org.springframework.context-3.0.0.M3

org.springframework.context.support-3.0.0.M3

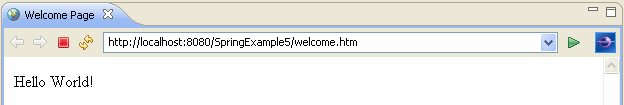
org.springframework.core-3.0.0.M3

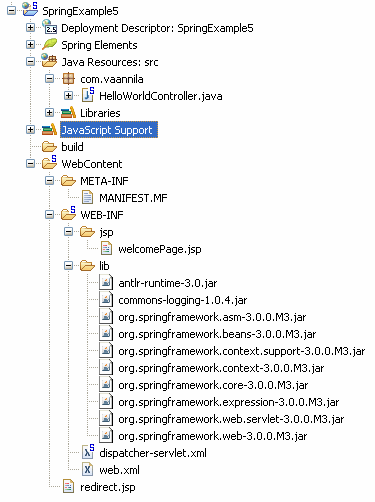
org.springframework.expression-3.0.0.M3

org.springframework.web-3.0.0.M3

org.springframework.web.servlet-3.0.0.M3

 To execute the example run the *redirect.jsp* file. The following page will be displayed.





# Spring MVC and Java-Based Configuration

In this aicle, we will see how to configure a Spring MVC application without using a web.xml. We will use Java-based configuration.

For this example, we will use a simple Maven web project.

## Step 1: Create a Pom.xml for the Required Libraries

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>

 <artifactId>SpringWebExample</artifactId>

  <packaging>war</packaging>

  <version>0.0.1-SNAPSHOT</version>

 <name>SpringWebExample Maven Webapp</name>

  <url>http://maven.apache.org</url>

  <properties>

       <java-version>1.7</java-version>

  </properties>

    <dependencies>

       <dependency>

           <groupId>org.springframework</groupId>

           <artifactId>spring-webmvc</artifactId>

           <version>4.3.0.RELEASE</version>

       </dependency>

       <dependency>

           <groupId>javax.servlet</groupId>

           <artifactId>javax.servlet-api</artifactId>

           <version>3.0.1</version>

           <scope>provided</scope>

       </dependency>

       <dependency>

           <groupId>javax.servlet</groupId>

           <artifactId>jstl</artifactId>

           <version>1.2</version>

       </dependency>

       <dependency>

          <groupId>junit</groupId>

          <artifactId>junit</artifactId>

          <version>3.8.1</version>

          <scope>test</scope>

        </dependency>

    </dependencies>

    <build>

       <finalName>HelloWorld</finalName>

       <pluginManagement>

           <plugins>

               <plugin>

                   <groupId>org.apache.maven.plugins</groupId>

                   <artifactId>maven-compiler-plugin</artifactId>

                   <version>2.3.2</version>

                   <configuration>

                       <source>${java-version}</source>

                       <target>${java-version}</target>

                   </configuration>

               </plugin>

               <plugin>

                   <groupId>org.apache.maven.plugins</groupId>

                   <artifactId>maven-war-plugin</artifactId>

                   <version>2.4</version>

                   <configuration>

                       <warSourceDirectory>src/main/webapp</warSourceDirectory>

                       <warName>SpringWebExample</warName>

                       <failOnMissingWebXml>false</failOnMissingWebXml>

                   </configuration>

               </plugin>

           </plugins>

       </pluginManagement>

    </build>

</project>

We're using Spring 4.3.0 and Servlet 3.

## Step 2: Creating the SpringConfig Class

As we want to do Java-based configuration, we will create a class called SpringConfig, where we will register all Spring-related beans using Spring's Java-based configuration style.

This class will replace the need to create a SpringApplicationContext.xml file, where we use two important tags

<context:component-scan/>

<mvc:annotation-driven/>

Please note that this class has to extend the

org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter class.

Let's see the class.

**SpringConfig.java**:

package com.example.anotatedconfiguration;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.ViewResolver;

import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

import org.springframework.web.servlet.view.JstlView;

@EnableWebMvc

@ComponentScan(basePackages = "com.example")

public class SpringConfig extends WebMvcConfigurerAdapter{

   @Bean

    public ViewResolver viewResolver() {@Configuration

       InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

       viewResolver.setViewClass(JstlView.class);

       viewResolver.setPrefix("/WEB-INF/pages/");

       viewResolver.setSuffix(".jsp");

       return viewResolver;

    }

    @Override

    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

       configurer.enable();

    }

}

Please note that here, we will use three different annotations at the top level. They will serve the purpose of the XML-based tags used earlier.

|  |  |  |
| --- | --- | --- |
| XML Tag | Annotation | Description |
| <context:component-scan/> | @ComponentScan() | Scan starts from base package and registers all controllers, repositories, service, beans, etc. |
| <mvc:annotation-driven/> | @EnableWebMvc | Enable Spring MVC-specific annotations like @Controller |
| Spring config file | @Configuration | Treat as the configuration file for Spring MVC-enabled applications. |

Also, we use the @Bean tag to register ViewResolver. We use InternalResourceViewResolver.

## Step 3: Replacing Web.xml

Create another class, which will replace our traditional web.xml. We use Servlet 3.0 and extend the org.springframework.web.WebApplicationInitializer class.

**WebServletConfiguration.java**:

package com.example.anotatedconfiguration;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;

import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

import org.springframework.web.servlet.DispatcherServlet;

public class WebServletConfiguration implements WebApplicationInitializer{

   public void onStartup(ServletContext ctx) throws ServletException {

       AnnotationConfigWebApplicationContext webCtx = new AnnotationConfigWebApplicationContext();

       webCtx.register(SpringConfig.class);

       webCtx.setServletContext(ctx);

       ServletRegistration.Dynamic servlet = ctx.addServlet("dispatcher", new DispatcherServlet(webCtx));

       servlet.setLoadOnStartup(1);

       servlet.addMapping("/");

    }

}

Here we provide our SpringConfig class and add DispatcherServlet, which acts as the FrontController of the Spring MVC application.

SpringConfig class is the source of Spring beans, before which we used contextConfiglocation.

## Step 4: Create a Controller Class

Now we will create a Controller class, Which will take a parameter from request URL and greet a message in the browser.

package com.example.anotatedconfiguration;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class GreetController {

@RequestMapping(path= "/greet/{name}",method=RequestMethod.GET)

public String greet(@PathVariable String name, ModelMap model){

    String greet =" Hello !!!" + name + " How are You?";

    model.addAttribute("greet", greet);

    System.out.println(greet);

return "greet";

}

}

## Step 5: Create a JSP Page to Show the Message

So, with all the configuration

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

<html>

<head>

<%@ page isELIgnored="false" %>

</head>

<h1>Welcome to Spring 4 and Servlet 3 Based Application</h1>

<body>

<div>

</div>

</body>

</html>

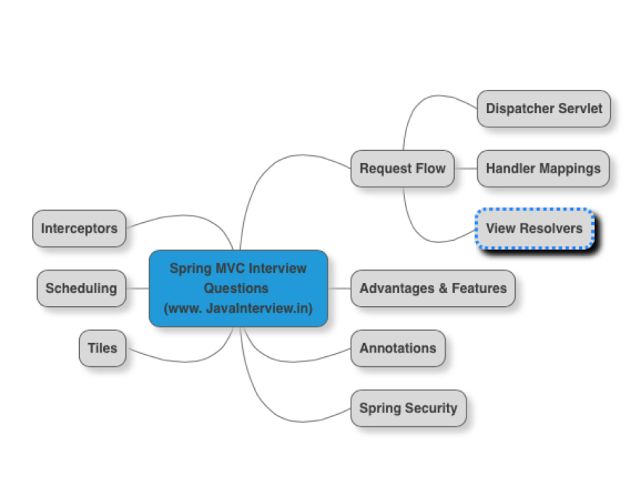
Please note that here that I use <%@ page isELIgnored="false" %>. If the JSTL version is old (before 1.2), it can’t  evaluate ${} EL. If your JSTL version is 1.2, you can safely remove that tag.

**Output**:

http://localhost:8080/SpringWebExample/greet/shamik

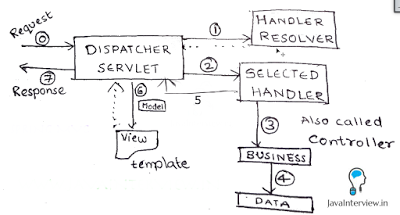
Hello !!!shamik How are You?

# Spring MVC Interview Questions and Answers

[](http://4.bp.blogspot.com/-6W72sFW3v4I/VVDVvI_L_EI/AAAAAAAAAIs/FXHW_NoOR5I/s1600/SpringMVCInterviewQuestions.png)

1. [How does request flow happen in Spring MVC?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question1)
2. [Can you list a few advantages of using Spring MVC framework?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question2)
3. [Give examples of important Spring MVC annotations?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question3)
4. [Can you explain the concept of Interceptors in Spring MVC?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question4)
5. [Interceptors can be configured using the interceptors property.](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question5)
6. [How do you schedule tasks with Spring?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question6)
7. [How do you integrate Spring MVC with tiles?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question7)
8. [How do you configure Spring MVC web application to use UTF-8 encoding for handling forms?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question8)
9. [How do you enable spring security for a web application?](http://www.javainterview.in/p/spring-mvc-interview-questions.html#question9)

### How does request flow happen in Spring MVC?

[](http://4.bp.blogspot.com/-iz_lkHWqkNM/VVDHgFUkLsI/AAAAAAAAAHY/i9l_PNDFe3U/s1600/Spring+MVC+Request+Flow.png)

Shown in the picture below. DispatcherServlet acts as the front controller. Simplified actions taken by DispatcherServlet are listed below.

* All requests arrive at the DispatcherServlet (Front Controller) - STEP 0 in Figure
* DispatcherServlet resolves theme and locale as configured.
* Find’s appropriate Controller (Handler) to handle the request. (pre-processors and post-processors, if configured) (STEP 1)
* Redirect to the Controller (Handler) - STEP 2. Controller executes the request and returns a view name and a view model object. (STEP 3,4,5)
* DispatcherServlet resolves the view name and redirects to the view template. The response html is returned to DispatcherServlet. (STEP 6)
* DispatcherServlet send the response back to the browser. (STEP 7)

### Can you list a few advantages of using Spring MVC framework?

* In Spring Web MVC, any POJO can be used as a command or form-backing object.
* Highly flexible databinding – If there is a type mismatch, it is shown as a validation error on the screen. Business POJO’s can directly be used as form-backing objects.
* Flexible view resolution: Controller can either select a view name and prepare model map for it or write directly to response stream.
* Supports JSP, Velocity and Freemarker view technologies.
* Can directly generate XML, JSON, Atom, and many other types of content.
* Highly convenient tag library.

### Give examples of important Spring MVC annotations?

Important Spring MVC annotations are listed below.

* @Controller : This class would serve as a controller.
* @RequestMapping : Can be used on a class or a method. Maps an url on the class (or method).
* @PathVariable : Used to map a dynamic value in the url to a method argument.

Example 1 : Maps a url “/players “ for the controller method.

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

public String findAllPlayers(Model model) {

Example 2 : If a url /players/15 is keyed in, playerId is populated with value 15.

@RequestMapping(value="/players/{playerid}", method=RequestMethod.GET)

public String findPlayer(@PathVariable String playerId, Model model) {

### Can you explain the concept of Interceptors in Spring MVC?

Handler interceptors are used when you want to apply specific functionality to certain requests. Handler Interceptors should implement the interface HandlerInterceptor.

Three methods are defined:

* preHandle(..) is called before the actual handler is executed;
* postHandle(..) is called after the handler is executed;
* afterCompletion(..) is called after the complete request has finished.

### Interceptors can be configured using the interceptors property.

<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<list>

<ref bean="yourCustomHandlerInterceptor"/>

</list>

</property>

</bean>

### How do you schedule tasks with Spring?

Spring 3.0 introduced TaskScheduler abstract to deal with scheduling jobs. Spring has support for Timer (Jdk) and Quartz. Sample methods in the interface TaskScheduler are shown below:

ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);

Scheduling can also be done using an annotation

@Scheduled(fixedDelay=5000)

public void doSomething() {

// something that should execute periodically

}

Below example shows scheduling with xml configuration

<task:scheduler id="customScheduler" pool-size="30"/>

<task:scheduled-tasks scheduler=" customScheduler ">

<task:scheduled ref="someBean" method="someOtherMethod" fixed-delay="5000" initial-delay="1000"/>

<task:scheduled ref="someOtherBean" method="someMethod" cron="\*/5 \* \* \* \* MON-FRI"/>

</task:scheduled-tasks>

### How do you enable spring security for a web application?

Spring Security is used to implement Authentication and Authorization for a web application. We can enable spring security by configuring an appropriae security filter. Example shown below. We can create a separate security-context.xml to define the authentication and authorization roles and accesses.

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>

org.springframework.web.filter.DelegatingFilterProxy

</filter-class>

</filter>

<filter-mapping>

<filter-name>springSecurityFilterChain</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

**Spring Interceptor Tutorial**

Spring Interceptors has the ability to pre-handle and post-handle the web requests. Each interceptor class should extend the HandlerInterceptorAdapter class. Here we will create a Logger Interceptor by extending the HandlerInterceptorAdapter class. You can override any of the three callback methods preHandle(), postHandle() and afterCompletion(). As the names indicate the preHandle() method will be called before handling the request, the postHandle() method will be called after handling the request and the afterCompletion() method will be called after rendering the view.

In each method we will log information using log4j. First instantiate the logger in the static context, then set up the basic configuration so that the log messages will be logged on the console.

The LoggerInterceptor class is shown below.

package com.vaannila.interceptor;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.apache.log4j.BasicConfigurator;

import org.apache.log4j.Logger;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.handler. HandlerInterceptorAdapter;

public class LoggerInterceptor extends HandlerInterceptorAdapter {

static Logger logger = Logger.getLogger(LoggerInterceptor.class);

static{

BasicConfigurator.configure();

}

@Override

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

logger.info("Before handling the request");

return super.preHandle(request, response, handler);

}

@Override

public void postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {

logger.info("After handling the request");

super.postHandle(request, response, handler, modelAndView);

}

@Override

public void afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

throws Exception {

logger.info("After rendering the view");

super.afterCompletion(request, response, handler, ex);

}

}

 Now the logger interceptor is created you need to associate this interceptor with the handler mapping. Here we use BeanNameUrlHandlerMapping, incase you are using more than one handler mapping you need to associate the interceptor with each one of them. The code below shows how to associate an interceptor with the handler mapping.

<?xml version="1.0" encoding="UTF-8"?>

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

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

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

<bean id="handlerMapping" class="org.springframework.web.servlet.handler. BeanNameUrlHandlerMapping" p:interceptors-ref="loggerInterceptor" />

<bean id="loggerInterceptor" class="com.vaannila.interceptor.LoggerInterceptor" />

<bean id="userService" class="com.vaannila.service.UserServiceImpl" />

<bean name="/userRegistration.htm" class="com.vaannila.web.UserController" p:userService-ref="userService" p:formView="userForm" p:successView="userSuccess" />

</beans>

When you execute the example you can see the log messages getting dispalyed on the console.

**Using Spring Interceptors in your MVC Webapp**

I thought that it was time to take a look at Spring’s MVC interceptor mechanism, which has been around for a good number of years and is a really useful tool.   
  
A Spring Interceptor does what it says on the tin: intercepts an incoming HTTP request before it reaches your Spring MVC controller class, or conversely, intercepts the outgoing HTTP response after it leaves your controller, but before it’s fed back to the browser.  
  
You may ask what use is this to you? The answer is that it allows you to perform tasks that are common to every request or set of requests without the need to cut ‘n’ paste boiler plate code into every controller class. For example, you could perform user authentication of a request before it reaches your controller and, if successful, retrieve some additional user details from a database adding them to the HttpServletRequest object before your controller is called. Your controller can then simply retrieve and use these values or leave them for display by the JSP. On the other hand, if the authentication fails, you could re-direct your user to a different page.  
  
The demonstration code shows you how to modify the incoming HttpServletRequest object before it reaches your controller. This does nothing more than add a simple string to the request, but, as I said above, you could always make a database call to grab hold of some data that’s required by every request... you could even add some kind of [optimization](http://www.captaindebug.com/2011/04/code-optimization.html) and do some caching at this point.

public class RequestInitializeInterceptor extends HandlerInterceptorAdapter {

// Obtain a suitable logger.

private static Log logger = LogFactory

.getLog(RequestInitializeInterceptor.class);

/\*\*

\* In this case intercept the request BEFORE it reaches the controller

\*/

@Override

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

try {

logger.info("Intercepting: " + request.getRequestURI());

// Do some changes to the incoming request object

updateRequest(request);

return true;

} catch (SystemException e) {

logger.info("request update failed");

return false;

}

}

/\*\*

\* The data added to the request would most likely come from a database

\*/

private void updateRequest(HttpServletRequest request) {

logger.info("Updating request object");

request.setAttribute("commonData",

"This string is required in every request");

}

/\*\* This could be any exception \*/

private class SystemException extends RuntimeException {

private static final long serialVersionUID = 1L;

// Blank

}

}

In the code above, I’ve chosen the simplest implementation method by extending the HandlerInterceptorAdaptor class, overriding preHandle(..) method. My preHandle(...) method does the error handling, deciding what to do if an error occurs and returning false if one does. In returning false the interceptor chain is broken and your controller class is not called. The actual business of messing with the request object is delegated to updateRequest(request).

The HandlerInterceptorAdaptor class has three methods, each of which are stubbed and, if desired, can be ignored. The methods are: prehandle(...), postHandle(...) and afterCompletion(...) and more information on these can be found in the [Spring API documentation](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/). Be aware that this can be somewhat confusing as the Handler Interceptor classes documentation still refer to MVC controller classes by their Spring 2 name of handlers. This point is easily demonstrated if you look at prehandle(...)’s third parameter of type Object and called handler. If you examine this in your debugger, you’ll see that it is an instance of your controller class. If you’re new to this technique, just remember that controller == handler.

The next step in implementing an interceptor is, as always, to add something to the Spring XML config file:

<!-- Configures Handler Interceptors -->

<mvc:interceptors>

<!-- This bit of XML will intercept all URLs - which is what you want in a web app -->

<bean class="marin.interceptor.RequestInitializeInterceptor" />

<!-- This bit of XML will apply certain URLs to certain interceptors -->

<!--

<mvc:interceptor>

<mvc:mapping path="/gb/shop/\*\*"/>

<bean class="marin.interceptor.RequestInitializeInterceptor" />

</mvc:interceptor>

-->

</mvc:interceptors>

The XML above demonstrates an either/or choice of adding an interceptor to all request URLs, or if you look at the commented out section, adding an interceptor to specific request URLs, allowing you to choose which URLs are connected to your interceptor class.

The eagle eyed readers may have noticed that the interceptor classes use inheritance and XML config as its method of implementation. In these days of convention over configuration, this pattern is beginning to look a little jaded and could probably do with a good overhaul. One suggestion would be to enhance the whole lot to use annotations, applying the same techniques that have already been added to the controller mechanism. This would add extra flexibility without the complication of using all the interfaces and abstract base classes. As a suggestion, a future interceptor class implementation could look something like this:

@Intercept(value = "/gb/en/\*", method = RequestMethod.POST)

public boolean myAuthenticationHandler(HttpServletRequest request,

Model model) {

// Put some code here

}

Spring MVC 4 - How to intercept request with a HandlerInterceptor

**Technologies used:** Java SE 1.8 | Spring 4.3.10.RELEASE | Maven 3.3.9 | Eclipse Neon.3 | Apache Tomcat  7.0.47

In this post, I will show you how to use the HandlerInterceptor interface and HandlerInterceptorAdapter class in Spring MVC application to intercept the web requests.

The HandlerInterceptor interface has three abstract methods as-

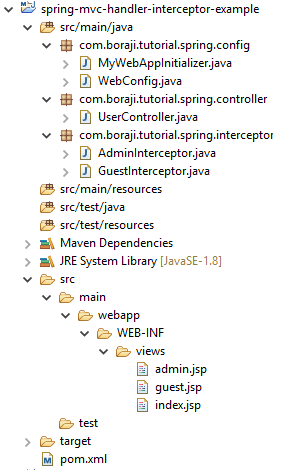
* **preHandle()** - It is called before the handler method of a controller class is invoked.
* **afterCompletion()** - It is called after completion of a request i.e. after render the view.
* **postHandle()** - It is called after the handler method of a controller class is invoked, but before rendering the view.

The HandlerInterceptorAdapter class is an implementation of the HandlerInterceptor interface, so use this class if you want to override the specific method.

To create a custom handler interceptor, you can use either the HandlerInterceptor interface or HandlerInterceptorAdapter class. Let’s see the complete example to demonstrate the above.

### Project structure

Review the following web project structure build using Maven.



Related - [How to create a web project using maven build tool in eclipse IDE](https://www.boraji.com/how-to-create-a-web-project-using-maven-in-eclipse).

### Jar dependencies

Open pom.xml file of your maven project and add the following dependencies in it.

<dependencies>

*<!-- Spring MVC Dependency -->*

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.10.RELEASE</version>

</dependency>

*<!-- JSTL Dependency -->*

<dependency>

<groupId>javax.servlet.jsp.jstl</groupId>

<artifactId>javax.servlet.jsp.jstl-api</artifactId>

<version>1.2.1</version>

</dependency>

<dependency>

<groupId>taglibs</groupId>

<artifactId>standard</artifactId>

<version>1.1.2</version>

</dependency>

*<!-- Servlet Dependency -->*

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

*<!-- JSP Dependency -->*

<dependency>

<groupId>javax.servlet.jsp</groupId>

<artifactId>javax.servlet.jsp-api</artifactId>

<version>2.3.1</version>

<scope>provided</scope>

</dependency>

</dependencies>

### Custom handler interceptor

Create a handler interceptor class, named as GuestInterceptor, by implementing the HandlerInterceptor interface as follows.

**GuestInterceptor.java**

**package** com.boraji.tutorial.spring.interceptor;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** org.springframework.web.servlet.HandlerInterceptor;

**import** org.springframework.web.servlet.ModelAndView;

**public** **class** **GuestInterceptor** **implements** **HandlerInterceptor** {

*// Called before handler method invocation*

@Override

**public** **boolean** **preHandle**(HttpServletRequest req, HttpServletResponse res,

Object handler) **throws** Exception {

System.out.println("Called before handler method");

req.setAttribute("fname", "Elizabeth");

**return** **true**;

}

*// Called after handler method request completion, before rendering the view*

@Override

**public** **void** **postHandle**(HttpServletRequest req, HttpServletResponse res,

Object handler, ModelAndView model) **throws** Exception {

System.out.println("Called after handler method request completion,"

+ " before rendering the view");

model.addObject("lname", "Brown");

}

*// Called after rendering the view*

@Override

**public** **void** **afterCompletion**(HttpServletRequest req, HttpServletResponse res,

Object handler, Exception ex) **throws** Exception {

System.out.println("Called after rendering the view");

}

}

Now, create another handler interceptor class, named as AdminInterceptor, by extending the HandlerInterceptorAdapter class and override the postHandle() method as follows.

**AdminInterceptor.java**

**package** com.boraji.tutorial.spring.interceptor;

**import** java.time.LocalTime;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** org.springframework.web.servlet.ModelAndView;

**import** org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

**public** **class** **AdminInterceptor** **extends** **HandlerInterceptorAdapter** {

@Override

**public** **void** **postHandle**(HttpServletRequest req, HttpServletResponse res,

Object handler, ModelAndView model) **throws** Exception {

System.out.println("Called after handler method request completion,"

+ " before rendering the view");

LocalTime time = LocalTime.now();

**int** hrs = time.getHour();

**if** (hrs >= 0 && hrs <= 12) {

model.addObject("greeting", "Good morning!");

} **else** **if** (hrs > 12 && hrs <= 17) {

model.addObject("greeting", "Good afternoon!");

} **else** {

model.addObject("greeting", "Good evening!");

}

}

}

### Spring configuration + Registering interceptors

Create a web @Configuration class annotated with @EnableWebMvc and @ComponentScan.

Override the addInterceptors() method of the WebMvcConfigurerAdapter class to register the custom handler interceptors GuestInterceptor and AdminInterceptor as follows.

**WebConfig.java**

**package** com.boraji.tutorial.spring.config;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;

**import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

**import** org.springframework.web.servlet.view.InternalResourceViewResolver;

**import** com.boraji.tutorial.spring.intercepter.AdminIntercepter;

**import** com.boraji.tutorial.spring.intercepter.GuestIntercepter;

@Configuration

@EnableWebMvc

@ComponentScan(basePackages = { "com.boraji.tutorial.spring.controller" })

**public** **class** **WebConfig** **extends** **WebMvcConfigurerAdapter** {

@Override

**public** **void** **addInterceptors**(InterceptorRegistry registry) {

*// Register guest interceptor with single path pattern*

registry.addInterceptor(**new** GuestInterceptor()).addPathPatterns("/guest");

*// Register admin interceptor with multiple path patterns*

registry.addInterceptor(**new** AdminInterceptor())

.addPathPatterns(**new** String[] { "/admin", "/admin/\*" });

}

@Bean

**public** InternalResourceViewResolver **resolver**() {

InternalResourceViewResolver resolver = **new** InternalResourceViewResolver();

resolver.setPrefix("/WEB-INF/views/");

resolver.setSuffix(".jsp");

**return** resolver;

}

}

### Controller class

Create a simple UserController class to handle the requests for guest and admin users.

**UserController.java**

**package** com.boraji.tutorial.spring.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.RequestAttribute;

**import** org.springframework.web.bind.annotation.RequestMapping;

@Controller

**public** **class** **UserController** {

@RequestMapping("/")

**public** String **index**() {

**return** "index";

}

@RequestMapping("/guest")

**public** String **guestHandler**(@**RequestAttribute**("fname")String fname,Model model) {

model.addAttribute("mname", "Smith");

**return** "guest";

}

@RequestMapping("/admin")

**public** String **adminHandler**(Model model) {

model.addAttribute("name", "Mike");

**return** "admin";

}

}

**JSP Views**

Create a new folders as \WEB-INF\views under src\main\webapp folder.

Now, create index.jsp, guest.jsp and admin.jsp files under src\main\webapp\WEB-INF\views under folder.

**index.jsp**

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

**<!DOCTYPE html>**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>BORAJI.COM</title>

</head>

<body>

<h2>Spring MVC 4 - HandlerInterceptor example</h2>

<p>

<a href="guest">Guest</a> | <a href="admin">Admin</a>

</p>

</body>

</html>

**guest.jsp**

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

**<!DOCTYPE html>**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>BORAJI.COM</title>

</head>

<body>

<h2>Spring MVC 4 - HandlerInterceptor example</h2>

<h3>Guest user name is : ${fname} ${mname} ${lname}</h3>

</body>

</html>

**admin.jsp**

<%@ page language="java" contentType="text/html; charset=ISO-8859-1"

pageEncoding="ISO-8859-1"%>

**<!DOCTYPE html>**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>BORAJI.COM</title>

</head>

<body>

<h2>Spring MVC 4 - HandlerInterceptor example</h2>

<h3>${greeting} ${name}</h3>

</body>

</html>

### Servlet container initialization

Finally, create a container initializer class by extending the AbstractAnnotationConfigDispatcherServletInitializer to bootstrap the Spring MVC application.

**MyWebAppInitializer.java**

**package** com.boraji.tutorial.spring.config;

**import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

**public** **class** **MyWebAppInitializer** **extends** **AbstractAnnotationConfigDispatcherServletInitializer** {

@Override

**protected** Class<?>[] **getRootConfigClasses**() {

**return** **new** Class[] {};

}

@Override

**protected** Class<?>[] **getServletConfigClasses**() {

**return** **new** Class[] { WebConfig.class };

}

@Override

**protected** String[] **getServletMappings**() {

**return** **new** String[] { "/" };

}

}

The AbstractAnnotationConfigDispatcherServletInitializer class, implements the WebApplicationInitializer, is implemented in Servlet 3.0+ environments in order to configure the ServletContext programmatically.

### Build + Deploy + Run application

Use the following maven commands to build, package, deploy and run application.

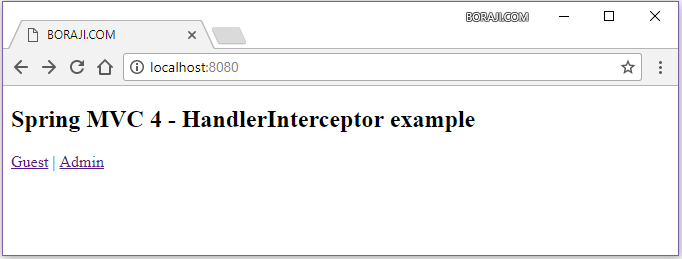
mvn clean install  (This command triggers war packaging)

mvn tomcat7:run (This command run embedded tomcat and deploy war file automatically)

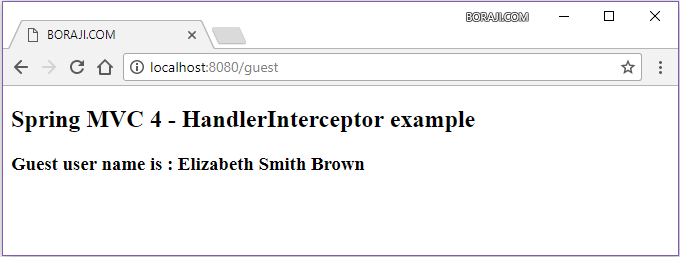
You can refer this [link](https://www.boraji.com/how-to-create-a-web-project-using-maven-in-eclipse#maven-run-tomcat)to learn how to run the above commands in Eclipse IDE.

Type the following URLs in browser's address bar to open the home page.

http://localhost:8080/



Now click on the 'Guest' link to open the guest page.



Now go to home page and click on the 'Admin' link to open the admin page.



(For above example refer <https://www.boraji.com/spring-mvc-4-how-to-intercept-request-with-a-handlerinterceptor>)

**How Spring MVC Really Works**

## Project Setup

Throughout this article, we’ll use the latest and greatest Spring Framework 5. We’re focusing here on the Spring’s classic web stack, which has been available from the very first versions of the framework and is still the primary way of building web applications with Spring.

For starters, to set up your test project, you’ll use Spring Boot and some of its starter dependencies; you’ll also need to define the parent:

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.0.M5</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

</dependencies>

Note that, in order to use Spring 5, you need to also use Spring Boot 2.x. At the time of writing, this is a milestone release, available in the [Spring Milestone Repository](https://repo.spring.io/libs-milestone/org/springframework/boot/spring-boot-starter-web/). Let’s add this repository to your Maven project:

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

## Sample Project

To understand how Spring Web MVC works, you’ll implement a simple application with a login page. To show the login page, create a @Controller-annotated class InternalController with a GET mapping for the context root.

The hello() method is parameterless. It returns a String which is interpreted by Spring MVC as a view name (in our case, the login.html template):

import org.springframework.web.bind.annotation.GetMapping;

@GetMapping("/")

public String hello() {

return "login";

}

To process a user login, create another method that handles POST requests with login data. It then redirects the user either to the success or failure page, depending on the result.

Note that the login() method receives a domain object as an argument and returns a ModelAndViewobject:

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.servlet.ModelAndView;

@PostMapping("/login")

public ModelAndView login(LoginData loginData) {

if (LOGIN.equals(loginData.getLogin())

&& PASSWORD.equals(loginData.getPassword())) {

return new ModelAndView("success",

Collections.singletonMap("login", loginData.getLogin()));

} else {

return new ModelAndView("failure",

Collections.singletonMap("login", loginData.getLogin()));

}

}

ModelAndView is a holder of two distinct objects:

* Model – a key-value map of data used to render the page
* View – a template of the page that is filled with data from the model

These are joined for convenience so that the controller method can return them both at once.

To render your HTML page, use [Thymeleaf](http://www.thymeleaf.org/) as a view template engine, which has solid, out-of-the-box integration with Spring.

## Servlets as the Foundation of a Java Web Application

So, what does actually happen when you type [*http://localhost:8080/*](http://localhost:8080/) in the browser, press Enter, and the request hits the web server? How do you get from this request to seeing a web form in the browser?

Given the project is a simple Spring Boot application, you’ll be able to run it via the Spring5Application.

Spring Boot uses [Apache Tomcat](https://stackify.com/tomcat-performance-monitoring/) by default. Hence, running the application, you are likely to see the following information in the log:

2017-10-16 20:36:11.626 INFO 57414 --- [main]

o.s.b.w.embedded.tomcat.TomcatWebServer :

Tomcat initialized with port(s): 8080 (http)

2017-10-16 20:36:11.634 INFO 57414 --- [main]

o.apache.catalina.core.StandardService :

Starting service [Tomcat]

2017-10-16 20:36:11.635 INFO 57414 --- [main]

org.apache.catalina.core.StandardEngine :

Starting Servlet Engine: Apache Tomcat/8.5.23

Since Tomcat is a Servlet container, naturally every HTTP request sent to a Tomcat web server is processed by a Java servlet. So the Spring Web application entry point is, not surprisingly, a servlet.

[A servlet](https://en.wikipedia.org/wiki/Java_servlet) is, simply put, a core component of any Java web application; it’s low-level and does not impose too much in the way of specific programming patterns, such as MVC.

An HTTP servlet can only receive an HTTP request, process it in some way, and send a response back.

And, starting with the Servlet 3.0 API, you can now move beyond XML configuration and start leveraging Java configuration (with minor restrictions).

## DispatcherServlet as the Heart of Spring MVC

What we really want to do as developers of a web application is to abstract away the following tedious and boilerplate tasks and focus on useful business logic:

* mapping an HTTP request to a certain processing method
* parsing of HTTP request data and headers into data transfer objects (DTOs) or domain objects
* model-view-controller interaction
* generation of responses from DTOs, domain objects, etc.

The Spring [DispatcherServlet](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html) provides exactly that. It is the heart of the Spring Web MVC framework; this core component receives all requests to your application.

As you’ll see, DispatcherServlet is very extensible. For example, it allows you to plug in different existing or new adapters for a lot of tasks:

* map a request to a class or method that should handle it (implementations of the HandlerMappinginterface)
* handle a request using a specific pattern, like a regular servlet, a more complex MVC workflow, or just a method in a POJO bean (implementations of the HandlerAdapter interface)
* resolve views by name, allowing you to use different templating engines, XML, XSLT or any other view technology (implementations of the ViewResolver interface)
* parse multipart requests by using the default Apache Commons file uploading implementation or writing your own MultipartResolver
* resolve locale with any LocaleResolver implementation, including cookie, session, Accept HTTP header, or any other way of determining the locale expected by the user

## Processing of an HTTP Request

First, let’s trace the processing of simple HTTP requests to a method in your controller layer and back to the browser/client.

The DispatcherServlet has a long inheritance hierarchy; it’s worth understanding these individual aspects one by one, top-down. The request processing methods will interest us the most.

### https://lh3.googleusercontent.com/4ahYReme6gkGU8NIU--JzvxgCckXNYzCBa_Fi7Xk6DwqNctyNj0pOB_UPU4Euboy66vfURfovJK8VUgJMTz0ms2yQtFnqEhw9iJnWd_pCCpWN5tNXIYhUFtkCxakO-GTyuKlHoqs

Understanding the HTTP request, both locally during standard development, [as well as remotely](https://stackify.com/prefix-remote-http-calls/), is a critical part of understanding the MVC architecture.

### GenericServlet

GenericServlet is a part of the Servlet specification not directly focused on HTTP. It defines the service()method that receives incoming requests and produces responses.

Note how ServletRequest and ServletResponse method arguments are not tied to the HTTP protocol:

public abstract void service(ServletRequest req, ServletResponse res)

throws ServletException, IOException;

This is the method that is eventually called on any request to the server, including a simple GET request.

### HttpServlet

HttpServlet class is, as the name suggests, the HTTP-focused Servlet implementation, also defined by the specification.

In more practical terms, HttpServlet is an abstract class with a service() method implementation that splits the requests by the HTTP method type and looks roughly like this:

protected void service(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, IOException {

String method = req.getMethod();

if (method.equals(METHOD\_GET)) {

// ...

doGet(req, resp);

} else if (method.equals(METHOD\_HEAD)) {

// ...

doHead(req, resp);

} else if (method.equals(METHOD\_POST)) {

doPost(req, resp);

// ...

}

### HttpServletBean

Next, HttpServletBean is the first Spring-aware class in the hierarchy. It injects the bean’s properties using the servlet init-param values received from the web.xml or from WebApplicationInitializer.

In case of the requests to your application, the doGet(), doPost(), etc methods are called for those specific HTTP requests.

### FrameworkServlet

FrameworkServlet integrates the Servlet functionality with a web application context, implementing the ApplicationContextAware interface. But it is also able to create a web application context on its own.

As you already saw, the HttpServletBean superclass injects init-params as bean properties. So, if a context class name is provided in the contextClass init-param of the servlet, then an instance of this class will be created as an application context. Otherwise, a default XmlWebApplicationContext class will be used.

As XML configuration is out of style nowadays, Spring Boot configures DispatcherServlet with AnnotationConfigWebApplicationContext by default. But you could change that easily.

For example, if you need to configure your Spring Web MVC application with a Groovy-based application context, you could use the following configuration of DispatcherServlet in the web.xmlfile:

dispatcherServlet

org.springframework.web.servlet.DispatcherServlet

contextClass

org.springframework.web.context.support.GroovyWebApplicationContext

The same configuration may be done in a more modern Java-based way using the WebApplicationInitializer class.

### DispatcherServlet: Unifying the Request Processing

The HttpServlet.service() implementation, which routes requests by the type of HTTP verb, makes perfect sense in the context of low-level servlets. However, at the Spring MVC level of abstraction, method type is just one of the parameters that can be used to map the request to its handler.

And so, the other main function of the FrameworkServlet class is to join the handling logic back into a single processRequest() method, which in turn calls the doService() method:

@Override

protected final void doGet(HttpServletRequest request,

HttpServletResponse response) throws ServletException, IOException {

processRequest(request, response);

}

@Override

protected final void doPost(HttpServletRequest request,

HttpServletResponse response) throws ServletException, IOException {

processRequest(request, response);

}

// …

### DispatcherServlet: Enriching the Request

Finally, the DispatcherServlet implements the doService() method. Here, it adds to the request some useful objects that may come in handy down the processing pipeline: web application context, locale resolver, theme resolver, theme source etc.:

request.setAttribute(WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE,

getWebApplicationContext());

request.setAttribute(LOCALE\_RESOLVER\_ATTRIBUTE, this.localeResolver);

request.setAttribute(THEME\_RESOLVER\_ATTRIBUTE, this.themeResolver);

request.setAttribute(THEME\_SOURCE\_ATTRIBUTE, getThemeSource());

Also, the doService() method prepares input and output flash maps. Flash map is basically a pattern to pass parameters from one request to another request that immediately follows. This may be very useful during redirects (like showing the user a one-shot information message after the redirect):

FlashMap inputFlashMap = this.flashMapManager

.retrieveAndUpdate(request, response);

if (inputFlashMap != null) {

request.setAttribute(INPUT\_FLASH\_MAP\_ATTRIBUTE,

Collections.unmodifiableMap(inputFlashMap));

}

request.setAttribute(OUTPUT\_FLASH\_MAP\_ATTRIBUTE, new FlashMap());

Then, the doService() method calls the doDispatch() method that is responsible for request dispatching.

### DispatcherServlet: Dispatching the Request

The main purpose of the dispatch() method is to find an appropriate handler for the request and feed it the request/response parameters. The handler is basically any kind of Object and is not limited to a specific interface. This also means that Spring needs to find an adapter for this handler that knows how to “talk” to the handler.

To find the handler that matches the request, Spring goes through the registered implementations of the [HandlerMapping](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/HandlerMapping.html) interface. There are many different implementations that could suit your needs.

SimpleUrlHandlerMapping allows mapping a request by its URL to a certain processing bean. For example, it can be configured by injecting its mappings property with a java.util.Properties instance similar to this:

/welcome.html=ticketController

/show.html=ticketController

Probably the most widely-used class for handler mapping is RequestMappingHandlerMapping, which maps a request to a @RequestMapping-annotated method of a @Controller class. This is exactly the mapping that connects the dispatcher with the hello() and login() methods of your controller.

Note that your Spring-aware methods are annotated with @GetMapping and @PostMappingcorrespondingly. These annotations, in turn, are marked with the @RequestMappingmeta-annotation.

The dispatch() method also takes care of some other HTTP-specific tasks:

* short-circuiting processing of the GET request in case the resource was not modified
* applying the multipart resolver for corresponding requests
* short-circuiting processing of the request if the handler chose to handle it asynchronously

### Handling the Request

Now that Spring determined the handler for the request and the adapter for the handler, it’s time to finally handle the request. Here’s the signature of the HandlerAdapter.handle() method. It’s important to note that the handler has a choice in how to handle the request:

* Write the data to the response object by itself and return null
* Return a ModelAndView object to be rendered by the DispatcherServlet

@Nullable

ModelAndView handle(HttpServletRequest request,

HttpServletResponse response,

Object handler) throws Exception;

There are several provided types of handlers. Here’s how the SimpleControllerHandlerAdapter processes a Spring MVC controller instance (do not confuse it with a @Controller-annotated POJO).

Notice how the controller handler returns ModelAndView object and does not render the view by itself:

public ModelAndView handle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

return ((Controller) handler).handleRequest(request, response);

}

The second is SimpleServletHandlerAdapter, which adapts a regular Servlet as a request handler.

A Servlet does not know anything about ModelAndView and simply handles the request by itself, rendering the result into the response object. So this adapter simply returns null instead ofModelAndView:

public ModelAndView handle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

((Servlet) handler).service(request, response);

return null;

}

In your case, a controller is a POJO with several @RequestMapping annotations, so any handler is basically a method of this class wrapped in a HandlerMethod instance. To adapt to this handler type, Spring uses the RequestMappingHandlerAdapter class.

### Processing Arguments and Return Values of Handler Methods

Note that the controller methods do not usually take HttpServletRequest and HttpServletResponsearguments, but instead receive and return many different types of data, such as domain objects, path parameters etc.

Also, note that you are not required to return a ModelAndView instance from a controller method. You may return a view name, or a ResponseEntity or a POJO that will be converted to a JSON response etc.

The RequestMappingHandlerAdapter makes sure the arguments of the method are resolved from the HttpServletRequest. Also, it creates the ModelAndView object from the method’s return value.

There is an important piece of code in the RequestMappingHandlerAdapter that makes sure all this conversion magic takes place:

ServletInvocableHandlerMethod invocableMethod

= createInvocableHandlerMethod(handlerMethod);

if (this.argumentResolvers != null) {

invocableMethod.setHandlerMethodArgumentResolvers(

this.argumentResolvers);

}

if (this.returnValueHandlers != null) {

invocableMethod.setHandlerMethodReturnValueHandlers(

this.returnValueHandlers);

}

The argumentResolvers object is a composite of different [HandlerMethodArgumentResolver](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/method/support/HandlerMethodArgumentResolver.html)instances.

There are over 30 different argument resolver implementations. They allow extracting of any kind of information from the request and providing it as method arguments. This includes URL path variables, request body parameters, request headers, cookies, session data etc.

The returnValueHandlers object is a composite of [HandlerMethodReturnValueHandler](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/method/support/HandlerMethodReturnValueHandler.html) objects. There are also a lot of different value handlers that can process the result of your method to create ModelAndViewobject expected by the adapter.

For instance, when you return a string from the hello() method, the ViewNameMethodReturnValueHandlerprocesses the value. But when you return a ready ModelAndView from the login() method, Spring uses the ModelAndViewMethodReturnValueHandler.

### Rendering the View

By now, Spring has processed the HTTP request and received a ModelAndView object, so it has to render the HTML page that the user will see in the browser. It does that based on the model and the selected view encapsulated in the ModelAndView object.

Also note that you could render a JSON object, or XML, or any other data format that can be transferred via HTTP protocol. We’ll touch more on that in the upcoming REST-focused section here.

Let’s get back to the DispatcherServlet. The render() method first sets the response locale using the provided [LocaleResolver](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/LocaleResolver.html) instance. Let’s assume that your modern browser sets the Accept header correctly and that the AcceptHeaderLocaleResolver is used by default.

During rendering, the ModelAndView object could already contain a reference to a selected view, or just a view name, or nothing at all if the controller was relying on a default view.

Since both hello() and login() methods specify the desired view as a String name, it has to be looked up by this name. So, this is where the viewResolvers list comes into play:

for (ViewResolver viewResolver : this.viewResolvers) {

View view = viewResolver.resolveViewName(viewName, locale);

if (view != null) {

return view;

}

}

This is a list of [ViewResolver](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/ViewResolver.html) instances, including our ThymeleafViewResolver provided by the thymeleaf-spring5 integration library. This resolver knows where to search for the views, and provides the corresponding view instances.

After calling the view’s render() method, Spring finally completes the request processing by sending the HTML page to the user’s browser:

## REST Support

Beyond the typical MVC scenario, we can also use the framework to create REST web services.

Simply put, you can accept a Resource as an input, specify a POJO as a method argument, and annotate it with @RequestBody. You can also annotate the method itself with @ResponseBody to specify that its result has to be transformed directly to an HTTP response:

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.ResponseBody;

@ResponseBody

@PostMapping("/message")

public MyOutputResource sendMessage(

@RequestBody MyInputResource inputResource) {

return new MyOutputResource("Received: "

+ inputResource.getRequestMessage());

}

This is also possible thanks to the extensibility of Spring MVC.

To marshall the internal DTOs to a REST representation, the framework makes use of the [HttpMessageConverter](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/http/converter/HttpMessageConverter.html) infrastructure. For example, one of the implementations is MappingJackson2HttpMessageConverter, which is able to convert model objects to and from JSON using the Jackson library.

And to further simplify the creation of a REST API, Spring introduces the @RestControllerannotation. This is handy to assume @ResponseBody semantics by default and avoid explicitly setting that on each REST controller:

import org.springframework.web.bind.annotation.RestController;

@RestController

public class RestfulWebServiceController {

@GetMapping("/message")

public MyOutputResource getMessage() {

return new MyOutputResource("Hello!");

}

}

## Conclusion

In this article, you’ve gone through the processing of a request in the Spring MVC framework in detail. You’ve seen how different extensions of the framework work together to provide all the magic and spare you the necessity of handling the tough parts of the HTTP protocol.

## Difference between @PathVariable and @RequestParam in Spring

Now that we understand both the difference as well as how to use both  
@RequestParam and  
@PathVariable in Spring MVC application, let’s revise the key difference between them from an interview perspective.

1) The @RequestParam is used to extract query parameters while @PathVariable is used to extract data right from the URI.

2) @RequestParam is more useful on a traditional web application where data is mostly passed in the query abatements while @PathVariable is more suitable for RESTful web services where URL contains values e.g. http://localhost:8080/book/9783827319333, here data, which is ISBN number is part of URI.

Diff b/w @Controller and @RestController

some important differences between these two annotations.

1. The @Controller is a common annotation which is used to mark a class as Spring MVC Controller while the @RestController is a special controller used in [RESTFul web services](http://javarevisited.blogspot.sg/2015/08/difference-between-soap-and-restfull-webservice-java.html) and the equivalent of @Controller + @ResponseBody.
2. The @RestController is relatively new, added only on Spring 4.0 but @Controller is an old annotation, exists since Spring started supporting annotation, and officially it was added on Spring 2.5 version.
3. The @Controller annotation indicates that the class is a “Controller” e.g. a web controller while the @RestController annotation indicates that the class is a controller where @RequestMapping methods assume @ResponseBody semantics by default i.e. servicing REST API.
4. The @Controller is a specialization of @Componentannotation while @RestController is a specialization of @Controller annotation. It is actually a convenience controller annotated with @Controller and @ResponseBodyas shown below.

|  |  |
| --- | --- |
| 1 | @Target(value=TYPE) |
| 2 | @Retention(value=RUNTIME) |

|  |  |
| --- | --- |
| 3 | @Documented |
| 4 | @Controller |

|  |  |
| --- | --- |
| 5 | @ResponseBody |
| 6 | public @interface RestController |

1. and here is how the declaration of @Controller looks like:

|  |  |
| --- | --- |
| 1 | @Target(value=TYPE) |
| 2 | @Retention(value=RUNTIME) |

|  |  |
| --- | --- |
| 3 | @Documented |
| 4 | @Component |

|  |  |
| --- | --- |
| 5 | public @interface Controller |

1. One of the key difference between @Controler and @RestCotroller in Spring MVC is that once you mark a class as @RestController then every method is written a domain object instead of a view. You can see Bryan Hassen’s [Introduction to Spring MVC 4](http://www.shareasale.com/m-pr.cfm?merchantID=53701&userID=880419&productID=557072989) to learn more about how to use the @RestController annotation in your Spring based application.
2. Another key difference between @RestController and @Controller is that you don’t need to use @ResponseBodyon every handler method once you annotate the class with @RestController as shown below:

**with @RestControler:**

|  |  |
| --- | --- |
| 1 | @RestController |
| 2 | public class Book{ |

|  |  |
| --- | --- |
| 3 |  |
| 4 | @RequestMapping(value={"/book"}) |

|  |  |
| --- | --- |
| 5 | public Book getBook(){ |
| 6 | //... |

|  |  |
| --- | --- |
| 7 | return book; |
| 8 | } |

|  |  |
| --- | --- |
| 9 | } |

**without @RestController:**

|  |  |
| --- | --- |
| 01 | @Controller |
| 02 | public class Book{ |

|  |  |
| --- | --- |
| 03 |  |
| 04 | @RequestMapping(value={"/book"}) |

|  |  |
| --- | --- |
| 05 | @ResponseBody |
| 06 | public Book getBook(){ |

|  |  |
| --- | --- |
| 07 | //... |
| 08 | return book; |

|  |  |
| --- | --- |
| 09 | } |
| 10 | } |

You can see that if you use Spring MVC @Controllerannotation to create a [RESTful response](http://javarevisited.blogspot.sg/2017/02/how-to-consume-json-from-restful-web-services-Spring-RESTTemplate-Example.html) you need to annotate each method with the @ResponseBody annotation, which is not required when you use @RestController. It not only makes your code more readable but also saves a couple of key strokes for you.

Here is a simple HelloWorld example using @RestControllerand **SpringBoot** framework:

That’s all about the **difference between**@Controller**and**@RestController**annotation in Spring MVC** and REST. @RestController is nothing but the shortcut to use both @Controller and @ResponseBody annotation together.

Spring purposefully added this annotation in Spring 4 to make the development of RESTful web services easier using Spring framework. It can directly convert the response to JSON or XML depending upon MIME type of request.

So, if you are creating a RESTful Web Services it’s better to use @RestController than combining the @Controller to @ResponseBody.

In this aicle, we will see how to configure a Spring MVC application without using a web.xml. We will use Java-based configuration.

For this example, we will use a simple Maven web project.

## Step 1: Create a Pom.xml for the Required Libraries

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>

 <artifactId>SpringWebExample</artifactId>

  <packaging>war</packaging>

  <version>0.0.1-SNAPSHOT</version>

 <name>SpringWebExample Maven Webapp</name>

  <url>http://maven.apache.org</url>

  <properties>

       <java-version>1.7</java-version>

  </properties>

    <dependencies>

       <dependency>

           <groupId>org.springframework</groupId>

           <artifactId>spring-webmvc</artifactId>

           <version>4.3.0.RELEASE</version>

       </dependency>

       <dependency>

           <groupId>javax.servlet</groupId>

           <artifactId>javax.servlet-api</artifactId>

           <version>3.0.1</version>

           <scope>provided</scope>

       </dependency>

       <dependency>

           <groupId>javax.servlet</groupId>

           <artifactId>jstl</artifactId>

           <version>1.2</version>

       </dependency>

       <dependency>

          <groupId>junit</groupId>

          <artifactId>junit</artifactId>

          <version>3.8.1</version>

          <scope>test</scope>

        </dependency>

    </dependencies>

    <build>

       <finalName>HelloWorld</finalName>

       <pluginManagement>

           <plugins>

               <plugin>

                   <groupId>org.apache.maven.plugins</groupId>

                   <artifactId>maven-compiler-plugin</artifactId>

                   <version>2.3.2</version>

                   <configuration>

                       <source>${java-version}</source>

                       <target>${java-version}</target>

                   </configuration>

               </plugin>

               <plugin>

                   <groupId>org.apache.maven.plugins</groupId>

                   <artifactId>maven-war-plugin</artifactId>

                   <version>2.4</version>

                   <configuration>

                       <warSourceDirectory>src/main/webapp</warSourceDirectory>

                       <warName>SpringWebExample</warName>

                       <failOnMissingWebXml>false</failOnMissingWebXml>

                   </configuration>

               </plugin>

           </plugins>

       </pluginManagement>

    </build>

</project>

We're using Spring 4.3.0 and Servlet 3.

## Step 2: Creating the SpringConfig Class

As we want to do Java-based configuration, we will create a class called SpringConfig, where we will register all Spring-related beans using Spring's Java-based configuration style.

This class will replace the need to create a SpringApplicationContext.xml file, where we use two important tags

<context:component-scan/>

<mvc:annotation-driven/>

Please note that this class has to extend the

org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter class.

Let's see the class.

**SpringConfig.java**:

package com.example.anotatedconfiguration;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.ViewResolver;

import org.springframework.web.servlet.config.annotation.DefaultServletHandlerConfigurer;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

import org.springframework.web.servlet.view.JstlView;

@EnableWebMvc

@ComponentScan(basePackages = "com.example")

public class SpringConfig extends WebMvcConfigurerAdapter{

   @Bean

    public ViewResolver viewResolver() {@Configuration

       InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

       viewResolver.setViewClass(JstlView.class);

       viewResolver.setPrefix("/WEB-INF/pages/");

       viewResolver.setSuffix(".jsp");

       return viewResolver;

    }

    @Override

    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

       configurer.enable();

    }

}

Please note that here, we will use three different annotations at the top level. They will serve the purpose of the XML-based tags used earlier.

|  |  |  |
| --- | --- | --- |
| XML Tag | Annotation | Description |
| <context:component-scan/> | @ComponentScan() | Scan starts from base package and registers all controllers, repositories, service, beans, etc. |
| <mvc:annotation-driven/> | @EnableWebMvc | Enable Spring MVC-specific annotations like @Controller |
| Spring config file | @Configuration | Treat as the configuration file for Spring MVC-enabled applications. |

Also, we use the @Bean tag to register ViewResolver. We use InternalResourceViewResolver.

## Step 3: Replacing Web.xml

Create another class, which will replace our traditional web.xml. We use Servlet 3.0 and extend the org.springframework.web.WebApplicationInitializer class.

**WebServletConfiguration.java**:

package com.example.anotatedconfiguration;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;

import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

import org.springframework.web.servlet.DispatcherServlet;

public class WebServletConfiguration implements WebApplicationInitializer{

   public void onStartup(ServletContext ctx) throws ServletException {

       AnnotationConfigWebApplicationContext webCtx = new AnnotationConfigWebApplicationContext();

       webCtx.register(SpringConfig.class);

       webCtx.setServletContext(ctx);

       ServletRegistration.Dynamic servlet = ctx.addServlet("dispatcher", new DispatcherServlet(webCtx));

       servlet.setLoadOnStartup(1);

       servlet.addMapping("/");

    }

}

Here we provide our SpringConfig class and add DispatcherServlet, which acts as the FrontController of the Spring MVC application.

SpringConfig class is the source of Spring beans, before which we used contextConfiglocation.

## Step 4: Create a Controller Class

Now we will create a Controller class, Which will take a parameter from request URL and greet a message in the browser.

package com.example.anotatedconfiguration;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.ui.ModelMap;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class GreetController {

@RequestMapping(path= "/greet/{name}",method=RequestMethod.GET)

public String greet(@PathVariable String name, ModelMap model){

    String greet =" Hello !!!" + name + " How are You?";

    model.addAttribute("greet", greet);

    System.out.println(greet);

return "greet";

}

}

## Step 5: Create a JSP Page to Show the Message

So, with all the configuration

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

<html>

<head>

<%@ page isELIgnored="false" %>

</head>

<h1>Welcome to Spring 4 and Servlet 3 Based Application</h1>

<body>

<div>

</div>

</body>

</html>

Please note that here that I use <%@ page isELIgnored="false" %>. If the JSTL version is old (before 1.2), it can’t  evaluate ${} EL. If your JSTL version is 1.2, you can safely remove that tag.

**Output**:

http://localhost:8080/SpringWebExample/greet/shamik

Hello !!!shamik How are You?