

# **Spring MVC Reference Doc**

## **Table of Content-**

- 1. Setting up Java**
- 2. Installing a web Server(TomCat)**
- 3. Configuring Tomcat with Eclipse**
- 4. Creating First Spring Mvc Project**
- 5. Exploring Spring Mvc Components**
- 6. An Library Management Spring MVC Project Overview**

# 1. Setting up Java

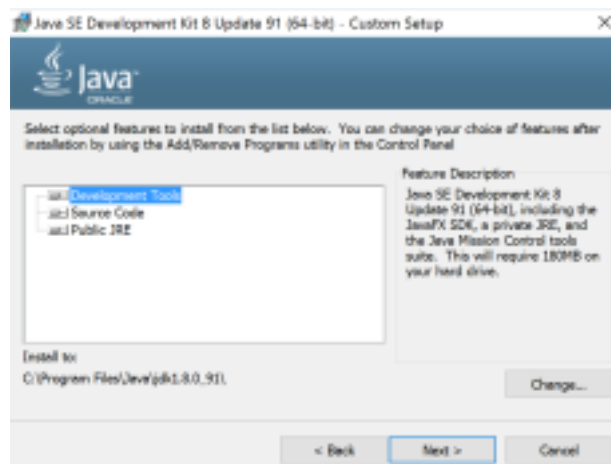
Obviously, the first thing that we need to do is to install Java. The more technical name for Java is Java Development Kit (JDK). JDK includes a Java compiler (javac), a Java virtual machine, and a variety of other tools to compile and run Java programs.

We are going to use Java 8, which is the latest and greatest version of Java, but Java 6 or any higher version is also sufficient to complete this chapter, but I strongly recommend you use Java 8 since in later chapters of this book we may use some of the Java 8 features such as, streams and lambda expressions. Let's take a look at how we can install JDK on a Windows operating system:

1. Go to the Java SE download page on the Oracle website at <https://www.oracle.com/java/technologies/javase/javase8u211-later-archive-downloads.html>

2. Now, click on the listed download link that corresponds to your Windows operating system architecture; for instance, if your operating system is of type 32 bit, click on the download link that corresponds to Windows x86. If your operating system is of type 64 bit, click on the download link that corresponds to Windows x64.

4. Now it will start downloading the installer. Once the download is finished, go to the downloaded directory and double-click on the installer. This will open up a wizard window. Just click through the next buttons in the wizard, leaving the default options alone, and click on the Close button at the end of the wizard:



Additionally, a separate wizard also prompts you to install Java Runtime Environment (JRE). Go through that wizard as well to install JRE in your system.

5. Now you can see the installed JDK directory in the default location; in our case, the default location is C:\Program Files\Java\jdk1.8.0\_60.

After installing JDK, we still need to perform some more configurations to use Java conveniently from any directory on our computer. By setting up the environment variables for Java in the Windows operating system, we can make the Java compiler and tools accessible from anywhere in the file system:

1. Navigate to Start Menu | Settings | System | About | System info | Advanced system settings.

2. A System Properties window will appear; in this window, select the Advanced tab and click on the Environment Variables button to open the Environment Variables window.

3. Now, click on the New button in the System variables panel and enter JAVA\_HOME as the variable name and enter the installed JDK directory path as the variable value; in our case, this would be C:\Program Files\Java\jdk1.8.0\_91. If you do not have proper rights for the operating system, you will not be able to edit System variables; in that case, you can create the JAVA\_HOME variable under the User variables panel.

4. Now, in the same System variables panel, double-click on the path variable entry; an Edit System Variable window will appear.

5. Edit Variable value of Path by clicking the new button and enter the following text %JAVA\_HOME%\bin as the value.

If you are using a Windows operating system prior to version 10, edit the path variable carefully; you should only append the text at the end of an existing path value. Don't delete or disturb the existing values; make sure you haven't missed the ; (semi-colon) delimited mark as the first letter in the text that you append:

5. Now click on the OK button.

Now we have installed JDK in our computer. To verify whether our installation has been carried out correctly, open a new command window, type `java -version`, and press Enter; you will see the installed version of Java compiler on the screen:

```
C:\Users\Amuthan>java -version
```

```
java version "1.8.0_91"
```

```
Java(TM) SE Runtime Environment (build 1.8.0_91-b15)
```

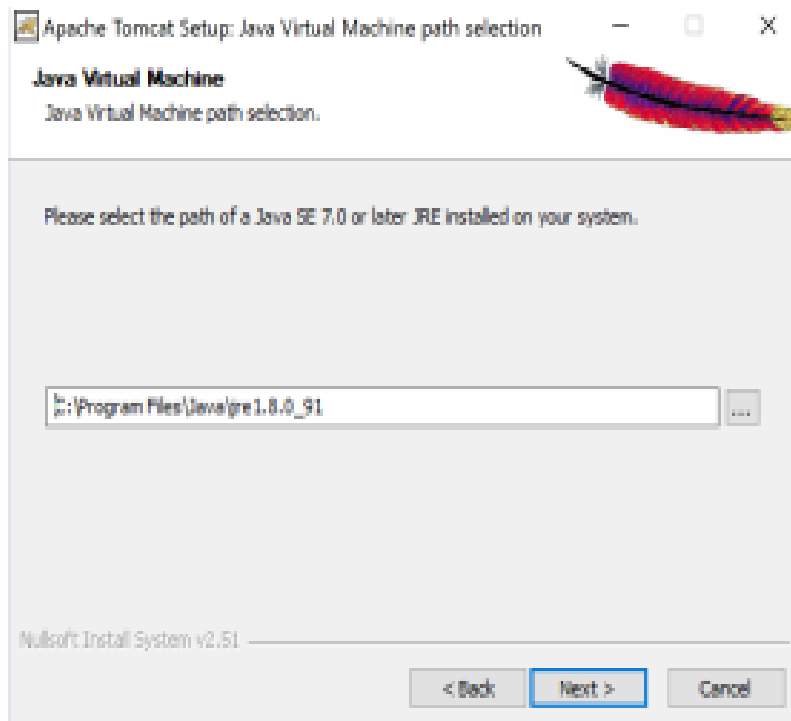
```
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b15, mixed mode)
```

## 2. Installing a web Server(TomCat)

Apache Tomcat is a popular Java web server and servlet container. We are going to use Apache Tomcat Version 8.0, which is the latest, but you can even use version 7.0. Let's take a look at how we can install the Tomcat web server:

1. Go to the Apache Tomcat home page at <https://tomcat.apache.org/download-90.cgi>.
2. Click on the **32-bit/64-bit Windows Service Installer (pgp,sha512)** link; it will start downloading the installer.
3. Once the download is finished, go to the downloaded directory and double-click on the installer; this will open up a wizard window.
4. Just click through the **Next** buttons in the wizard, leaving the default options alone, and click on the **Finish** button at the end of the wizard. Note that before clicking on the **Finish** button, just ensure that you have unchecked **Run Apache Tomcat** checkbox.

Installing Apache Tomcat with the default option works successfully only if you have installed Java in the default location. Otherwise, you have to correctly provide the JRE path according to the location of your Java installation during the installation of Tomcat, as shown in the following screenshot:



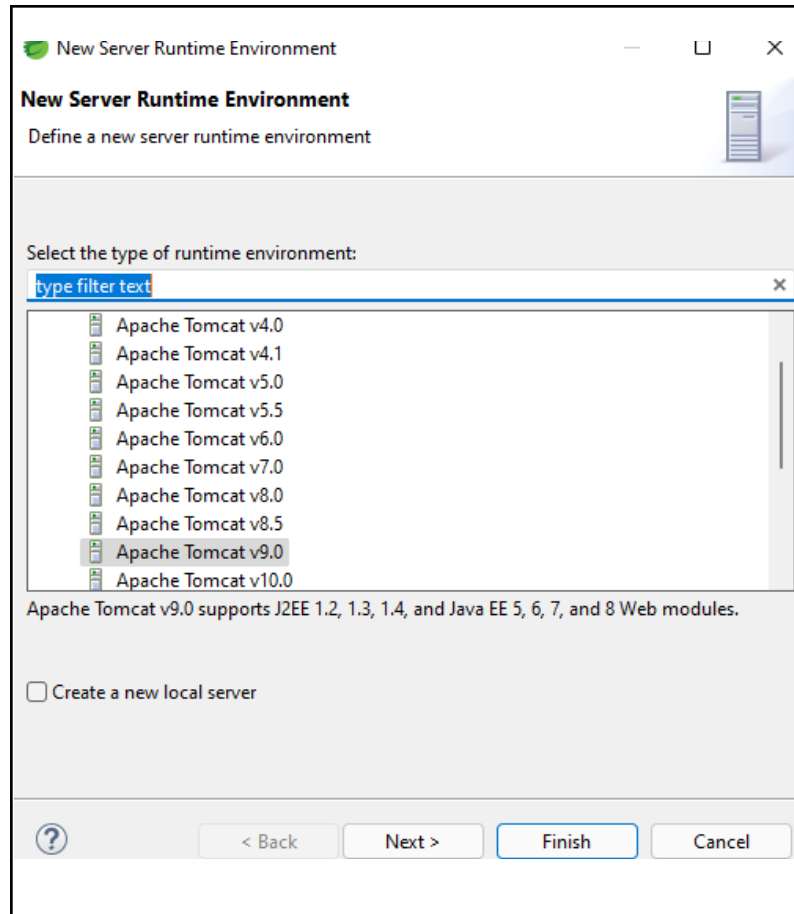
### 3. Configuring Tomcat with Eclipse

We can use the Tomcat web server to deploy our application, but we have to inform Eclipse about the location of the Tomcat container so that we can easily deploy our project in to Tomcat. Let's configure Tomcat on Eclipse:

1. Open Eclipse from the desktop icon, if it is not already open.
2. Go to the menu bar and navigate to **Window | Preferences | Server | Runtime Environments**.
3. You can see the available servers listed on the right panel. Now click on the **Add** button to add our Tomcat web server.

You may also see **Pivotal tc Server Developer Edition (Runtime) v3.1** listed under the available servers, which comes along with the Eclipse installation. Although Eclipse might come with an internal Pivotal tc Server, we chose to use the Tomcat web server as our server runtime environment because of its popularity.

4. A wizard window will appear; type tomcat in the **Select the type of runtime environment:** text box, and a list of available Tomcat versions will be shown. Just select **Tomcat v9.0** and select the **Create a new local server** checkbox. Finally, click on the **Next** button, as shown in the following screenshot:




Selecting the server type during the Tomcat configuration on Eclipse

5. In the next window, click on the **Browse** button and locate Tomcat's installed directory, and then click on the **OK** button. You can find Tomcat's installed directory under C:\Program Files\Apache Software Foundation\Tomcat 9.0 if you have installed Tomcat in the default location. Then, click on the **Finish** button, as shown in the following screenshot:

New Server Runtime Environment

Tomcat Server

Specify the installation directory



Name:

Apache Tomcat v9.0 (2)

Tomcat installation directory:

C:\Program Files\Apache Software Foundation\Tomcat 9.0

Browse...


apache-tomcat-9.0.46

Download and Install...

JRE:

Workbench default JRE

Installed JREs...



< Back

Next >

Finish

Cancel

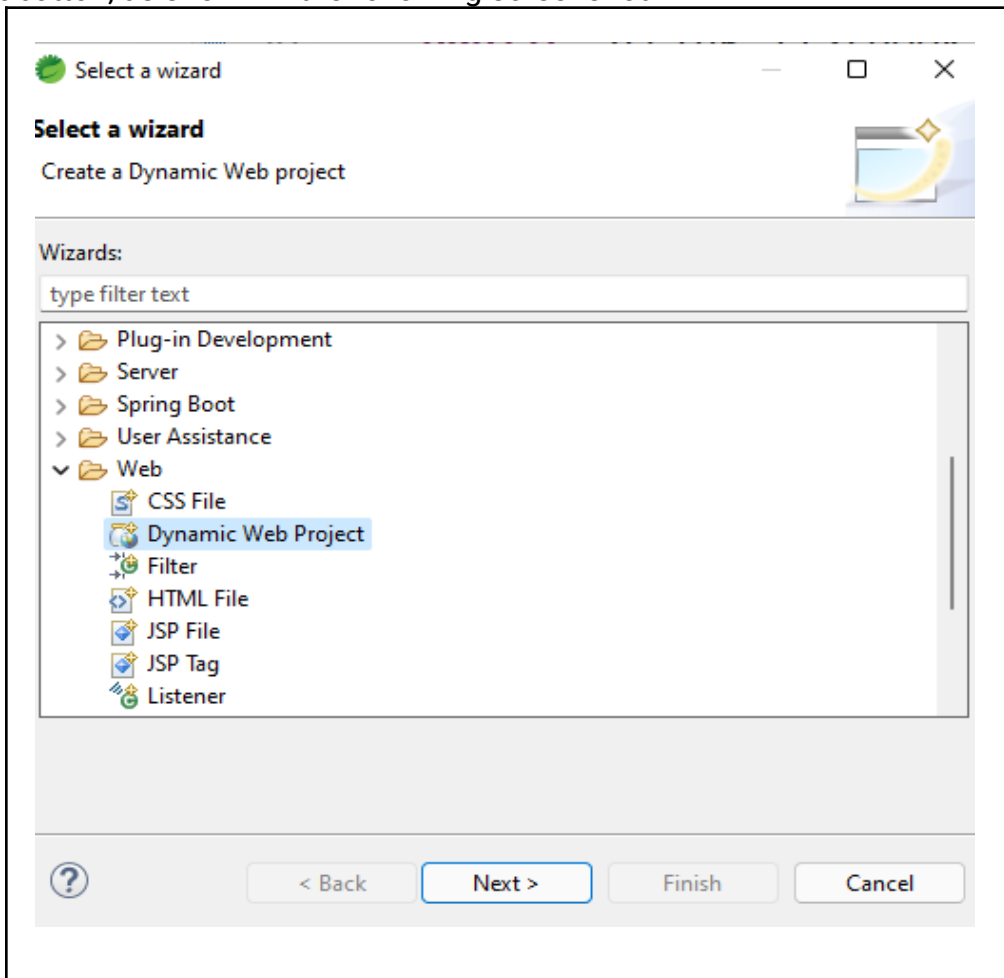


## 4. Creating First Spring MVC Project

So far, we have seen how we can install all the prerequisite tools and software. Now we are going to develop our first Spring MVC application using Eclipse. Eclipse provides an easy-to-use project template. Using these templates, we can quickly create our project directory structures without much problem.

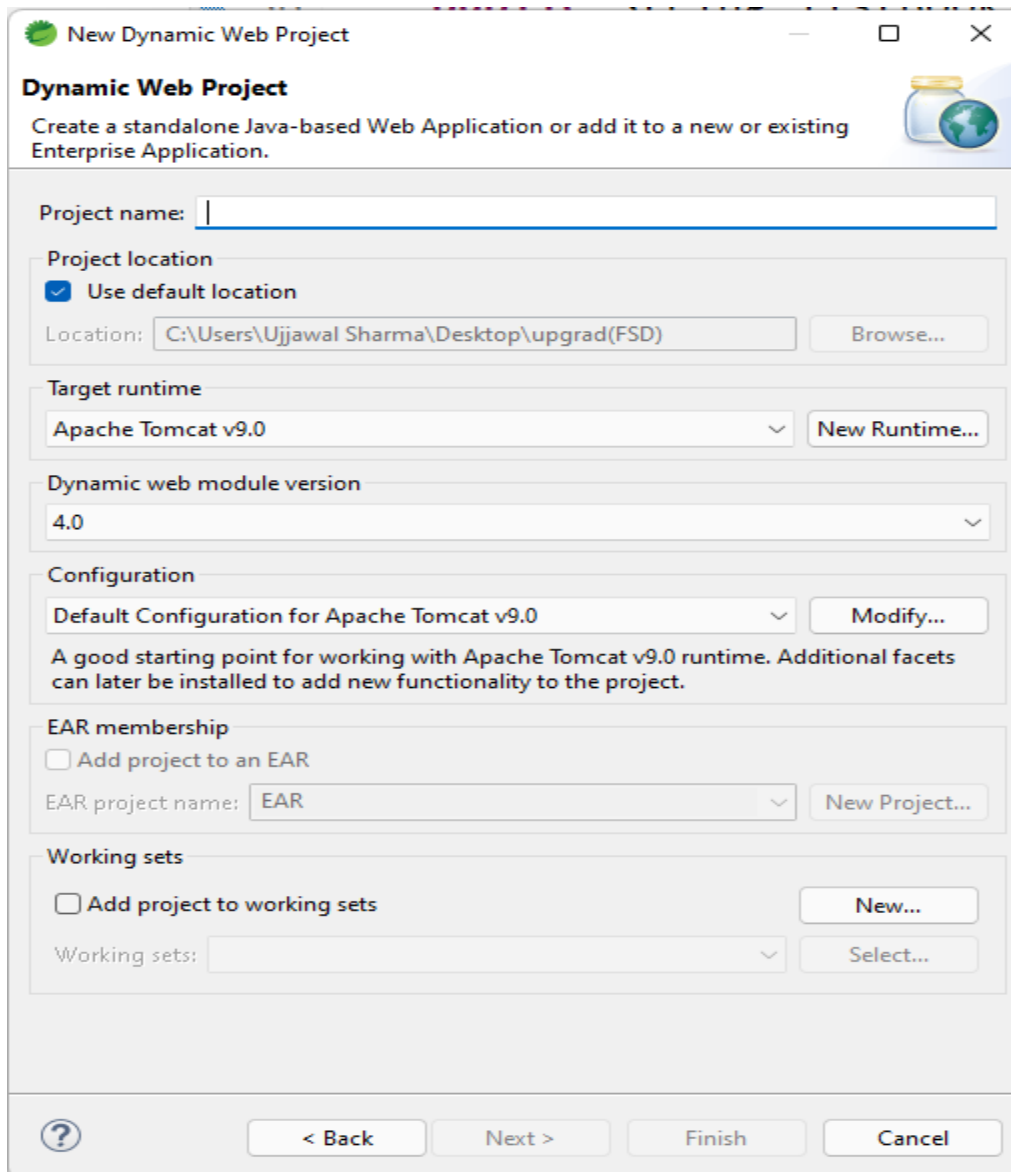
Let's create our first Spring MVC project in Eclipse:

1. In Eclipse, navigate to **File | New | Project**; a **New Project** wizard window will appear.
2. Select **Dynamic Web Project** from the list under Web and click on the **Next** button, as shown in the following screenshot:



3. Now, a **Dynamic Web Project** dialog window will appear; Just enter the project name you want to give to your project and select the target runtime to the installed tomcat server that we have previously configured with eclipse.

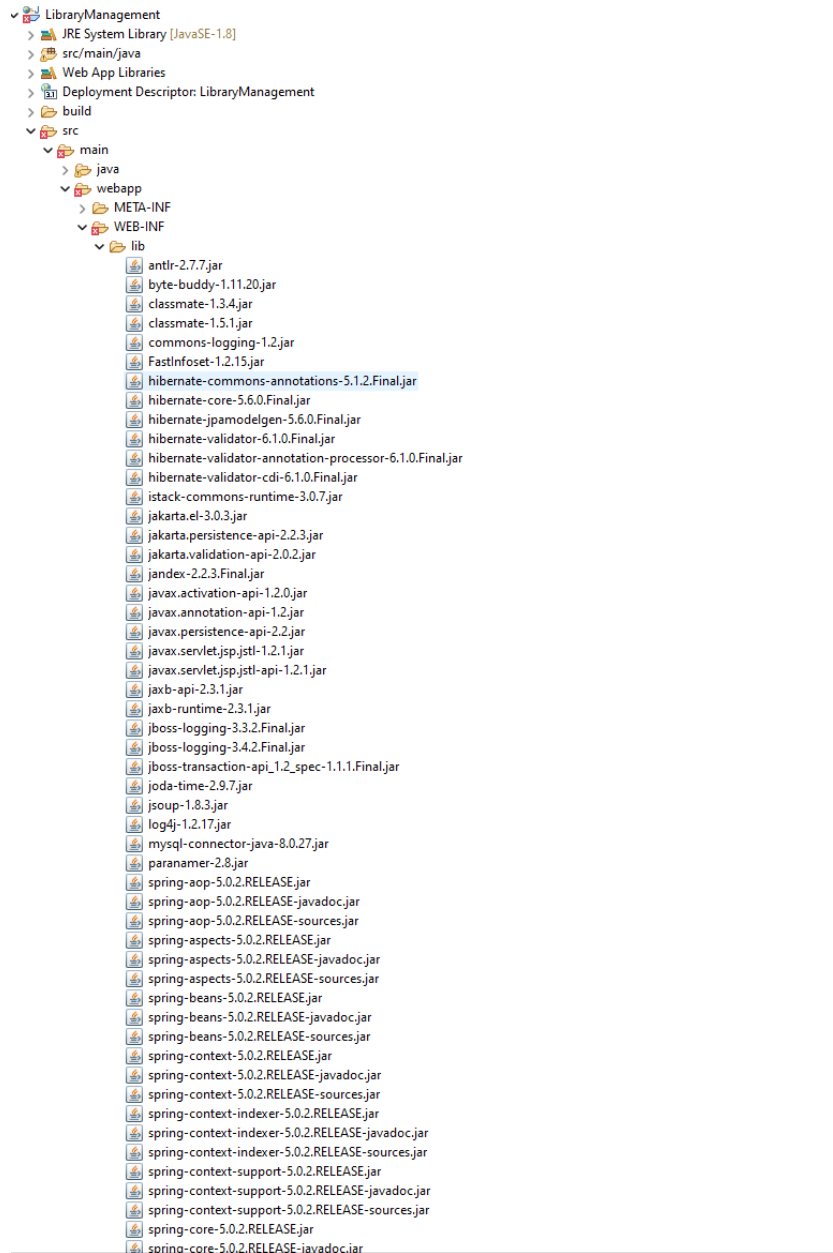
4. Then click on finish ,it will create a new project in your project explorer.



The screenshot shows the 'New Dynamic Web Project' dialog box in Eclipse. The title bar says 'New Dynamic Web Project'. The main heading is 'Dynamic Web Project' with a subtitle 'Create a standalone Java-based Web Application or add it to a new or existing Enterprise Application.' and a small globe icon. The dialog is divided into several sections: 'Project name:' with a text input field; 'Project location' with a checked 'Use default location' checkbox and a 'Location:' field showing 'C:\Users\Ujjawal Sharma\Desktop\upgrad(FSD)' and a 'Browse...' button; 'Target runtime' with a dropdown menu showing 'Apache Tomcat v9.0' and a 'New Runtime...' button; 'Dynamic web module version' with a dropdown menu showing '4.0'; 'Configuration' with a dropdown menu showing 'Default Configuration for Apache Tomcat v9.0' and a 'Modify...' button, followed by a descriptive text; 'EAR membership' with an unchecked 'Add project to an EAR' checkbox, an 'EAR project name:' field showing 'EAR', and a 'New Project...' button; and 'Working sets' with an unchecked 'Add project to working sets' checkbox, a 'Working sets:' field, a 'New...' button, and a 'Select...' button. At the bottom, there is a help icon, and buttons for '< Back', 'Next >', 'Finish', and 'Cancel'.

Congratulations,you have successfully created a spring mvc project, in the next steps we will be adding the jars under lib folder at location src/main/webapp/WEB-INF/lib.

5. Now to add dependencies we will be copying all required jars (dependencies) at src/main/webapp/WEB-INF/lib in our newly created dynamic web project.



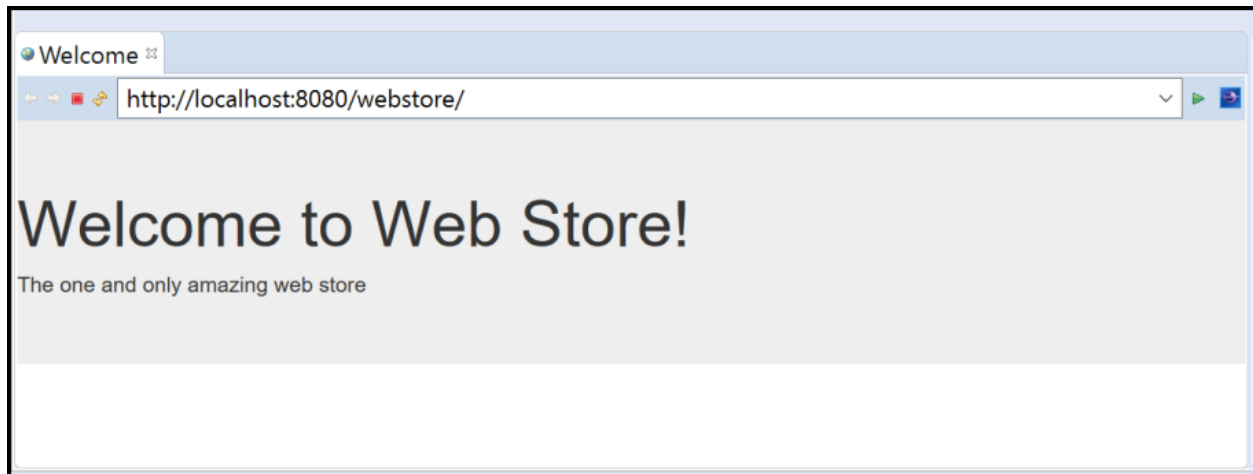
As shown in the above screenshot we have Dynamic web project named Library Management, and we have copied all required jars at src/main/webapp/WEB-INF/lib under our project.

## 5. Exploring and Running our First Spring MVC Project

We have created our project and added all the required jars, so we are ready to code.

As a first step, let's create a home page in our project to welcome our customers.

Our aim is simple; when we enter the URL `http://localhost:8080/webstore/` on the browser, we would like to show a welcome page that is similar to the following screenshot:



To create a welcome page, we need to execute the following steps:

1. Create a `webapp/WEB-INF/jsp/` folder structure under the `src/main/` folder; create a JSP file called `welcome.jsp` under the `src/main/webapp/WEB-INF/jsp/` folder, and add the following code snippets into it and save it:

```

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

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible"
content="IE=edge">
<meta name="viewport"
content="width=device-width, initial-scale=1">
<title>Welcome</title>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.5/css/bootstrap.min.css">
</head>
<body>
<div class="jumbotron">
<h1> ${greeting} </h1>
<p> ${tagline} </p>
</div>
</body>
</html>

```

2. Create a class called HomeController under the com.packt.webstore.controller package in the source directory src/main/java and add the following code into it:

```

package com.packt.webstore.controller;

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

@Controller
public class HomeController {

    @RequestMapping("/")
    public String welcome(Model model) {
        model.addAttribute("greeting", "Welcome to
Web Store!");
        model.addAttribute("tagline", "The one and
only amazing web store");
    }
}

```

```
        return "welcome";  
    } }
```

Lets understand the first two steps here-

In step 1, we just created a JSP file; the important thing we need to notice here is the `<h1>` tag and the `<p>` tag. Both the tags have some expression that is surrounded by curly braces and prefixed by the `$` symbol:

```
<h1> ${greeting} </h1>  
<p> ${tagline} </p>
```

So, what is the meaning of `${greeting}`? It means that `greeting` is a kind of variable; during rendering of this JSP page, the value stored in the `greeting` variable will be shown in the header 1 style, and similarly, the value stored in the `tagline` variable will be shown as a paragraph.

So now, the next question is where we will assign values to those variables arises. This is where the controller will be of help; within the `welcome` method of the `HomeController` class, take a look at the following lines of code:

```
model.addAttribute("greeting", "Welcome to Web Store!");  
model.addAttribute("tagline", "The one and only amazing web store");
```

You can observe that the two variable names, `greeting` and `tagline`, are passed as a first parameter of the `addAttribute` method and the corresponding second parameter is the value for each variable. So what we are doing here is simply putting two strings, "Welcome to Web Store!" and "The one and only amazing web store", into the model with their corresponding keys as `greeting` and `tagline`. As of now, simply consider the fact that model is a kind of map data structure.

Folks with knowledge of servlet programming can consider the fact that `model.addAttribute` works exactly like `request.setAttribute`.

So, whatever value we put into the model can be retrieved from the view (JSP) using the corresponding key with the help of the `${}` placeholder expression. In our case, `greeting` and `tagline` are keys.

## Role Of Dispatcher Servlet in SpringMVC program-

We put values into the model, and we created the view that can read those values from the model. So, the Controller acts as an intermediate between the View and the Model; with this, we have finished all the coding part required to present the welcome page. So will we be able to run our project now? No. At this stage, if we run our project and enter the URL `http://localhost:8080/webstore/` on the browser, we will get an **HTTP Status 404** error. This is because we have not performed any servlet mapping yet.

So what is servlet mapping? Servlet mapping is a configuration of mapping a servlet to a URL or URL pattern. For example, if we map a pattern like `/status/*` to a servlet, all the HTTP request URLs starting with a text status such as `http://example.com/status/synopsis` or `http://example.com/status/complete?date=today` will be mapped to that particular servlet. In other words, all the HTTP requests that carry the URL pattern `/status/*` will be handed over to the corresponding mapped servlet class.

In a Spring MVC project, we must configure a servlet mapping to direct all the HTTP requests to a single front servlet. The front servlet mapping is a design pattern where all requests for a particular web application are directed to the same servlet. This pattern is sometimes called as **Front Controller Pattern**. By adapting the Front Controller design, we make front servlet have total control over the incoming HTTP request so that it can dispatch the HTTP request to the desired controller.

One such front servlet given by Spring MVC framework is the Dispatcher servlet (`org.springframework.web.servlet.DispatcherServlet`). We have not configured a Dispatcher servlet for our project yet; this is why we get the **HTTP Status 404** error if we run our project now.

## Configuring The Dispatcher Servlet using web.xml file-

The Dispatcher servlet is what examines the incoming HTTP request and invokes the right corresponding controller method. In our case, the welcome method from the HomeController class needs to be invoked if we make an HTTP request by entering the `http://localhost:8080/webstore/` URL on the browser. So let's configure the Dispatcher servlet for our project:

1. Create a web.xml file under location src/main/webapp/WEB-INF having content given below-

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    id="WebApp_ID" version="3.1">

    <display-name>SpringMVCDemo</display-name>

    <absolute-ordering />

    <!-- Spring MVC Configurations -->

    <!-- Configure Spring MVC Dispatcher Servlet -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.Dispatcher
Servlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</par
am-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Set up URL mapping for Spring MVC Dispatcher Servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

</web-app>
```



2. We are providing a reference to another xml file inside the code of our web.xml file

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

These parameters are for the configuration of our Dispatcher servlet in our application and those configuration are located in spring-mvc-demo-servlet.xml file under src/main/webapp/WEB-INF location.

3. So create a file named spring-mvc-demo-servlet.xml file under src/main/webapp/WEB-INF location having code given below-

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

  <!-- Add support for component scanning -->
  <context:component-scan
    base-package="com.greatlearning.library" />

  <!-- Define Spring MVC view resolver -->
  <bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    id="jspViewResolver">
    <property name="viewClass"
      value="org.springframework.web.servlet.view.JstlView"></property>
```

```

        <property name="prefix" value="/WEB-INF/views/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
    <bean
class="org.springframework.jdbc.datasource.DriverManagerDataSource"
id="dataSource">
        <property name="driverClassName"
            value="com.mysql.jdbc.Driver"></property>
        <property name="url"
value="jdbc:mysql://localhost:3306/hibernate_one-to-one_uni"></property>
        <property name="username" value="root"></property>
        <property name="password" value="070898"></property>
    </bean>
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"></property>
        <property name="annotatedClasses">
            <list>
                <value>com.greatlearning.library.Book</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</prop>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
            </props>
        </property>
    </bean>
    <bean id="myTransactionManager"

class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>

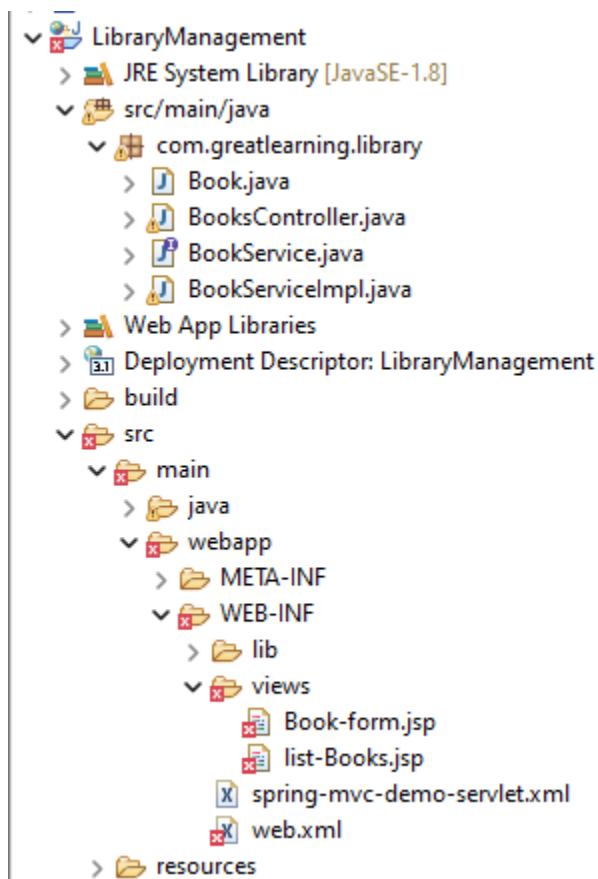
```

## 6. Library Management

We have created a Library Management project using Spring MVC along with Hibernate.

We will see how the flow is established between the different components in our project.

Lets see our project structure and the code in different files in our project-



And the code for different files is given below-

## 1. Book.java-

```
package com.greatlearning.library;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="book")
public class Book {

    // define fields

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="name")
    private String name;

    @Column(name="category")
    private String category;

    @Column(name="author")
    private String author;
```

```
// define constructors
```

```
public Book()  
{
```

```
}
```

```
public Book(String name, String category, String author) {  
    super();
```

```
    this.name = name;
```

```
    this.category = category;
```

```
    this.author = author;
```

```
}
```

```
// define getter/setter
```

```
public int getId() {  
    return id;  
}
```

```
public void setId(int id) {  
    this.id = id;  
}
```

```
public String getName() {  
    return name;  
}
```

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

```
public String getCategory() {
```

```
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    // define toString
    @Override
    public String toString() {
        return "Book [id=" + id + ", name=" + name + ",
category=" + category + ", author=" + author + "]";
    }
}
```

## 2. BooksController.java-

```
package com.greatlearning.library;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```
@Controller
```

```
@RequestMapping("/books")
```

```
public class BooksController {
```

```
    @Autowired
```

```
    private BookService bookService;
```

```
    // add mapping for "/list"
```

```
    @RequestMapping("/list")
```

```
    public String listBooks(Model theModel) {
```

```
        // get Books from db
```

```
        List<Book> theBooks = bookService.findAll();
```

```
        // add to the spring model
```

```
        theModel.addAttribute("Books", theBooks);
```

```
        return "list-Books";
```

```
    }
```

```
    @RequestMapping("/showFormForAdd")
```

```
    public String showFormForAdd(Model theModel) {
```

```
        // create model attribute to bind form data
```

```
        Book theBook = new Book();
```

```

        theModel.addAttribute("Book", theBook);
        return "Book-form";
    }

    @RequestMapping("/showFormForUpdate")
    public String showFormForUpdate(@RequestParam("bookId") int
theId,Model theModel) {
        // get the Book from the service
        Book theBook = bookService.findById(theId);
        // set Book as a model attribute to pre-populate the form
        theModel.addAttribute("Book", theBook);

        // send over to our form
        return "Book-form";
    }

    @PostMapping("/save")
    public String saveBook(@RequestParam("id") int
id,@RequestParam("name") String name,@RequestParam("category")
String category,@RequestParam("author") String author) {
        System.out.println(id);
        Book theBook;
        if(id!=0)
        {
            theBook=bookService.findById(id);
            theBook.setName(name);
            theBook.setCategory(category);
            theBook.setAuthor(author);
        }
        else
            theBook=new Book(name, category, author);
        // save the Book
        bookService.save(theBook);
        // use a redirect to prevent duplicate submissions
        return "redirect:/books/list";
    }
}

```



```

@RequestMapping("/delete")
public String delete(@RequestParam("bookId") int theId) {
    // delete the Book
    bookService.deleteById(theId);
    // redirect to /Books/list
    return "redirect:/books/list";
}

```

```

@RequestMapping("/search")
public String search(@RequestParam("name") String name,
                    @RequestParam("author") String author,
                    Model theModel) {

    // check names, if both are empty then just give list of all
Books

    if (name.trim().isEmpty() && author.trim().isEmpty()) {
        return "redirect:/books/list";
    }
    else {
        // else, search by first name and last name
        List<Book> theBooks =
            bookService.searchBy(name, author);

        // add to the spring model
        theModel.addAttribute("Books", theBooks);

        // send to list-Books
        return "list-Books";
    }
}
}

```

3. BookService.java and BookServiceImpl.java contains the code of hibernate to fetch the data from the database.

4. Book-form.jsp and list-Books.jsp are the jsp files which have embedded html code which shows some static and dynamic data that we got from the controllers.

5. Web.xml and spring-mvc-demo-servlet.xml are files which map dispatcher servlet to the application url "/" and also initialize the required beans in the application context.

### **Data and request flow in the application-**

1. Say if ,User makes a request to endpoint say  
["http://localhost:8080/LibarayManagement/books/list"](http://localhost:8080/LibarayManagement/books/list)
2. Then our tomcat will check web.xml and find the mapped servlet which is dispatcher servlet in this case and the configuration for dispatcher servlet is provided in spring-mvc-demo-servlet.xml where it will initialize all the defined beans like for hibernate session factory, jsp view resolver and many more which are required by the application.
3. Now the dispatcher servlet will check for all the classes annotated with @Controller annotation because we have mapping for url endpoints in those classes.
4. In our case BooksController contains the request mapping for "/books/list" which is mapped to the listBooks method of the BooksController class.
5. So the method listBooks will fetch the books records from the database using hibernate code which is written in BookServiceImpl class.
6. Hibernate uses Entity Pojo java class to map that to the database table. So we have created a class called Book inside our Book.java file which is the Pojo for database book tables.
7. After fetching data from the database our controller will bind that data to a model and return it to the list-Books.jsp view.
8. list-Books.jsp file uses that data which was sent from the controller to list those records on the web page using html.
9. That's how we see the book's record on the web page.