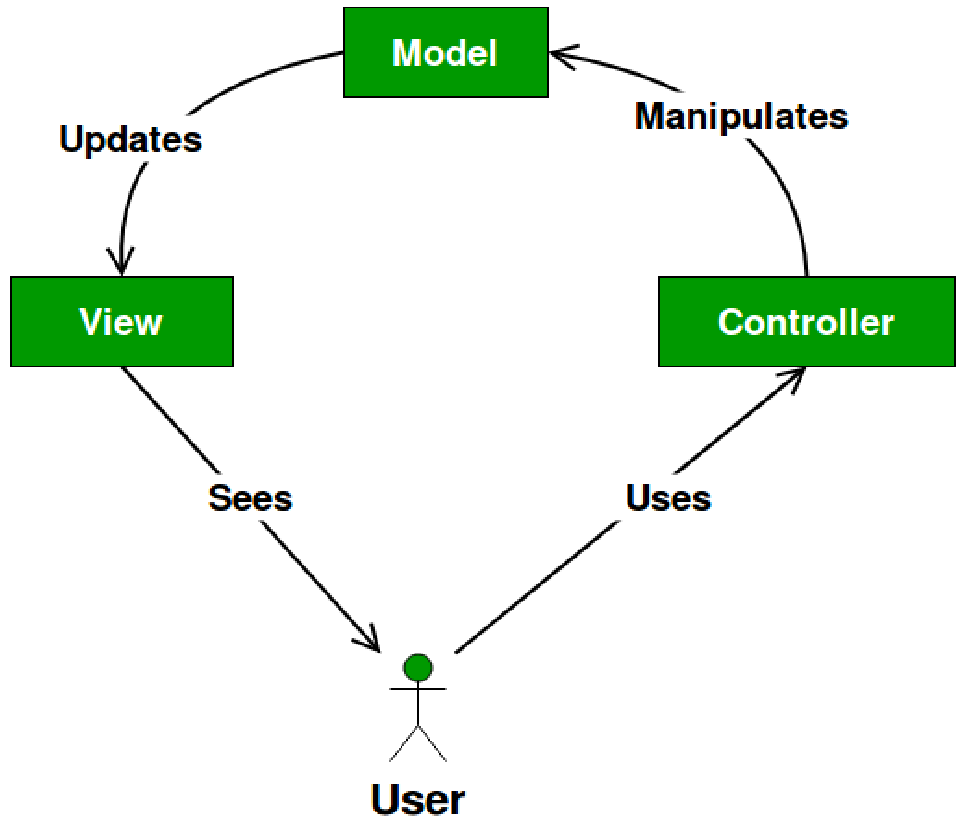
MVC Design Pattern

What is MVC Design Pattern

- The **Model View Controller** (MVC) design pattern specifies that an application consist of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.
- MVC is more of an architectural pattern, but not for complete application. MVC mostly relates to the UI / interaction layer of an application. You’re still going to need business logic layer, maybe some service layer and data access layer.

MVC UML Diagram



Model

- The **Model** contains only the pure application data, it contains no logic describing how to present the data to a user.

View

- The **View** presents the model’s data to the user. The view knows how to access the model’s data, but it does not know what this data means or what the user can do to manipulate it.

Controller

- The **Controller** exists between the view and the model. It listens to events triggered by the view (or another external source) and executes the appropriate reaction to these events.
- In most cases, the reaction is to call a method on the model. Since the view and the model are connected through a notification mechanism, the result of this action is then automatically reflected in the view.

Lab – Classic MVC in Plain Java

Create a Student.java class with “rollNo” and “name” properties.

This will act as our **Model**.

Lab – Classic MVC in Plain Java

- Student.java

```
public class Student {
    private String rollNo;
    private String name;

    // getters and setters
}
```

Lab – Classic MVC in Plain Java

Create a StudentView.java class.

This will act as our View.

Lab – Classic MVC in Plain Java

- StudentView.java

```
class StudentView
{
    public void printStudentDetails(String studentName, String studentRollNo)
    {
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("Roll No: " + studentRollNo);
    }
}
```

Lab – Classic MVC in Plain Java

Create a StudentController.java class.

This will act as our **Controller**.

Lab – Classic MVC in Plain Java

- StudentController.java

```java
class StudentController
{
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view)
    {
        this.model = model;
        this.view = view;
    }
```

Lab – Classic MVC in Plain Java

- StudentController.java (continued…)

```
public void setStudentName(String name)
	{
		model.setName(name);
	}

	public String getStudentName()
	{
		return model.getName();
	}
```

Lab – Classic MVC in Plain Java

- StudentController.java (continued…)

```
•     public void setStudentRollNo(String rollNo)
•     {
•         model.setRollNo(rollNo);
•     }
•
      public String getStudentRollNo()
•     {
•         return model.getRollNo();
•     }
```

Lab – Classic MVC in Plain Java

- StudentController.java (continued…)

```
•       public void updateView()
•       {
•           view.printStudentDetails(model.getName(), model
  .getRollNo());
•       }
• }
```

Lab – Classic MVC in Plain Java

Create a MVCPattern.java class.

This will act as main class.

Lab – Classic MVC in Plain Java

- MVCPattern.java

```java
class MVCPattern
{
    public static void main(String[] args)
    {
        Student model = retriveStudentFromDatabase();

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName(“Jon Snow");

        controller.updateView();
    }
```

Lab – Classic MVC in Plain Java

- MVCPattern.java (continued…)

```
• private static Student retriveStudentFromDatabase()
•     {
•         Student student = new Student();
•         student.setName(“Foo Bar");
•         student.setRollNo("15UCS157");
•         return student;
•     }
•
• }
```

Lab – Classic MVC in Plain Java

- Run the code.
- Observe how view (console) get updated.

spring

MVC

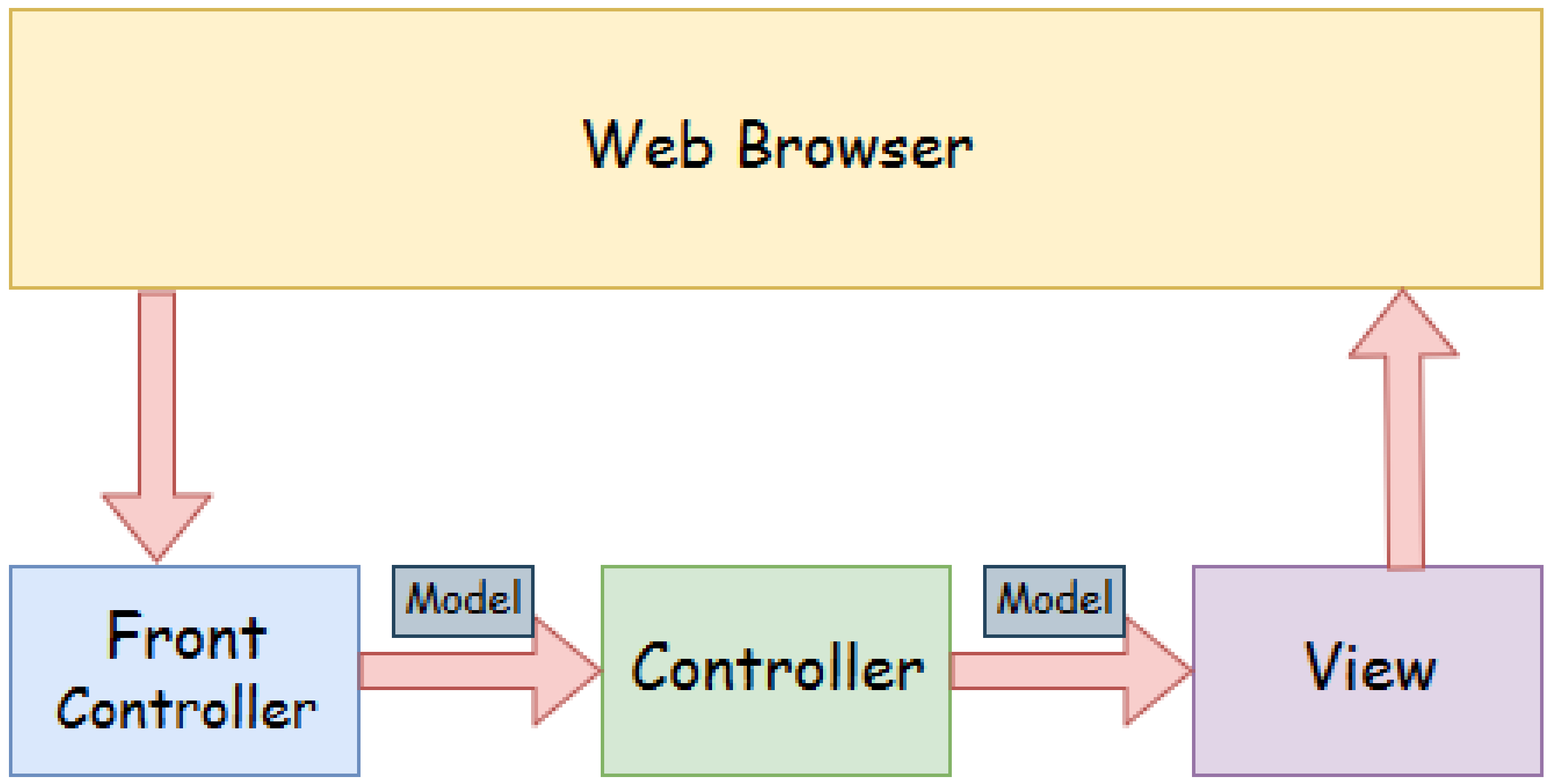
Spring MVC

- A Spring MVC is a Java framework which is used to build web applications.
- It follows the Model-View-Controller design pattern.
- It implements all the basic features of a core spring framework like **Inversion of Control**, **Dependency Injection**.

Spring MVC

- A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**.
- Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

Spring MVC Flow Diagram



Spring MVC Flow Components

Model - A model contains the data of the application. A data can be a single object or a collection of objects.

Controller - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.

Spring MVC Flow Components

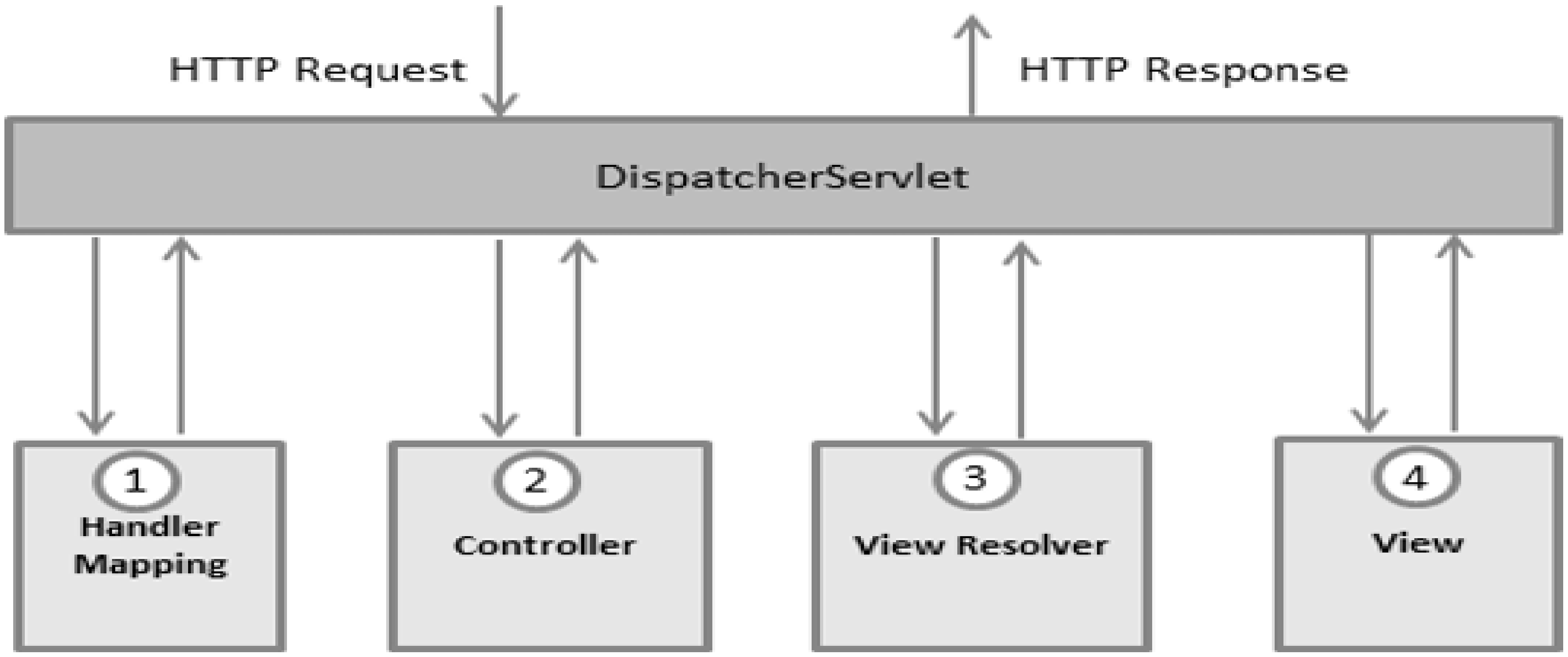
View - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

Front Controller - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

The DispatcherServlet

- The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses.

The request processing workflow of the Spring Web MVC *DispatcherServlet*



Sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.

Sequence of events corresponding to an incoming HTTP request to *DispatcherServlet*

- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

Advantages of Spring MVC Framework

- **Separate roles** - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.
- **Light-weight** - It uses light-weight servlet container to develop and deploy your application.

Advantages of Spring MVC Framework

- **Powerful Configuration** - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.
- **Rapid development** - The Spring MVC facilitates fast and parallel development.

Advantages of Spring MVC Framework

- **Reusable business code** - Instead of creating new objects, it allows us to use the existing business objects.
- **Easy to test** - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.
- **Flexible Mapping** - It provides the specific annotations that easily redirect the page.

MySql Setup

- Go to https://dev.mysql.com/downloads/installer/
- Download the mysql installer.
- Double click the downloaded “msi” file.
- Keep the defaults and follow the instructions on the window.
- Enter “root” password on “Accounts and Roles” window. Remember the password you have entered.
- On “Apply Configuration” screen click “Execute”.
- Finally click Finish.

Lab2 - Write MySql Query to create table and enter data.

1

Open Workbench.

2

Connect to MySql Server.

3

Create a new *schema/db* with name “spring_mvc”.

4

Write script to create a *product* table with following columns :

- *Id – primary key*
- *Name – string*
- *Brand – string*
- *Madein – string*
- *Price - float*

Lab2 - Solution

- CREATE TABLE `product` (
- `id` int(11) NOT NULL AUTO_INCREMENT,
- `name` varchar(45) NOT NULL,
- `brand` varchar(45) NOT NULL,
- `madein` varchar(45) NOT NULL,
- `price` float NOT NULL,
- PRIMARY KEY (`id`)
-) ENGINE=InnoDB DEFAULT CHARSET=utf8;

Lab2 - Solution

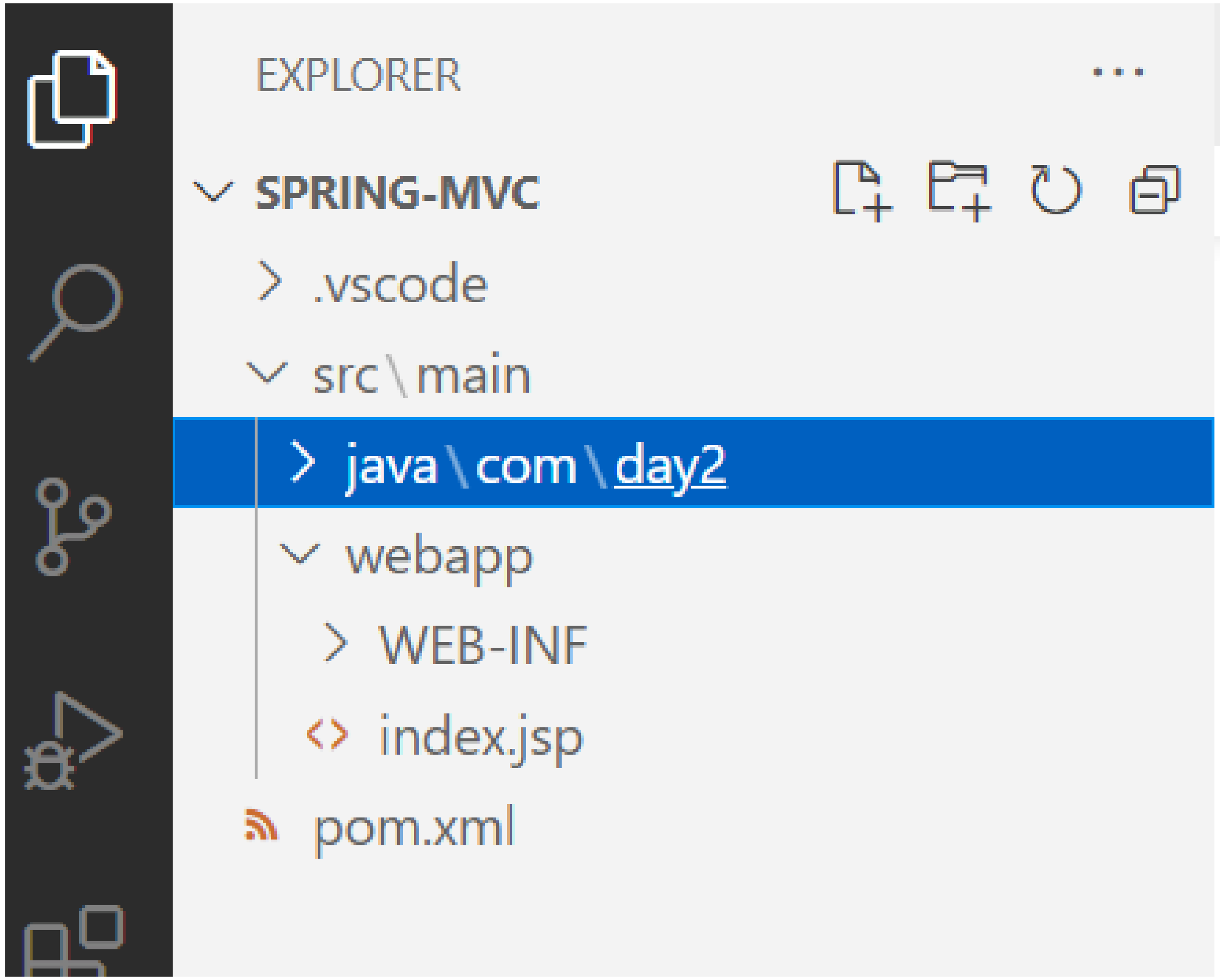
- `INSERT INTO product(name, brand, madein, price) VALUES("laptop", "Mac", "USA", 400)`

Lab3 – Spring MVC Hello World

- Create a web project using ‘maven-archetype-webapp’.
- Add ‘java’ folder under ‘src/main’.
- Create package under ‘java’ and name it same as ‘groupid’.

Lab3 – Spring MVC Hello World

- Now you project structure should look something like this.
- In my case ‘groupid’ id ‘com.day2’ and ‘artifactid’ is ‘spring-mvc’.



Lab3 – Delete index.jsp and web.xml

Remove index.jsp file under the webapp.

Remove web.xml from WEB-INF folder.

Lab3 – Update pom.xml

- Add spring-webmvc dependency under ‘dependencies’ tag.

```
<dependency>
        <groupId>org.springframework</groupId>
          <artifactId>spring-webmvc</artifactId>
          <version>5.2.7.RELEASE</version>
 </dependency>
```

Lab3 – Update pom.xml

- Add servlet and jsp dependency under ‘dependencies’ tag.

```
•       <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>
```

Lab3 – Update pom.xml

- Add jstl dependency under ‘dependencies’ tag.
- The JavaServer Pages Standard Tag Library (**JSTL**) is a collection of useful JSP tags which encapsulates the core functionality common to many JSP applications. **JSTL** has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags.

```
•       <dependency>
•         <groupId>javax.servlet</groupId>
•         <artifactId>jstl</artifactId>
•         <version>1.2</version>
•       </dependency>
```

Lab3 – Create *Configuration* class

- Create a new package ‘configuration’ under the base package (groupid) and create a new class ‘WebConfig.java’ inside it.
- Annotate this class with ‘@Configuration’ and ‘@EnableWebMvc’ annotations.

```
• import org.springframework.context.annotation.Configuration;
• import org.springframework.web.servlet.config.annotation.EnableWebMvc;

• @Configuration
• @EnableWebMvc
• public class WebConfig {


• }
```

Lab3 – Define '*ViewResolver*'

- This defines where spring should look for our views.

```
• @Configuration
• @EnableWebMvc
• public class WebConfig {
•     @Bean
•     public ViewResolver viewResolver() {
•         InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
•         viewResolver.setViewClass(JstlView.class);
•         viewResolver.setPrefix("/WEB-INF/views/");
•         viewResolver.setSuffix(".jsp");
•         return viewResolver;
•     }
• }
```

Lab3 – Define *Config Initializer class*

To bootstrap an application that loads this configuration, we also need an initializer class.

Create ‘MainWebAppInitializer.java’ under ‘configuration’ package.

Implement the ‘WebApplicationInitializer’ interface.

Override the ‘onStartup’ method.

Lab3 – Define *Config Initializer class*

```java
public class MainWebAppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(final ServletContext sc) throws ServletException {
        AnnotationConfigWebApplicationContext ctx = new AnnotationConfigWebApplicationContext();
        ctx.register(WebConfig.class);
        ctx.setServletContext(container);

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

        servlet.setLoadOnStartup(1);
        servlet.addMapping("/");
    }
}
```

Lab3 – Define Controller

- Create a new package ‘controllers’ under base package (groupid).
- Add “HelloWorldController.java” class to it.
- Annotate the class with ‘@Controller’.

```
@Controller
public class HelloWorldController {

}
```

Lab3 – Define Controller

- Add a method inside controller and annotate it with ‘@RequestMapping’.
- Return the name of the view you want render.

```
@Controller
public class HelloWorldController {
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String sayHello(Model model) {
        model.addAttribute("greeting", "Hello World from Spring 5 MVC");
        return "welcome";
    }
}
```

Lab3 – Update WebConfig

- Update WebConfig.java so that it know where our controllers live.
- To do that add @ComponentScan annotation to the class.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.day2")
public class WebConfig {
```

Lab3 – Add views

Create a folder ‘views’ under ‘WEB-INF’.

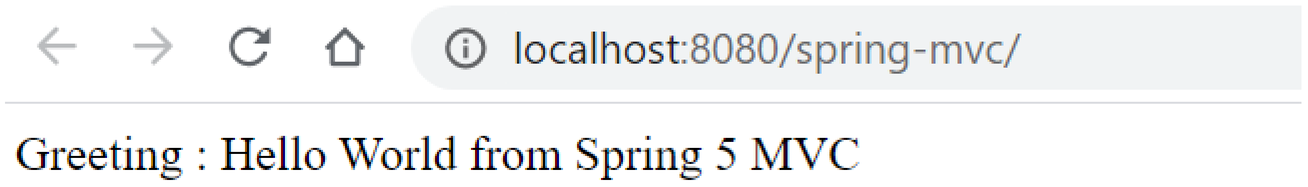
Create ‘welcome.jsp’ inside ‘views’ folder.

Lab3 – 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>HelloWorld page</title>
• </head>
• <body>
•     Greeting : ${greeting}
• </body>
• </html>
```

Lab3 – Run the App.

- Create a *war* by running *mvn clean package.*
- Right click the *war* and select “Run on Tomcat server”.
- Open the app in browser. You should see welcome message.



```
localhost:8080/spring-mvc/
Greeting : Hello World from Spring 5 MVC
```

Understanding the Lab3 key concepts

- @Configuration ??
- @Bean ??
- @EnableWebMVC ??
- @ComponentScan ??
- ViewResolver ??
- WebApplicationInitializer ??
- @Controller ??
- @RequestMapping

@Configuration & @Bean

Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.

The **@Bean** annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context.

@EnableWebMVC

Adding this annotation to a @Configuration class imports the Spring MVC configuration from which is the main class providing the configuration behind the MVC Java config.

If you don't use this annotation you might not initially notice any difference but things like content-type and accept header, generally content negotiation won't work.

@ComponentScan

With Spring, **we use the *@ComponentScan* annotation along with *@Configuration* annotation to specify the packages that we want to be scanned.**

@ComponentScan without arguments tells Spring to scan the current package and all of its sub-packages.

ViewResolver

All MVC frameworks provide a way of working with views.

Spring does that via the view resolvers, which enable you to render models in the browser without tying the implementation to a specific view technology.

The *ViewResolver* maps view names to actual views.

And the Spring framework comes with quite a few view resolvers
e.g. *InternalResourceViewResolver*, *XmlViewResolver*, *ResourceBundleViewResolver* and a few others.

WebApplicationInitializer

WebApplicationInitializer is used for booting Spring web applications.

WebApplicationInitializer registers a Spring DispatcherServlet and creates a Spring web application context.

WebApplicationInitializer

Mostly, developers use AbstractAnnotationConfigDispatcherServletInitializer, which is an implementation of the WebApplicationInitializer, to create Spring web applications.

Traditionally, Java web applications based on Servlets were using file to configure a Java web application. Since Servlet 3.0, web applications can be created programatically via Servlet context listeners.

@Controller

The @Controller annotation is a specialization of the generic stereotype @Component annotation, which allows a class to be recognized as a Spring-managed component.

The @Controller annotation extends the use-case of @Component and marks the annotated class as a business or presentation layer.

When a request is made, this will inform the DispatcherServlet to include the controller class in scanning for methods mapped by the @RequestMapping annotation.

@RequestMapping

This annotation maps HTTP requests to handler methods of MVC and REST controllers.

In Spring MVC applications, the RequestDispatcher (Front Controller Below) servlet is responsible for routing incoming HTTP requests to handler methods of controllers.

When configuring Spring MVC, you need to specify the mappings between the requests and handler methods. To configure the mapping of web requests, you use the @RequestMapping annotation.

@RequestMapping

The @RequestMapping annotation can be applied to class-level and/or method-level in a controller.

The class-level annotation maps a specific request path or pattern onto a controller.

You can then apply additional method-level annotations to make mappings more specific to handler methods.

Lab4 – Enhancing our HelloWorld App

Create a Form in *welcome.jsp*.

Form should have input field which will accept a username.

Form should have a *Add User* button which sends POST request.

On click of *Add User* button, list of users should appear on screen.

New screen also has button to *Add More Users* which will allow users to add more users.

Lab4 – Updated Controller

```
@Controller
public class HelloWorldController {
    List<String> users = new ArrayList<>();
    @RequestMapping(value = "/", method = RequestMethod.GET)
    public String sayHello(Model model) {
        model.addAttribute("greeting", "Hello World from Spring 5 MVC");
        return "welcome";
    }

    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public String printName(Model model, @ModelAttribute("username") String username) {
        users.add(username);
        model.addAttribute("users", users);
        return "users";
    }
}
```

Lab4 – Updated 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>HelloWorld page</title>
</head>
<body>
    Greeting : ${greeting}
    <form action="./users" method="POST">
        UserName: <input type="text" name="username">
        <br />
        <input type="submit" value="Add User" />
    </form>
</body>
</html>
```

Lab4 – New users.jsp

- Add this to top of jsp file.
  - <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
- This helps you write jstl in jsp.

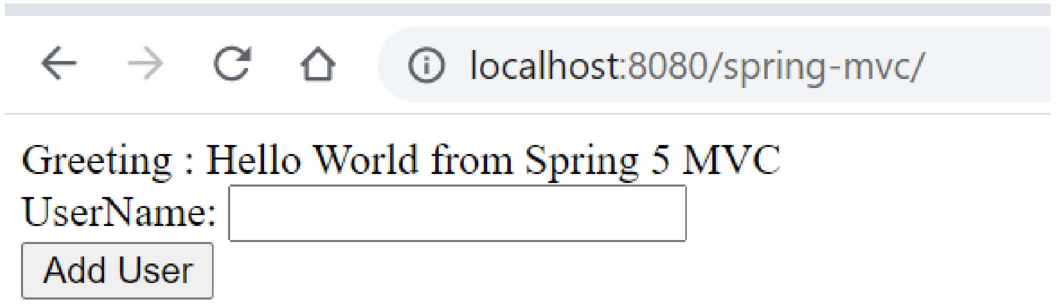
```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
    <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
    <html>

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

    <body>

    </body>

    </html>
```

Lab4 – Update users.jsp

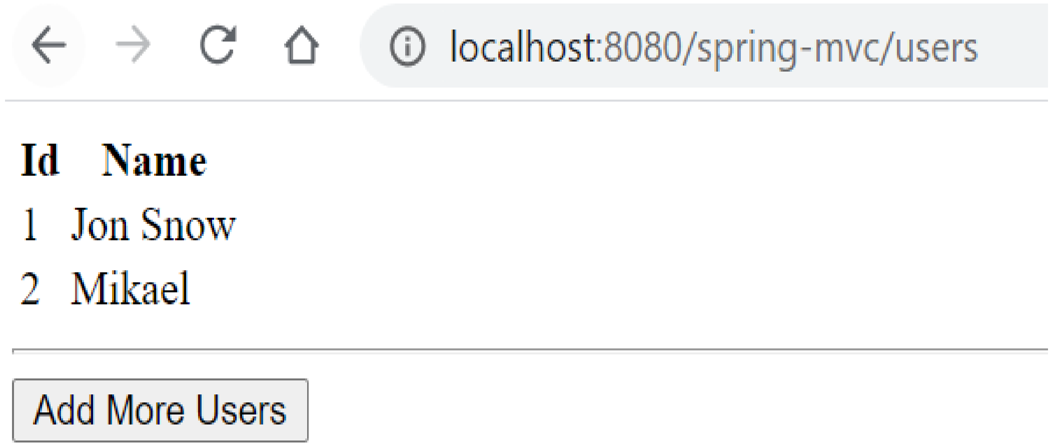
- Put this inside <body> tag.

```
<table>
            <tr>
                <th>Id</th>
                <th>Name</th>
            </tr>
            <c:forEach items="${users}" var="user" varStatus="loopCounter">
                <tr>
                    <td>
                        <c:out value="${loopCounter.index + 1}" />
                    </td>
                    <td>
                        <c:out value="${user}" />
                    </td>
                </tr>
            </c:forEach>
        </table>
        <hr>
        <form action="./" method="GET">
            <input type="submit" value="Add More Users" />
        </form>
```

Lab4 – Package and Run



Greeting : Hello World from Spring 5 MVC
UserName:
Add User



Id	Name
1	Jon Snow
2	Mikael

Add More Users

All our users keep disappearing when we restart the server

Let's Save our users in Database

Update pom.xml

- Add these dependencies in pom.xml

```xml
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.25</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-jdbc</artifactId>
      <version>5.1.1.RELEASE</version>
    </dependency>
```

Update WebConfig.java

- Create these two beans in WebConfig.java

```
•       @Bean
•       public DataSource getDataSource() {
•           DriverManagerDataSource dataSource = new DriverManagerDataSource();
•           dataSource.setDriverClassName("com.mysql.jdbc.Driver");
•           dataSource.setUrl("jdbc:mysql://localhost:3306/test");
•           dataSource.setUsername(“root");
•           dataSource.setPassword(“root");
•           return dataSource;
•       }
•
        @Bean
•       public JdbcTemplate jdbcTemplate() {
•           JdbcTemplate template = new JdbcTemplate();
•           template.setDataSource(getDataSource());
•           return template;
•       }
```

That’s It !! Redeploy the App to connect to database.

Lab5 – Enhance HelloWorld App to save users in Database

Hints :-

Refer Lab2 for queries.

Use JdbcTemplate bean added in WebConfig.