UTM
UNIVERSITI TEKNOLOGI MALAYSIA

SECJ 3303 – INTERNET PROGRAMMING

TOPIC 7 – SPRING MVC

innovative • entrepreneurial • global

UTM JOHOR BAHRU

OBJECTIVES

Applied

- Test your web application using Apache Maven build tool.
- Write a code with Spring MVC framework.

Knowledge

- Describe the Apache Maven build tool.
- Describe the overview of Spring framework.
- Describe the Spring Web MVC in Spring framework.
- Understand the important components of Spring Web MVC.
- Explain the Spring Web MVC flow and know how to implement in a code.
- Understand the Spring Exception Handling

Apache Maven

- Maven is a build automation and dependency management tool that helps in project management.
- **Build tool** is essential for the process of building. It is needed for the following process:
  - Generating source code
  - Generating documentation from the source code
  - Compiling of source code
  - Packaging the compiled code into JAR files.
  - Installing the package code in local repository, server or central repository.

Apache Maven

- Maven is written is Java and C# and is based on the **Project Object Model** (pom.xml).
- Project Object Model is an **XML file** that has all the information regarding project and configuration details.
- When we tend to execute a task, Maven searches for the POM in the current directory.
- The tool is used to build and manage any **Java-based project**.
- It helps in downloading dependencies, which refer to the libraries or JAR files.

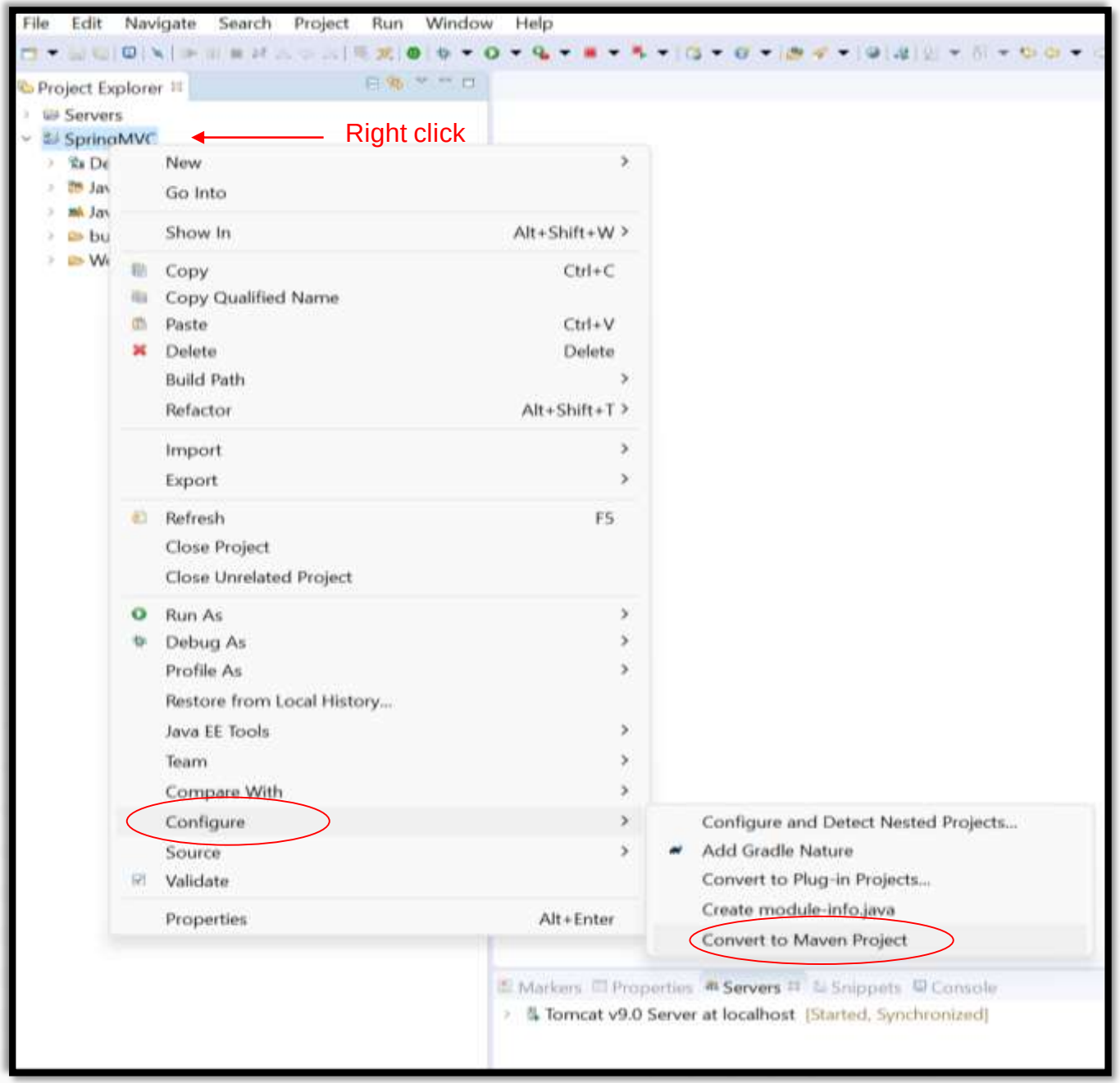
The Problem That Maven Solved

- Getting right JAR files for each project as there maybe different versions of separate packages.
- To download dependencies, visiting the official website of different software is not needed. We can refer to mvnrepository.com
- Helps to create the right project structure which is essential for execution.
- Building and deploying the project to make it works.

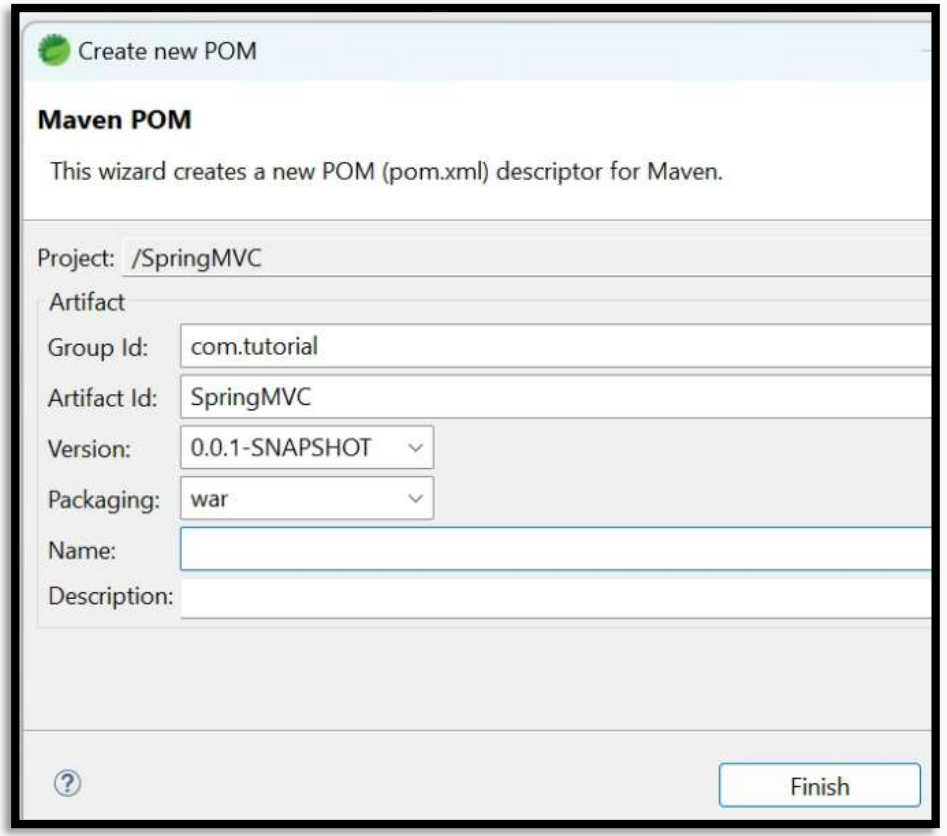
How to convert to Maven Project in STS?

Steps:

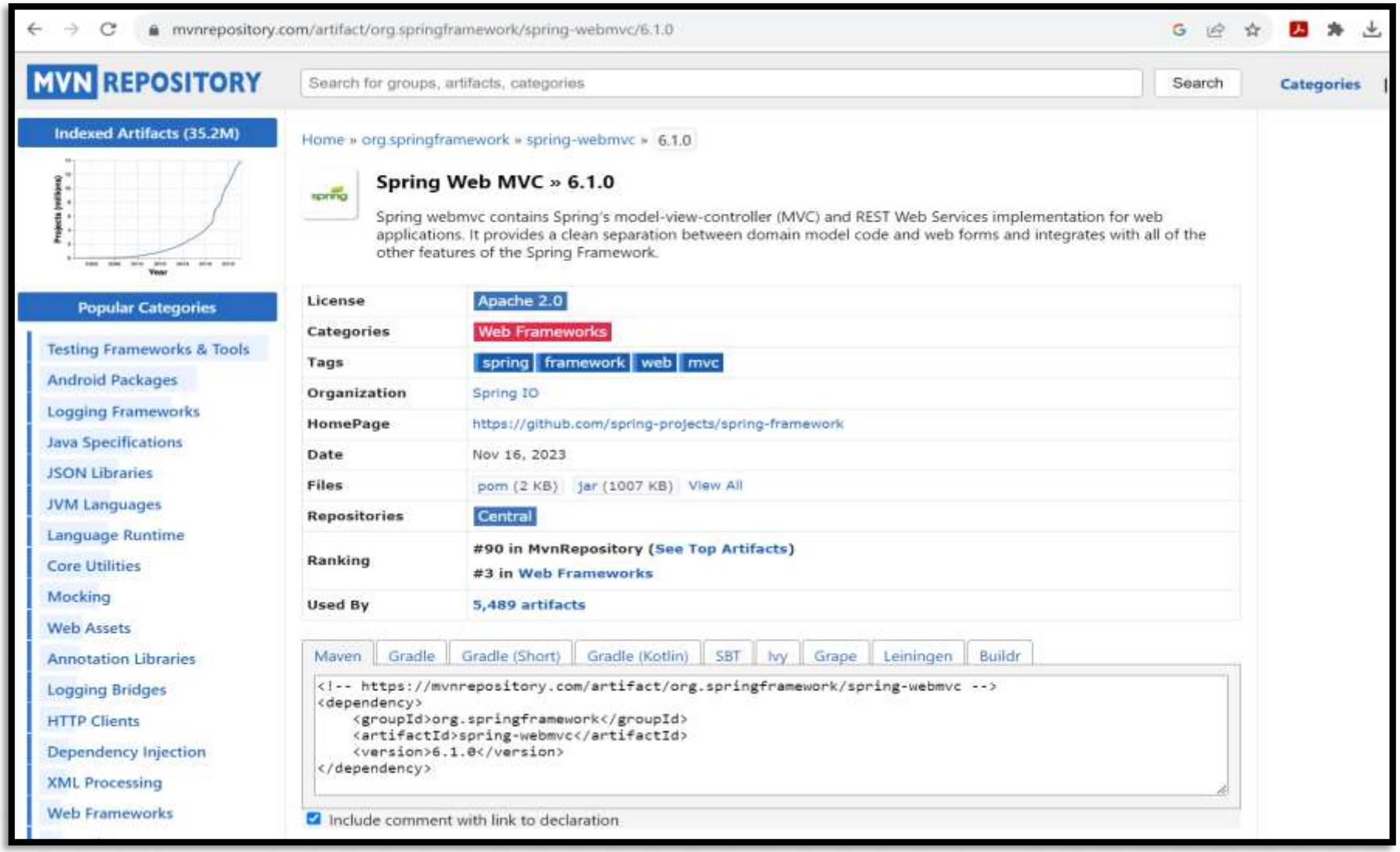
- Right click on your dynamic web project
- Select - Configure
- Select - Convert to Maven project



Group id : It uniquely identifies your project among all. Example: com.companyname
Artifact id : It is name of jar or war without version. It may be something like project.
Version : Version is used for version control for artifact id. If you distribute this project, you may incrementally create different version of it.



Platform – mvnrepository.com



Example of dependencies in pom.xml

If use previous version: use Spring version 5

If use latest version: use Spring version 6

If use latest version: jakarta.servlet

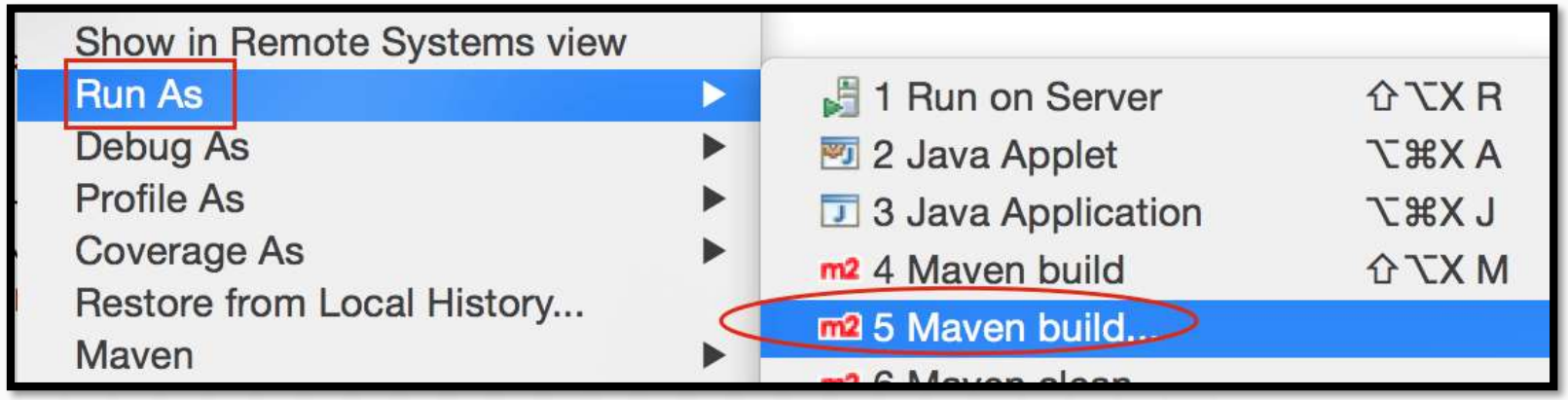
```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.3.18</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.3.20</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.20</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.springframework/spring-web -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>5.3.20</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>4.0.1</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

If use previous version: javax.servlet

```
<!--
https://mvnrepository.com/artifact/jakarta.servlet/
jakarta.servlet-api -->
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>6.0.0</version>
    <scope>provided</scope>
</dependency>
```

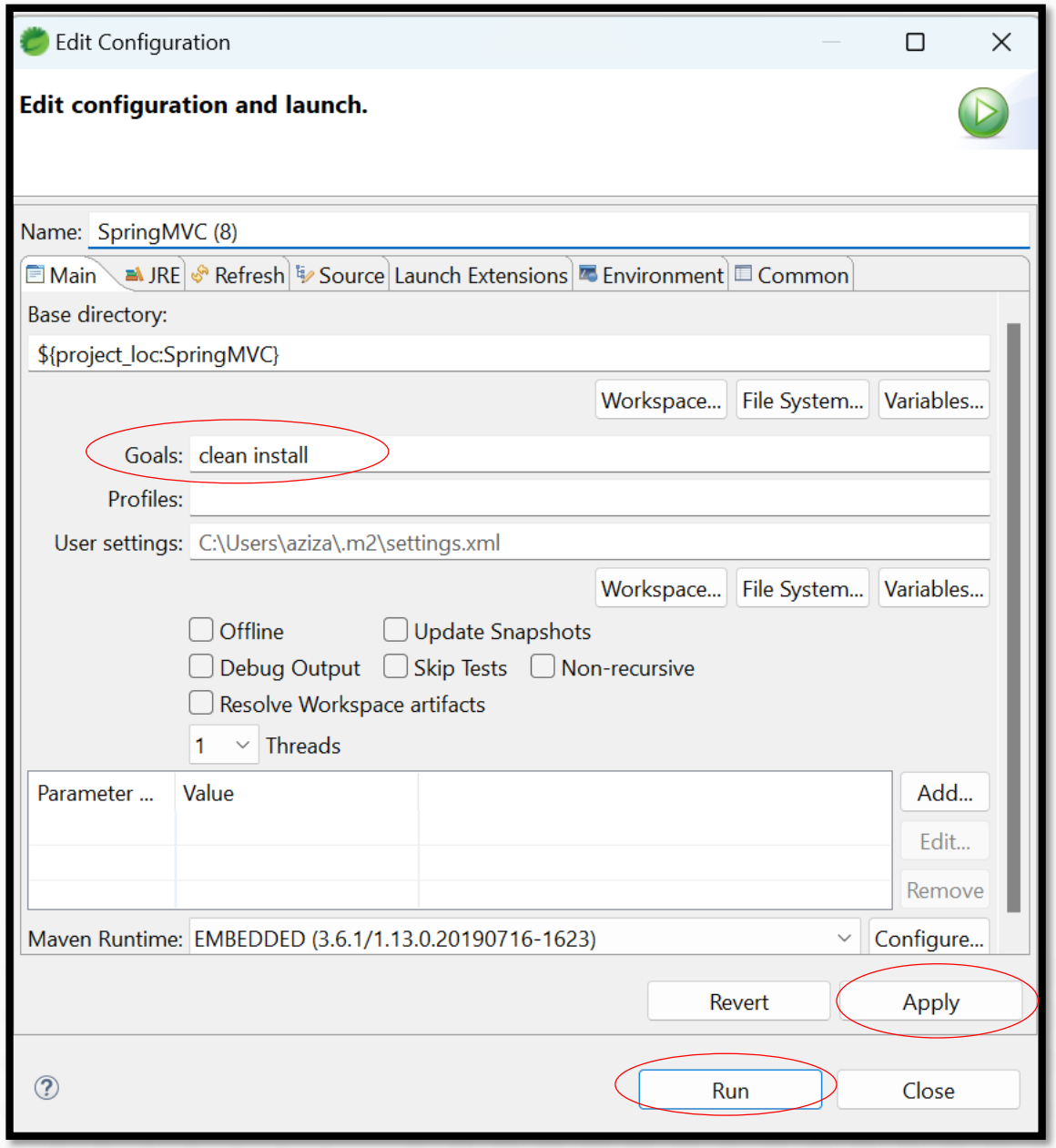
Run the Project

- To run the project:
  - Right click on your project
    - Select - Run As
    - Select - Maven build..



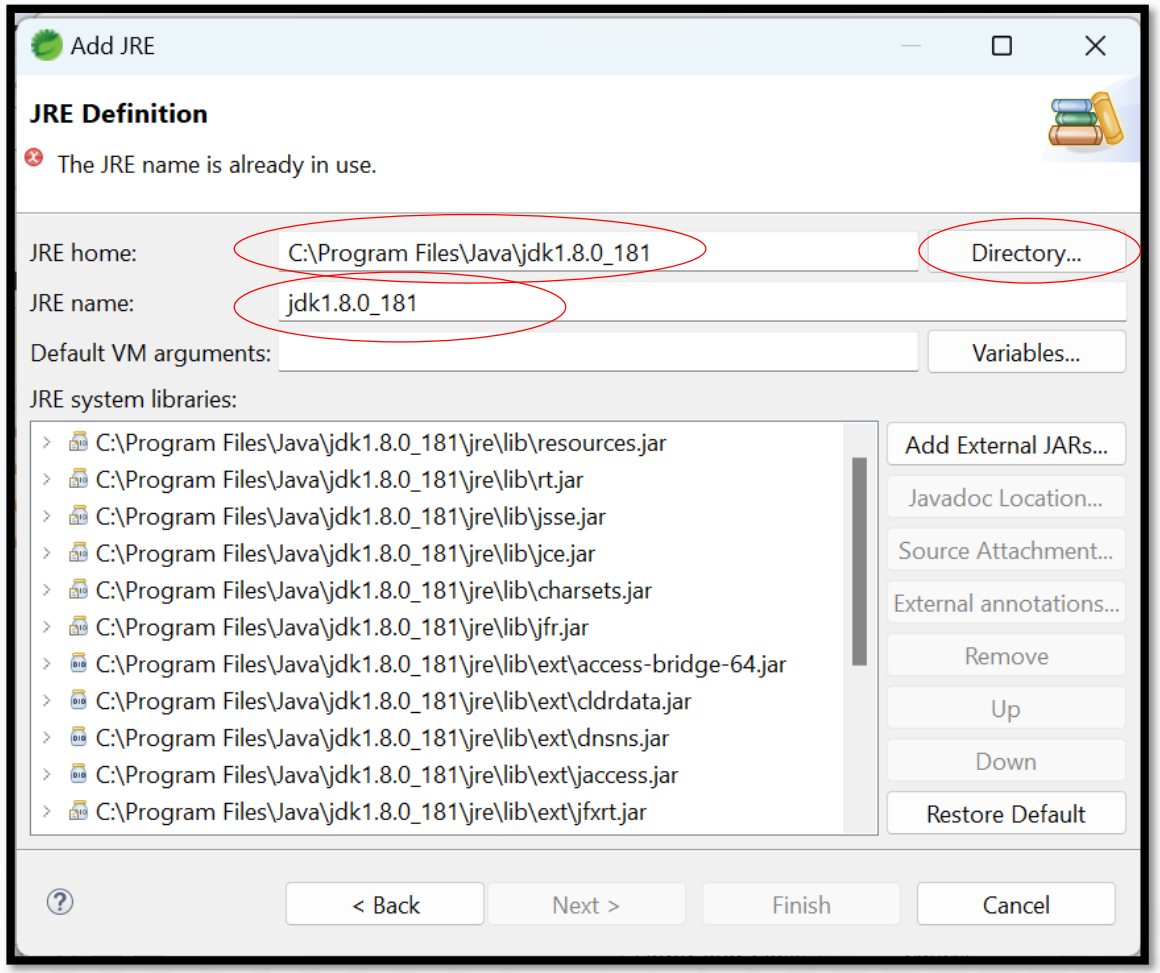
Run the Project

- Goals: Type - clean install
- Click - Apply
- Click - Run



How to fix if the Maven build is not success?

- Right Click on project
- -->Build Path
- --> Configure Build Path
- -->Add Library..
- -->JRE System Library --> Next
- --> Alternate JRE
- --> Installed JREs --> Add
- --> Standard VM --> Next
- --> Point to Java folder in C: drive (Windows) and select JDK folder and Finish.



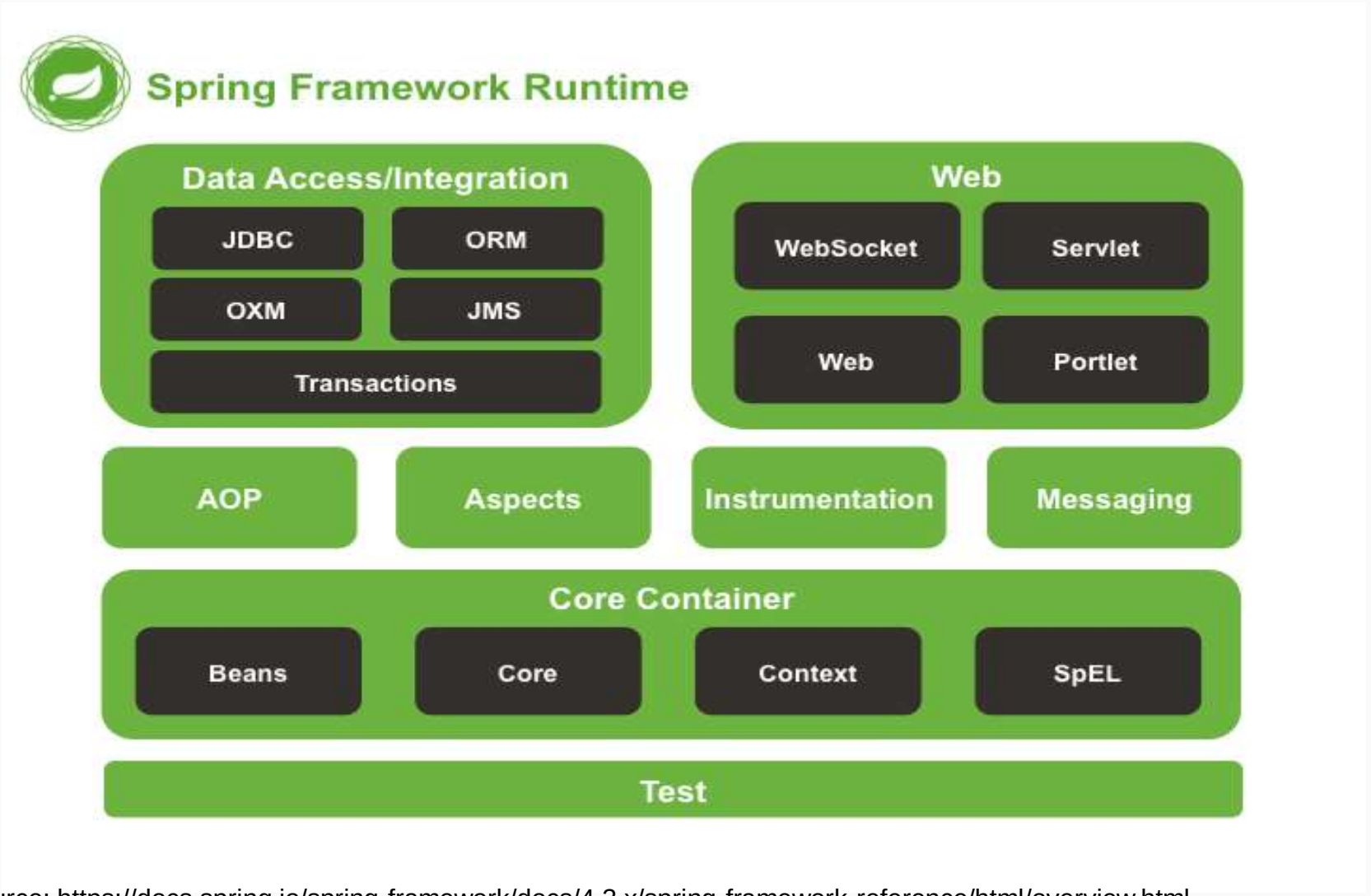
Display the Output

- To run the project:
  - ➢ Right click on your project
    - Select - Run As
    - Select - Run on Server
- Type in URL - http://localhost:8086/SpringMVC/welcome.html
- Example of output using Spring MVC:

Spring Framework

- Spring framework makes the easy development of JavaEE application.
- It is helpful for beginners and experienced persons.
- Spring is a *lightweight* framework.
- It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate, Tapestry, EJB, JSF etc.
- The Spring Framework consists of features organized into about 20 modules. These modules are grouped into:
  - Core Container
  - Data Access/Integration
  - Web
  - AOP (Aspect Oriented Programming)
  - Instrumentation
  - Messaging
  - Test

Spring Framework



Source: https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/overview.html

Spring Framework

Core Feature:

- **Dependency Injection** (DI) is also called as **Inversion of Control** (IoC) is the most important feature of the Spring Framework and it is at the core of all Spring Modules.

Why is this DI or IoC so important?

- It is a software design patterns which is really useful for designing loosely coupled software components.
- These loosely coupled applications can easily be tested and maintained.
- It is easy to reuse your code in other applications.
- The dependencies won't be hard coded inside all java objects/classes, instead they will be defined in XML configuration files or configuration classes (Java Config).
- Spring Container is responsible for injecting dependencies of objects.

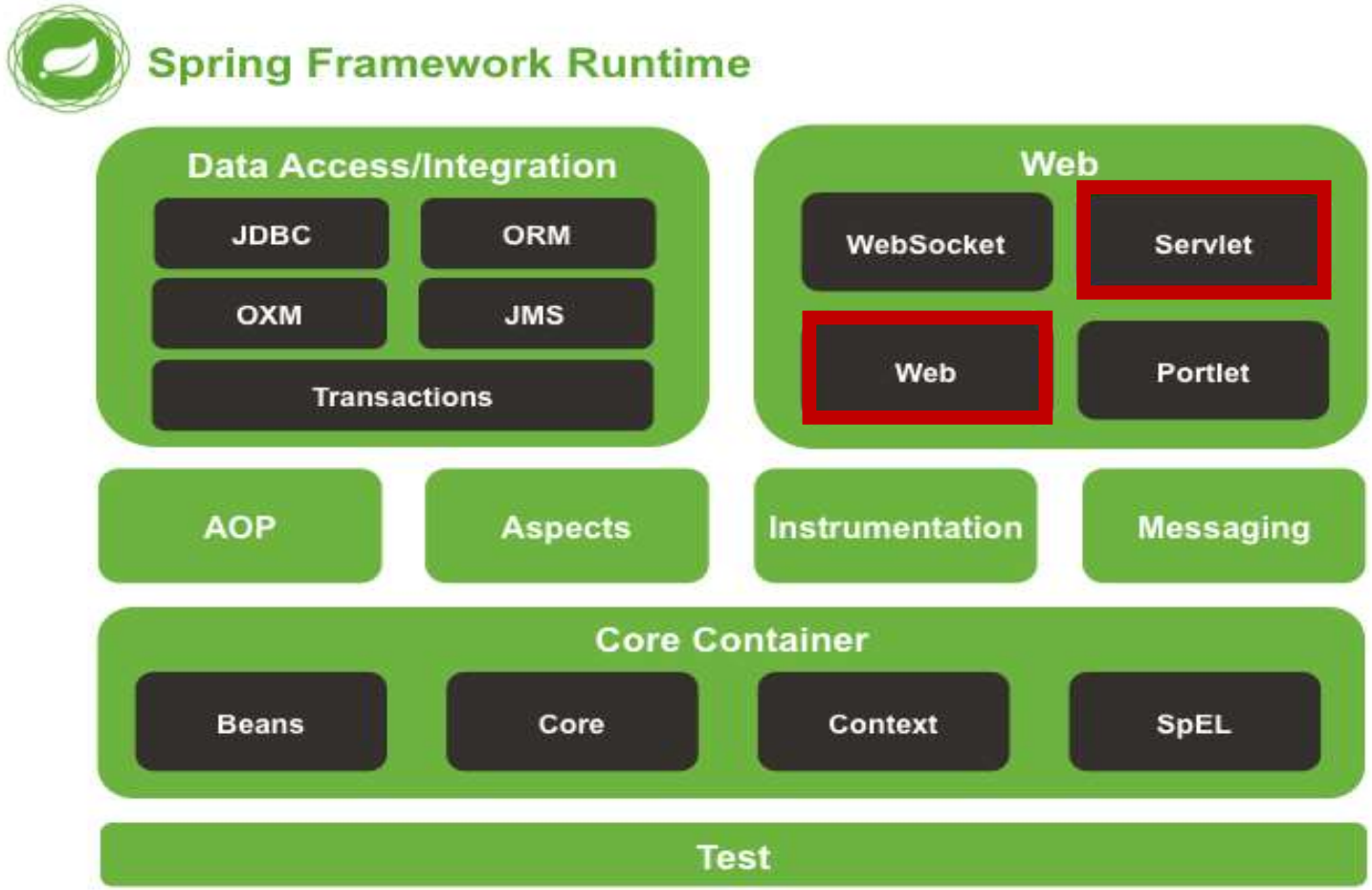
Advantages of Spring Framework

1) **Predefined templates**
   - Provides templates for JDBC, Hibernate, JPA etc. technologies. There is no need to write too much code. It hides the basic steps of these technologies.
2) **Loose coupling**
   - The Spring applications are loosely coupled because of dependency injection.
3) **Easy to test**
   - The Dependency Injection makes easier to test the application.
4) **Lightweight**
   - Lightweight because of its POJO implementation. It doesn't force the programmer to inherit any class or implement any interface.
5) **Fast development**
   - The Dependency Injection feature of Spring Framework and it support to

What is Spring MVC?

Module of Spring framework on Web layer:

- **Web Module** provides basic web-oriented integration features and the initialization of the IoC container using servlet listener and web application context.
- **Servlet Module** contains Spring MVC implementation for web application.



Source: https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/overview.html

Spring MVC

- Spring MVC is an open source MVC 2 framework for developing Java 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 and Dependency Injection.
- A Spring MVC provides a solution to use MVC in Spring framework by the help of **DispatcherServlet**.
- **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

Spring MVC Components

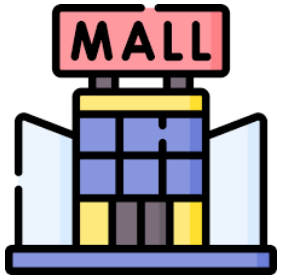
- **DispatcherServlet** responsible for intercepting the request and dispatching for specific URL.
- **HandlerMapping** is an interface that defines a mapping between requests and handler objects. It determines which controller to call.
- **Controller** responsible for processing user requests and building appropriate model and passes it to the view for rendering.
- **ModelAndView** class object encapsulates view and model linking.
- **Model** encapsulates the application data, will consist of POJO.
- **ViewResolver** provides a mapping between view names and actual view.
- **View** interface represents presentation logic and is responsible for rendering content

Spring Web MVC (Analogy)

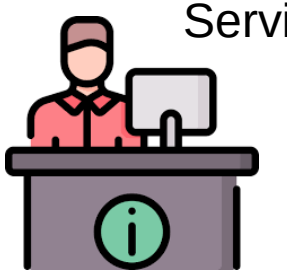
Customer

Request toy as a gift

Response toy as a gift



Customer Service



Spring Web MVC Flow

Request page 1

10 Response page

Handler Mapping

Get the URL and tell the exact address of the page that can generate data for the request

2 3 Dispatcher Servlet

Front Controller

4 5

Controller

Prepare the data, create a Java object to hold the data (it may take help from other Java classes if required)

Model And View

Java object holding the data and name of the final response builder

9 8

6 7

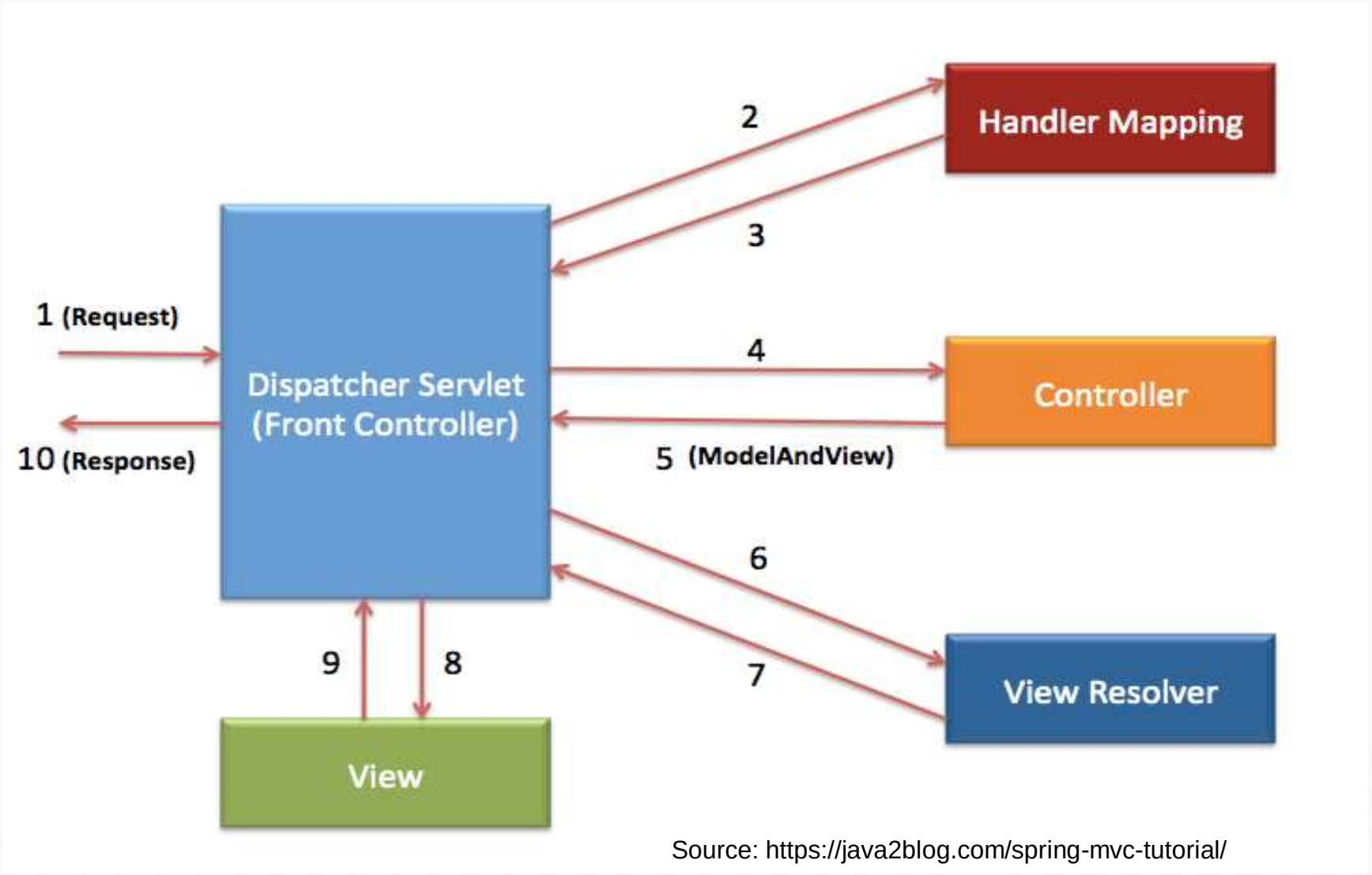
View

Display the actual view with the data

View Resolver

Mapping between view name and actual view (final response builder)

Spring Web MVC Flow



Source: https://java2blog.com/spring-mvc-tutorial/

Spring Web MVC Flow

1. The request will be received by Front Controller i.e. **DispatcherServlet**.
2. DispatcherServlet will pass this request to HandlerMapping. **HandlerMapping** will find suitable Controller for the request.
3. **HandlerMapping** will send the details of the controller to DispatcherServlet.
4. DispatcherServlet will call the **Controller** identified by HandlerMapping. The **Controller** will process the request by calling appropriate method and prepare the data. It may call some business logic or directly retrieve data from the database.
5. The **Controller** will send **ModelAndView**(Model data and view name) to **DispatcherServlet**.
6. Once DispatcherServlet receives ModelAndView object, it will pass it to **ViewResolver** to find appropriate View.
7. **ViewResolver** will identify the view and send it back to **DispatcherServlet**.
8. **DispatcherServlet** will call appropriate **View** identified by ViewResolver.
9. The **View** will create Response in form of **HTML** and send it to **DispatcherServlet**.
10. **DispatcherServlet** will send the response to the **browser**. The browser will render the html code and display it to **end user**.

Front Controller - DispatcherServlet

- Spring MVC is designed around a central servlet named **DispatcherServlet**.
- Every request is handled by the DispatcherServlet.
- DispatcherServlet is also called a **Front Controller**.
- It uses to handle all incoming requests and uses customizable logic to determine which controllers should handle which requests.
- It forwards all responses to, through view handlers to determine the correct views to route responses.
- It exposes all beans defined in Spring to controllers for dependency injection.

Front Controller (Analogy)

Customer

Request cake

Response cake

MyCafé

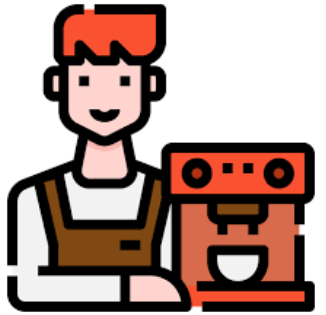
Waiter

Baker

cake

cake

Barista



coffee

coffee

Cashier

receipt

receipt

Front Controller

Client

/myCafe/cake

cake

myCafe

MyCafé

Front Controller

/myCafe/*

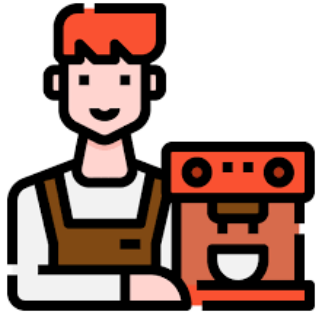
/cake

/coffee

/receipt

BakerController

BaristaController



PaymentController

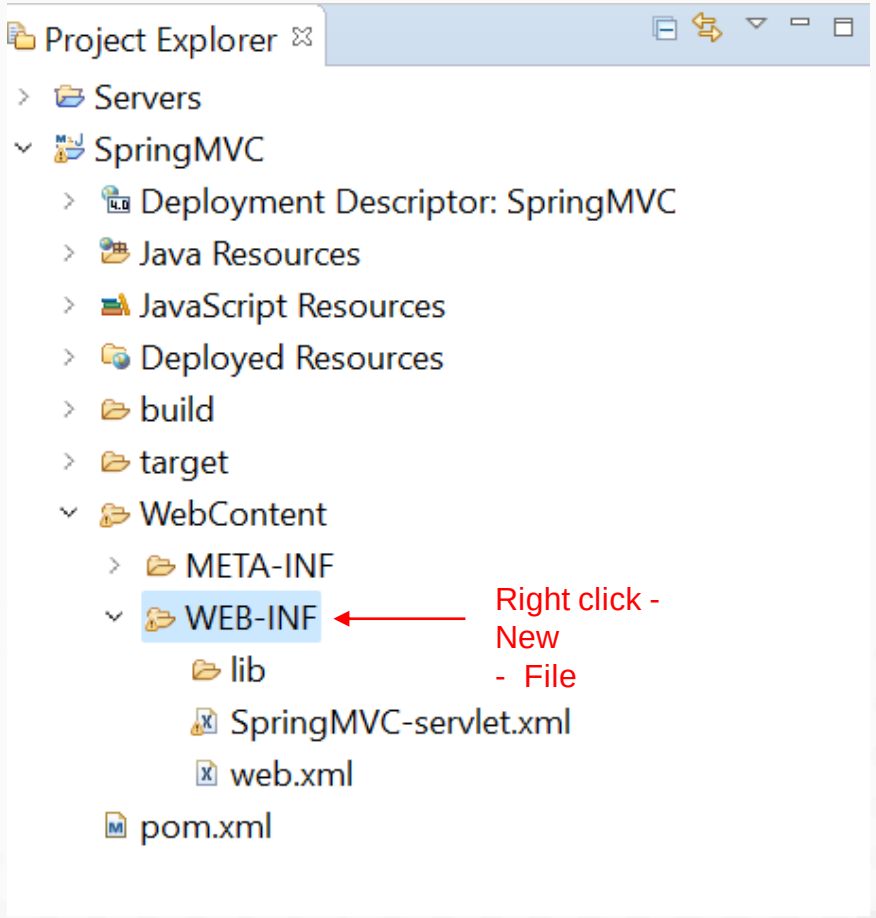
DispatcherServlet in web.xml

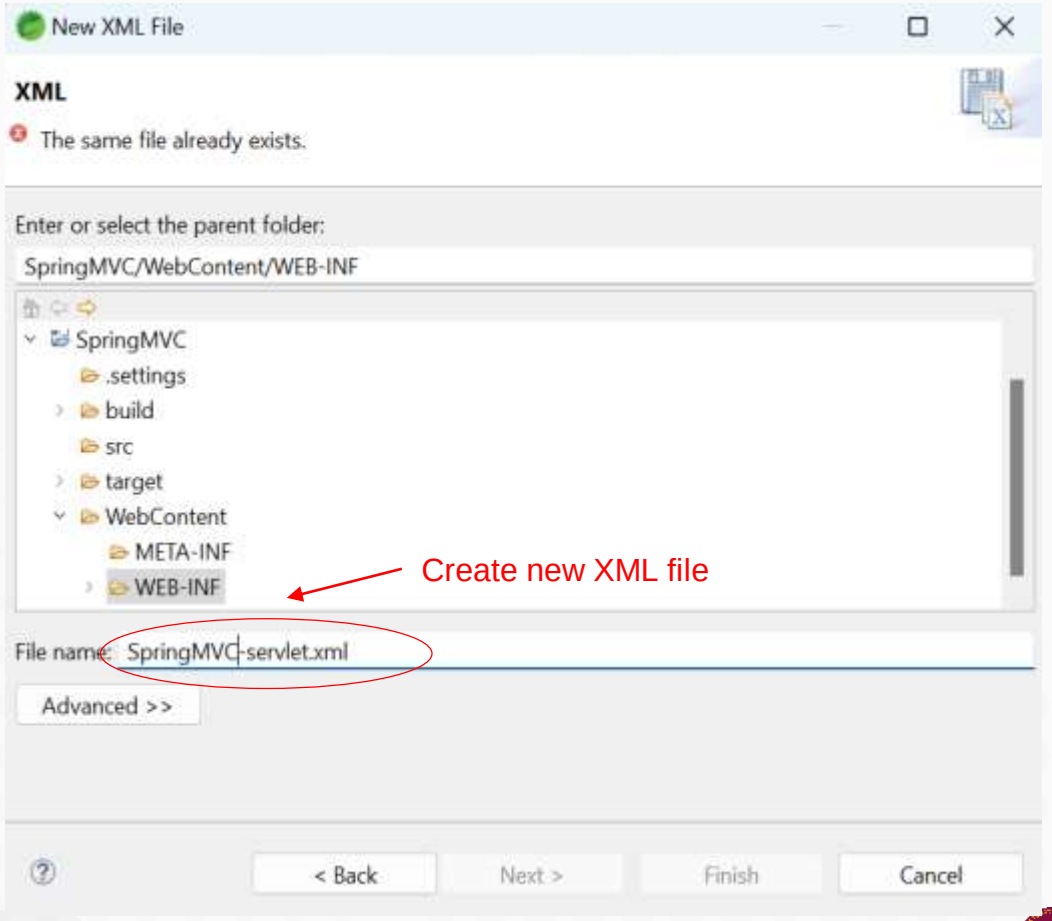
- **DispatcherServlet** is configured in order to dispatch incoming HTTP request to handlers and returns response to browsers.
- DispatcherServlet is configured in **web.xml** file.
- Example of DispatcherServlet in web.xml:

```
<servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/welcome.jsp</url-pattern>
        <url-pattern>/index.jsp</url-pattern>
        <url-pattern>/welcome.html</url-pattern>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
```

Spring Configuration

- By default, Spring looks for a **[servletname] – servlet.xml** file in the **/WEB-INF** folder to get the **DispatcherServlet** initialized**.**





Spring Configuration

- Example of **[servletname] – servlet.xml** file:

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

<bean id="HandlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name="/welcome.html" class="com.tutorial.controller.HelloController"/>

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

Controller

- **DispatcherServlet** delegates the request to the controllers to execute the functionality specific to it.
- The @Controller annotation indicates that a particular class serves the role of a controller.
- Controller interpret user input and transform this input into specific model which will be represented to the user by the view.
- **@Controller** annotation defines the class as a Spring MVC controller.
- **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

Controller – Without Annotation

- Example of **HelloController.java - controller** file:

```
package com.tutorial.controller;

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 HelloController extends AbstractController {

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

    ModelAndView modelandview = new ModelAndView("HelloPage");
    modelandview.addObject("welcomeMessage", "Hi user, welcome to the first Spring MVC
    Application");

    return modelandview;
    }
}
```

Controller – With Annotation

- The **@Controller** annotation indicates that a particular class serves the role of a **controller**.
- Spring uses the **@RequestMapping** method annotation to define the URI Template for the request. It can be applied to class-level and/or method-level in a controller.
- Example of **HelloController.java - controller** file with @Controller and @Request Mapping annotation:

```
package com.tutorial.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HelloController {

    @RequestMapping("/welcome")
    protected ModelAndView helloWorld(){

        ModelAndView modelandview = new ModelAndView("HelloPage");
        modelandview.addObject("welcomeMessage", "Hello World with annotation! ");

        return modelandview;
    }
}
```

Spring Configuration – With Annotation

- Example of **[servletname] – servlet.xml** file:

```
<?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.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">
<bean id="HandlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name="/welcome.html" class="com.tutorial.controller.HelloController"/>
<context:component-scan base-package="com.tutorial.controller" />

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

MultiAction Controller

- MultiAction Controller is a Controller implementation that allows multiple request types to be handled by the same class.
- It means inside one controller class we can have many handler methods.

class level

method level

http://localhost:8086/SpringMVC/greet/welcome

http://localhost:8086/SpringMVC/greet/selamat

```
@Controller
@RequestMapping("/greet")
public class HelloController {

    @RequestMapping("/welcome")
    protected ModelAndView helloWorld(){

    ModelAndView modelandview = new ModelAndView("HelloPage");
    modelandview.addObject("welcomeMessage", "Hello World with annotation! ");

    return modelandview;
    }

    @RequestMapping("/selamat")
    protected ModelAndView selamatDatang(){

    ModelAndView modelandview = new ModelAndView("HelloPage");
    modelandview.addObject("welcomeMessage", "Selamat Datang with annotation! ");

    return modelandview;
    }

}
```

@PathVariable Annotation

The **@PathVariable** annotation is used to extract the value of the template variables and assign their value to a method variable.

```
@Controller
public class HelloController {

	@RequestMapping("/welcome/{facultyName}/{userName}")
	protected ModelAndView helloWorld(@PathVariable ("facultyName") String faculty, @PathVariable ("userName") String name){

	ModelAndView modelandview = new ModelAndView("HelloPage");
	modelandview.addObject("welcomeMessage", "Hello " + name + " Your faculty is: " + faculty);

	return modelandview;
	}
}
```

User input

```
@Controller
public class HelloController {

	@RequestMapping("/welcome/{facultyName}/{userName}")
	protected ModelAndView helloWorld(@PathVariable Map<String,String> pathV ){

	String faculty = pathV.get("facultyName");
	String name = pathV.get("userName");

	ModelAndView modelandview = new ModelAndView("HelloPage");
	modelandview.addObject("welcomeMessage", "Hello " + name + "  Your faculty is: " + faculty);

	return modelandview;
	}
}
```

User input

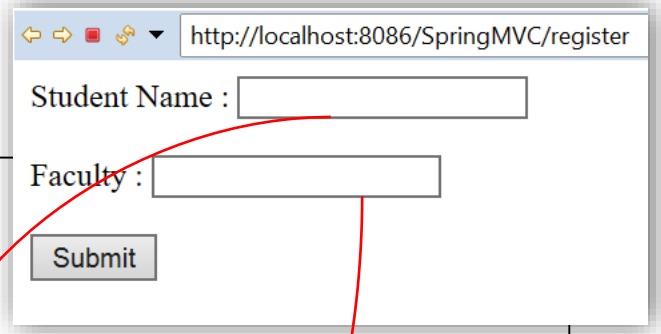
Same output with different way of using @PathVariable annotation in Map

Example of URL: http://localhost:8086/SpringMVC/welcome/FC/ali

@RequestParam Annotation

- **@RequestParam** is a Spring annotation used to bind a web request parameter to a method parameter.
- It has the following optional elements:
  - defaultValue - used as a fallback when the request parameter is not provided or has an empty value
  - name - name of the request parameter to bind to
  - required - tells whether the parameter is required
  - value - alias for name

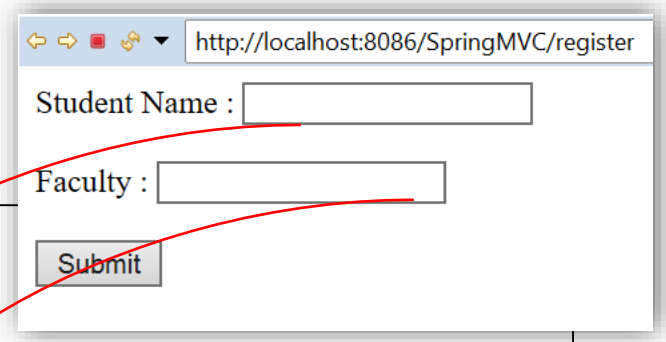
```
@Controller
public class RegisterController {

    @RequestMapping("/register")
    protected ModelAndView getRegisterForm( ){

        ModelAndView model = new ModelAndView("RegisterForm");

        return model;
    }

    @RequestMapping("/submit")
    protected ModelAndView submitRegisterForm(@RequestParam
    (“studentName”) String name, @RequestParam (“facultyName”)
    String faculty ){

        ModelAndView modelandview = new ModelAndView("HelloPage");
        modelandview.addObject("welcomeMessage", "Hello " + name +
        “, Your faculty is: " + faculty);

        return modelandview;
    }
}
```



@RequestParam Annotation

Same output with different way of using @RequestParam annotation in Map

```
@Controller
public class RegisterController {

	@RequestMapping("/register")
	protected ModelAndView getRegisterForm( ){

		ModelAndView model = new ModelAndView("RegisterForm");

		return model;
	}

	@RequestMapping("/submit")
	protected ModelAndView submitRegisterForm(@RequestParam Map<String,String> req ){

		String name = req.get("StudentName");
		String faculty = req.get("facultyName");

		ModelAndView modelandview = new ModelAndView("HelloPage");
		modelandview.addObject("welcomeMessage", "Hello " + name + ", Your faculty is: " + faculty);

		return modelandview;
	}
}
```



@ModelAttribute Annotation

- *@ModelAttribute* is an annotation that binds a method parameter or method return value to a named model attribute, and then exposes it to a web view.
- We can use *@ModelAttribute* either as a method parameter or at the method level.

Method level

```
@ModelAttribute
public void addAttributes(Model model) {
    model.addAttribute("msg", "Welcome to Malaysia!");
}
```

As a method parameter

```
@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)
public String submit(@ModelAttribute("employee") Employee employee) {
    // Code that uses the employee object

    return "employeeView";
}
```

@ModelAttribute Annotation

Method level (without @ModelAttribute)

```
@RequestMapping("/register")
protected ModelAndView getRegisterForm() {

     ModelAndView model = new
     ModelAndView("RegisterForm");
     model.addObject("headerMessage", "Welcome to
     University");
     return model;
}

@RequestMapping("/submit")
protected ModelAndView submitRegisterForm
(@RequestParam ("studentName") String name,
@RequestParam ("facultyName") String faculty ){

     ModelAndView model = new
     ModelAndView("HelloPage");
     model.addObject("welcomeMessage", "Hello " +
     name + ", Your faculty is: " + faculty);
     model.addObject("headerMessage", "Welcome to
     University");
     return model;
}
```

Method level (with @ModelAttribute)

```
@RequestMapping("/register")
protected ModelAndView getRegisterForm() {

     ModelAndView model = new ModelAndView("RegisterForm");
     return model;
}

@RequestMapping("/submit")
protected ModelAndView submitRegisterForm (@RequestParam
("studentName") String name, @RequestParam ("facultyName")
String faculty ){
     ModelAndView model = new ModelAndView("HelloPage");
     model.addObject("welcomeMessage", "Hello " + name + ",
     Your faculty is: " + faculty);
     return model;
}

@ModelAttribute
public void addAttributes(Model model) {
     model.addAttribute("headerMessage", "Welcome to
     University");
}
```

@ModelAttribute Annotation

As a method parameter (without @ModelAttribute)

```
@RequestMapping("/submit")
protected ModelAndView submitRegisterForm(@RequestParam ("studentName") String name,
@RequestParam ("facultyName") String faculty ){

    Student student1 = new Student();
    student1.setStudentName(name);
    student1.setFacultyName(faculty);

    ModelAndView model = new ModelAndView("HelloPage");
    model.addObject("student1", student1);

    return model;
}
```

As a method parameter (with @ModelAttribute)

Reduce line of code

```
@RequestMapping("/submit")
protected ModelAndView submitRegisterForm(@ModelAttribute ("student1") Student
student1 ){

    ModelAndView model = new ModelAndView("HelloPage");
    return model;
}
```

Model

- **Model** is generally defined as a MAP that can contain objects that are to be displayed in view.
- **ModelAndView** object encapsulates the relations between view and model and is returned by the corresponding Controller methods.
- **ModelAndView** class use **ModelMap** that is custom MAP implementation where values are added in key-value fashion.

ViewResolver

- The *ViewResolver* maps view names to actual views.
- Spring framework comes with quite a few view resolvers e.g. *InternalResourceViewResolver*, *BeanNameViewResolver,* and a few others.
- **InternalResourceViewResolver** is used to resolve the correct view based on **suffix** and **prefixes**, so the correct output (view) is resolved based on strings.

ViewResolver

/myCafe/cake

Hello here is your cake!

Front Controller

/myCafe/*

DispatcherServlet

/cake

HelloPage

BakerController

```
@Controller
public class BakerController {
      @RequestMapping("/cake")
      protected ModelAndView
      requestCake(){
            ModelAndView modelandview =
            new
            ModelAndView("HelloPage");
            modelandview.addObject("msg",
            "Hello here is your cake! ");
            return modelandview;
            }
            }
```

HelloPage

View Resolver

WEB-INF/HelloPage.jsp

View

prefix

suffix

WEB-INF/ HelloPage .jsp

ViewResolver

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

View

- Java Spring MVC can be configured with different view technologies such as JSF,JSP, Velocity, Freemaker etc.
- **View** is responsible for displaying the content of Model objects on browsers as response output.
- View page can be explicitly returned as part of **ModelAndView** object by the controller.
- The view name can be independent of view technology and resolved to specific technology by using **ViewResolver** and rendered by **View**.

View

- Example of **HelloPage.jsp - view** file:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"  %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>First Spring MVC</title>
</head>
     <body>
               <h1>First Spring MVC Application</h1>
               <h2>${welcomeMessage}</h2>
     </body>
</html>
```

Exception Handling: @ExceptionHandler Annotation

- Inside Spring MVC, users can define an exception handling class by implementing as below:

Example for Null Pointer Exception

```
@RequestMapping("/register")
protected ModelAndView getRegisterForm() throws Exception {

    String exceptionOccured = "NULL_POINTER";
    if (exceptionOccured.equalsIgnoreCase("NULL_POINTER")){
        throw new NullPointerException("Null Pointer Exception");
    }

    ModelAndView model = new ModelAndView("RegisterForm");
    return model;
}

@ExceptionHandler (value=NullPointerException.class)
public String handleNullPointerException(Exception e) {

//logging Null Pointer Exception
System.out.println ("Null Pointer Exception Occured: "+e);
return "NullPointerException";
}
```

Create a new JSP file with same name of the return value to display the output of error.

Exception Handling: @ControllerAdvice Annotation

Without @ControllerAdvice

```
//In part of RegisterController.java

@ExceptionHandler (value=NullPointerException.class)
public String handleNullPointerException(Exception e)
{

//logging Null Pointer Exception
System.out.println ("Null Pointer Exception Occured:
"+e);
return "NullPointerException";
}

@ExceptionHandler (value=IOException.class)
public String handleIOException(Exception e) {

//logging Null Pointer Exception
System.out.println ("IO Exception Occured: "+e);
return "IOException";
}
```

With @ControllerAdvice as Global Exception Handler

```
//In GlobalExceptionHandlerMethods.java

@ControllerAdvice
public class GlobalExceptionHandlerMethods {

    @ExceptionHandler (value=NullPointerException.class)
    public String handleNullPointerException(Exception e)
    {

    //logging Null Pointer Exception
    System.out.println ("Null Pointer Exception Occured:
    "+e);
    return "NullPointerException";
    }

    @ExceptionHandler (value=IOException.class)
    public String handleIOException(Exception e) {

    //logging Null Pointer Exception
    System.out.println ("IO Exception Occured: "+e);
    return "IOException";
    }
}
```

HTTP Messages

- 202 OK: Request successfully processed by the Server.
- 404 Not Found: Requested Resource is not available.
- 500 Internal Server Error: An unexpected condition/error occurred while processing a request.
- 503 Service Unavailable: The server is currently unavailable (because it is overloaded or down for maintenance).

- **Client Error 4xx
- **Server Error 5xx

- For complete list of Status code:

https://www.rfc-editor.org/rfc/rfc9110.html#name-client-error-4xx

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

TOPIC 7 – Spring Web MVC

The End

innovative • entrepreneurial • global

UTM JOHOR BAHRU