INTRODUCTION TO SPRING WEB

LESSON

TCP

- TCP (Transmission Control Protocol) is a standard that defines how to establish and maintain a network conversation via which application programs can exchange data. TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other.

HTTP

- HTTP (Hypertext Transfer Protocol) is the set of rules for transferring files (text, graphic images, sound, video, and other multimedia files) on the World Wide Web. As soon as a Web user opens their Web browser, the user is indirectly making use of HTTP. HTTP is an application protocol that runs on top of the TCP/IP suite of protocols (the foundation protocols for the Internet).

- For example, when a Web server sends an HTML file to a client, it uses the HTTP protocol to do so. The HTTP program layer asks the TCP layer to set up the connection and send the file. The TCP stack divides the file into packets, numbers them and then forwards them individually to the IP layer for delivery. Although each packet in the transmission will have the same source and destination IP addresses, packets may be sent along multiple routes. The TCP program layer in the client computer waits until all of the packets have arrived, then acknowledges those it receives and asks for the retransmission on any it does not (based on missing packet numbers), then assembles them into a file and delivers the file to the receiving application.

TOMCAT

- TomCat, in its simplest concept, is a web server. Well, it's more than that. While it can serve static files, its primary purpose is to act as a 'servlet container' that serves Java web applications. It can process [.jsp] files, which are like PHP scripts but for Java, and it can also run [Java Servlets], which are classes that process the GET, POST, and other HTTP requests. TomCat will listen on TCP ports for HTTP requests, and route requests to your Java classes, JSP files, or static files. It is also possible to embed TomCat in to a standalone application, but that is out of the scope of this document.

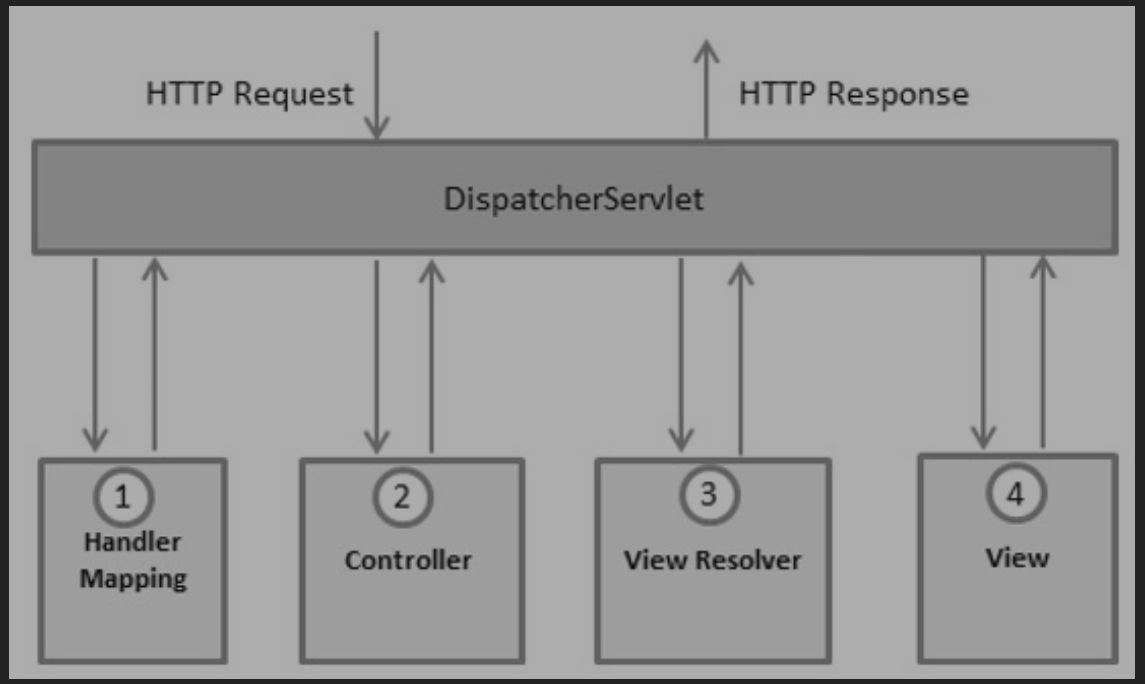
SPRING MVC

- The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
- The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

THE DISPATCHER SERVLET

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


HTTP Request
HTTP Response
DispatcherServlet
1
Handler Mapping
2
Controller
3
View Resolver
4
View

- Following is the 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*.

- 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.

- All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* w which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

REQUIRED CONFIGURATION

- You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>com.javajava.todolist.web.HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

</web-app>
```

HelloServlet.java

```java
public class HelloServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse resp) throws ServletException, IOException {

        String param1 = req.getParameter( name: "param1");

        // Set response content type
        resp.setContentType("text/html");

        // Prepare output html
        PrintWriter out = resp.getWriter();
        out.println("<h1>" + "Hello WWW world from Java!" + "</h1>");
        out.println("<h1>" + "Param 1 = " + param1 + "</h1>");
    }

}
```

DEFINING CONTROLLER

- The 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.
  The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

- The **@Controller** annotation defines the class as a Spring MVC controller. Here, the first usage of **@RequestMapping** indicates that all handling methods on this controller are relative to the **/hello** path. Next annotation**@RequestMapping(method = RequestMethod.GET)** is used to declare the printHello() method as the controller's default service method to handle HTTP GET request. You can define another method to handle any POST request at the same URL.

```java
@Controller
@RequestMapping("/hello")
public class HelloController {
    @RequestMapping(method = RequestMethod.GET)
    public String printHello(ModelMap model) {
        model.addAttribute( attributeName: "message",  attributeValue: "Hello Spring MVC Framework!");
        return "hello";
    }
}
```

- The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request.
    - You will define required business logic inside a service method. You can call another method inside this method as per requirement.
    - Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute “message".
    - A defined service method can return a String, which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

CREATING JSP VIEWS

- Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Hello Spring MVC</title>
</head>

<body>
<h2>${message}</h2>
</body>
</html>
```

Here **${message}** is the attribute which we have set up inside the Controller.
You can have multiple attributes to be displayed inside your view.

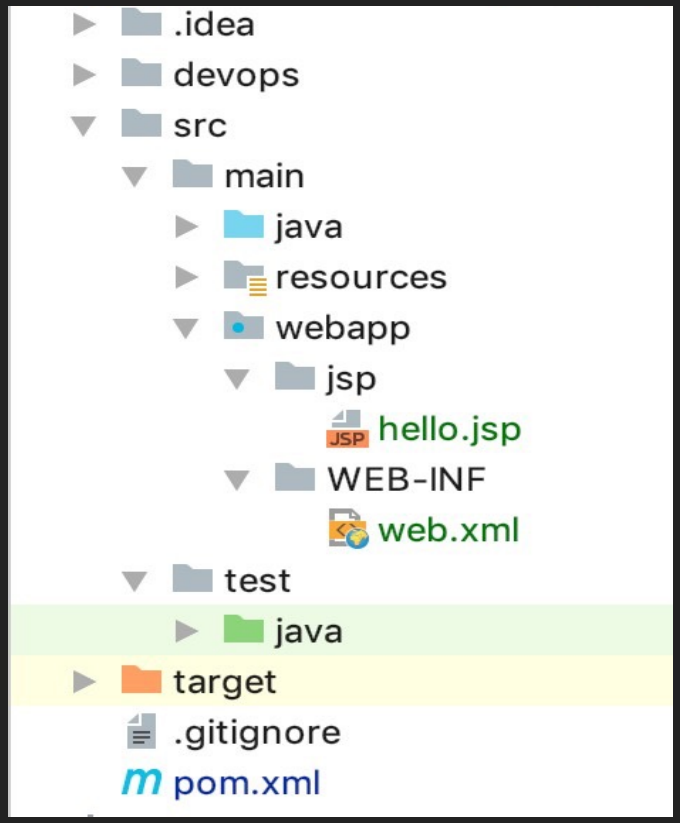
WEB MVC CONFIGURER

- Defines options for customizing or adding to the default Spring MVC configuration enabled through the use of **@EnableWebMvc**. The **@Configuration** class annotated with **@EnableWebMvc** is the most obvious place to implement this interface. However all **@Configuration** classes and more generally all Spring beans that implement this interface will be detected at startup and given a chance to customize Spring MVC configuration provided it is enabled through @EnableWebMvc.
- Implementations of this interface will find it convenient to extend WebMvcConfigurerAdapter that provides default method implementations and allows overriding only methods of interest.

```java
public class SpringWebMVCConfig extends WebMvcConfigurerAdapter {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

Project Structure



@RESTCONTROLLER

- The *@RestController* annotation was introduced in Spring 4.0 to simplify the creation of RESTful web services. **It's a convenience annotation that combines** *@Controller* **and** *@ResponseBody* – which eliminates the need to annotate every request handling method of the controller class with the *@ResponseBody* annotation.

REFERENCES

- https://www.baeldung.com/spring-mvc-tutorial
- https://www.baeldung.com/spring-mvc
- https://www.baeldung.com/spring-requestmapping
- https://www.devdungeon.com/content/tomcat-tutorial
- https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm
- https://www.baeldung.com/spring-controller-vs-restcontroller
- https://www.json.org/