**LOMBOK – JAVA LIBRARY:**

-Used to generate boilerplate codes.

-**Methods will be generated** by Lombok – simplifies development.

-Uses **Annotation processor API.**

**--------------------------------------------------------------------------------------------------------------------------------------**

**Annotations in LOMBOK:**

**@Getter**

**@Setter**

**@ToString**

**@EqualsAndHashcode**

**@RequiredArgsConstructor**

**@Slf4j**

**--------------------------------------------------------------------------------------------------------------------------------------**

**-@Data:**

Generate boilerplate codes for POJO.

Combine all the above mentioned Annotations together.

--------------------------------------------------------------------------------------------------------------------------------------Add **it above Fields or above Class**.

When added above class,

To avoid a Obj or any methods not to be involved.

USE: **@AnnotationType(AccessLevel.NONE)**

**Spring MVC?**

* **Spring Web MVC is a Web framework built on Servlet API.**

**MVC – M**odel, **V**iew and **C**ontroller

* This pattern separates the internal representation of information from the way it is presented to the user.
* Model: Responsible for managing the application’s data, business logic and rules.
* View: Output representation of information. Like displaying information or reports to the user either as WEB FORM or CHARTS.
* Controller: Invoking Models to perform business logic and then update Views based on the Model Outputs.

Central Servlet -> **DispatcherServlet**

- It expects a **WebApplicationContext,** which is a extension of a plain **ApplicationContext**.

- It delegates Special beans to process Request and Render Appropriate Responses.

**Steps:**

1.Logback.xml in resources.

2. Add **<packaging>war</packaging>** in pom.xml

3.Add dependencies and plugins (also war and cargo) in pom.xml.

The WAR Plugin is responsible for collecting all artifact dependencies, classes and resources of the web application and packaging them into a web application archive.

With CARGO Plugin, we can use goals to manipulate war projects within Tomcat servlet container. We can run tomcat in embedded mode.

4.Src -> main -> NEW DIRECTORY(webapp) ->NEW DIRECTORY(WEB-INF).

Webapp- css, html, js files…

WEB-INF- jsp files, thymeleaf templates, configurations files in this directory cannot be directly served to a client by container, but its visible to servlet code. We can use servlet to process and display it to the user.

**ALL REQUESTS ARE FORWARDED TO JSP VIA CONTROLLER – JSP= JAVA SERVER PAGES**

5.webapp-> index.html.

6.WEB-INF-> web.xml (its just a empty schema) or you can ignore .xml file and add **<failOnMissingWebXml>false</failOnMissingWebXml>**

**> Reload Project**

**> Start Tomcat? Maven Pane - Plugins - Cargo - cargo:run**

**> Go to http://localhost:8080/todo-list/**

**Add Dependencies in pom.xml:**

**<dependencies>**

<dependency>

<groupId>javax.annotation</groupId>

<artifactId>**javax.annotation-api**</artifactId>

<version>1.3.2</version>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>**lombok**</artifactId>

<version>1.18.24</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>**logback-classic**</artifactId>

<version>1.4.5</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>**spring-webmvc</**artifactId>

<version>5.3.24</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

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

<version>4.0.1</version>

<scope>**provided**</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>**jstl**</artifactId>

<version>1.2</version> </dependency>  **</dependencies>**

**Add plugins in pom.xml:**

<build>

<plugins>

<plugin>

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

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

<version>3.8.1</version>

<configuration>

<release>19</release>

<target>19</target>

<source>19</source>

</configuration>

</plugin>

<plugin>

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

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

<version>3.3.2</version>

<configuration>

**<failOnMissingWebXml>false</failOnMissingWebXml>**

</configuration>

</plugin>

<plugin>

<groupId>org.codehaus.cargo</groupId>

<artifactId>**cargo-maven3-plugin**</artifactId>

<version>1.9.8</version>

<configuration><container>

<containerId>**tomcat9x**</containerId>

<type>embedded</type>

</container></configuration>

</plugin>

</plugins>

</build>

**Add Logback.xml: [CHANGES]**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">

<pattern>%date [%thread] [%-5level] %logger{40} - %message%n</pattern>

</encoder>

</appender>

<logger name="com.timothy" level="**TRACE**"/>

<logger name="org.springframework" level="**TRACE**"/>

<root level="**TRACE**">

<appender-ref ref="STDOUT"/>

</root>

</configuration>

**Controller?**

We can map requests to methods in a class using @Controller. These classes are called annotated controllers or controller class.

@RequestMapping: used to map requests to controller methods.

**Shortcuts in @RequestMapping:**

@GetMapping, @PostMpaiing, @PutMapping, @DeleteMapping, @PatchMapping.

**View Resolver?**

ViewResolver and view interface will help us to render models in a browser without forcing to use specific technologies like JSP,ThymeLeaf

ViewResolver provides mapping between view names and actual views.

**JSP?**

* A text document that contains static data like html and jsp elements for dynamic context.
* We going to use JSTL(java server pages standard tag library)
* JSTL is a component of JAVA EE Web application development platform.

**@Service? [Works on service layer]:**

* Hold the business logic.
* Controllers will use the Service to get some data or to get the result of a calculation.

**JSTL Tags**

* Core,Formatting,XML,SQL and JSTL tags.
* Core tags:

(C: for each) used for iterating over a collection

(C: out) used to display the results of an expression

(C: url) format a URL into a string and then store it into a variable

**@RequestParam:**

* We are going to display something based on request parameter(query) instead of hard coded value inside the handler method.

**Instead of this,**

**@GetMapping**("welcome")

public String welcome(Model model){

model.addAttribute("helloMessage",demoService.getHelloMessage(" Paul"));

log.info("model={}", model);

return "welcome";

}

**We use this,**

**@GetMapping**("welcome")

public String welcome**(@RequestParam** **String user**, Model model){

model.addAttribute("helloMessage",demoService.getHelloMessage(**user**));

log.info("model={}", model);

return "welcome";

}

**To pass the parameter or query, Go to browser with the url and append it.**

http://localhost:8080/todo-list/welcome?user=**Paul**

**To use multiple parameters,**

**@GetMapping**("welcome")

public String welcome**(@RequestParam** **String user, @RequestParam int age**, Model model) {

model.addAttribute("helloMessage",demoService.getHelloMessage(**user**));

model.addAttribute("age",age);

log.info("model={}", model);

return "welcome"; }

**To pass the parameter or query, Go to browser with the url and append it.**

http://localhost:8080/todo-list/welcome?user=Paul**&**age=23

**Spring MVC Request Processing:**

**1.Request(**browser**)->DispatcherServlet.**

**2.DispatcherServelet->Handler Mapping(**identifying which handler can handle the request**).**

**3.Handler Mapping(**Returns Handler Methods) **-> DispatcherServlet.**

**4.DispatcherServlet->HandlerController(**Calls Methods in the controller).

**5.HandlerConrtoller(**Returns Model and view Name**)->DispatcherServlet.**

**6.DispatchServlet->Vie Resolver(**determines which view file to use**).**

**7.ViewResolover(**Finds the JSP file and returns it)->**DispatcherServlet.**

**8.DispatcherServelt(Executes the model and make it available to View)->View.**

**9.View(**renders the content and returns)->**DispatcherServlet.**

**10.DispatcherServlet->Response(Browser).**

**Setup Dispatcher Servlet using java config [No xml file]**

**DS – front controller of spring MVC and it is used to dispatch Http request to other controllers.**

**2 ways to register servlet in our application:**

1. Web.xml – xml based configuration
2. **Java based configuration**

**Steps:**

1. Src-> main-> java->NEW PACKAGE(com.timothy)->config Folder
2. Config->NEW CLASS(WebConfig.java)
3. ADD @Configuration @ComponentScan(basepackages=”com.timothy”) @EnableWebMVC Now our context is ready.
4. Now we need to configure Dispatcher Servlet – ADD Dependencies (servlet api)
5. Servlet Registration? Implement WebApplicationIntializer interface config-> NEW CLASS(WebAppIntializer)
6. There will be 404 error, could not see anything on localhost, so we **ADD Controller? Com.timothy->NEW PACKAGE(controller)->NEW JAVA CLASS(DemoController) ADD @Controller,** After using controller there will be error 500.(we didn’t used any view resolver) to display to user. **Use @ResponseBody** to view output without defining view resolver.
7. View resolver? WEB\_INF->NEW DIRECTORY(view)->right click->NEW FILE(welcome.jsp) type some html and We need to **configure viewResolver Bean** **Go to WebConfig class and add constants prefix and suffix** private final static String RESOLVER\_PREFIX="/WEB\_INF/view/";

private final static String RESOLVER\_SUFFIX=".jsp";

now we create bean,

**@Bean**

public ViewResolver viewResolver(){

UrlBasedViewResolver viewResolver= **new InternalResourceViewResolver();**

viewResolver.setPrefix(RESOLVER\_PREFIX);

viewResolver.setSuffix(RESOLVER\_SUFFIX);

return viewResolver;

}

Go to **Controller Class**,

@GetMapping(“welcome”)

public String welcome(){

return “**welcome”**;

}

About the above code, We can’t see what’s inside of WEB-INF folder,

the value of the method returns does matter and it makes use of Internal view resolver prefix and suffix…. By doing this we can access whats inside….

**Prefix + Returned Value+ suffix**

**WEB-INF/view/welcome.jsp**

1. Model Interface? Adding attributes to the model. The model is exposed to the view and it can access the jsp file. **@GetMapping("welcome")**

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

**model.addAttribute("user","paul");**

**log.info("model={}", model);**

**return "welcome";**

**}**

**Change JSP File, To use this attribute,**

**In our case, ${user} -> this will display our attribute value. Model is exposed to View and it reads the value to the output. PTO**

the model is created by the dispatcher servlet and it invokes our request methods once we get model as a parameter we can then add attributes to the model and then those attributes are available in the view. In general our model represents data that will be passed from a controller method to the view as well as data passed from the view back to the controller method whether forms included in the view.(model is more like a key value pair),

**WE CAN ALSO ADD MODEL ATTRIBUTE BY USING @ModelAttribute**

**@ModelAttribute("welcomeMessage")**

**public String welcomeMessage(){**

**log.info("model attributes get called first");**

**return "welcome to my page";**

**}**

**Changes in JSP file,**

**${ welcomeMessage }**

1. **Add @Service above Impl class, Add the interface object in the controller and Autowire Constructor of the interface object. Make changes in the handler methods according to service annotated class.**