**CDP: Infogain Certified Spring Boot Developer**

Objective of the course:

* Define and demonstrate Spring boot
* Developing and deploying application using spring boot
* Design and develop web applications using spring boot
* Develop and deploy applications using JPA and Spring Boot.
* Develop and deploy rest application
* Define and Demonstrate Actuator
* Define and demonstrate HAL
* Define and Demonstrate spring boot messaging (RabbitMQ)
* Develop and deploy application for Spring boot documentation
* Developing end to end application using spring boot
* Developing and deploying cloud native application.

1. **Developing Web Application Using Spring Boot:**

* Spring Boot Initializer
* Group Id
* Artifact ID
* Dependencies:
* --- Spring Boot Starter (Web Starter would bring us lot of dependencies)
* ----Add Dev Tools (for more faster development)
* Showing the how to create project using spring initializer and sts.

**Explaining the starter POM :**

**Open starter Web and see like java ,tomcat etc .**

**Important Annotations:**

* @Controller
* @RequestMapping
* @RequestParam
* @ModelAttribute
* @RequestBody and @ResponseBody
* @RequestHeader and @ResponseHeader

**Create a controller for default path :**

@Controller

**public** **class** LoginController {

@RequestMapping("/")

@ResponseBody //this annotation is used send text in response).

//The @ResponseBody annotation tells a controller that the object returned is //automatically serialized into JSON and passed back into the HttpResponse objec

**public** String login()

{

**return** "login modified ";

}}

**package** cdp.soft.infogain.controller.service;

@Component

**public** **class** LoginService {

**public** **boolean** validateUser(String username, String password) {

**if** (username.equals(password))

**return** **true**;

**else**

**return** **false**;

}

}

**Adding View pages (JSP in my case ):**

**Enable JSP :**

All JSP pages need to be placed in - /src/main/webapp/WEB-INF/jsp

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

**After adding all JSP pages we need to add view resolver:**

spring.mvc.view.prefix: /WEB-INF/jsp/

spring.mvc.view.suffix: .jsp

**Set the log level to know about errors :**

**logging.level.org.springframework.web=DEBUG**

Explore the request on DevTools in Firefox .

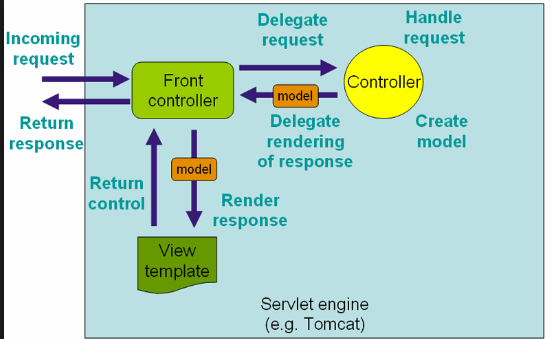
**What You Will Learn during this Step:**

* You first GET Parameter.
* Problem with using GET

Explain Model View Controller

## Spring MVC Request Flow

* DispatcherServlet receives HTTP Request.
* DispatcherServlet identifies the right Controller based on the URL.
* Controller executes Business Logic.
* Controller returns a) Model b) View Name Back to DispatcherServlet.
* DispatcherServlet identifies the correct view (ViewResolver).
* DispatcherServlet makes the model available to view and executes it.
* DispatcherServlet returns HTTP Response Back.
* Flow : <http://docs.spring.io/spring-framework/docs/2.0.8/reference/images/mvc.png>



**Adding Model data to view :**

* Model is an interface while **ModelMap** is a class.
* **ModelAndView** is just a container for both a **ModelMap** and a view object. It allows a controller to return both as a single value.

In Controller add below code :

ModelMap model

model.put("name", name);

My First JSP!!! My name is ${name}

@RequestMapping("/login")

public String loginMessage(@RequestParam String name, ModelMap model){

model.put("name", name);

return "login";

}

}

**Creating and Login application:**

Create an application with loginApp and Add login page, controller.

**${errorMessage}**

Step 9

Talking about Spring DI , Tightly coupled application

1. Annotation

**@Component, @Autowired, @Service**

@**Component**  generic stereotype for any Spring-managed component

@**Repository** stereotype for persistence layer

@**Service** stereotype for service layer

@**Controller** stereotype for presentation layer (spring-mvc)

Open @SpringBootApplication

**Adding Attribute in Session :**

* Session vs Model vs Request.
* Be cautious about what you use Session for.
* @SessionAttributes("name") and how it works?
* Why use Model? "adding elements directly to the HttpServletRequest (as request attributes) would seem to serve the same purpose. The reason to do this is obvious when taking a look at one of the requirements we have set for the MVC framework: It should be as view-agnostic as possible, which means we’d like to be able to incorporate view technologies not bound to the HttpServletRequest as well." - Rod Johnson et. al’s book Professional Java Development with the Spring Framework
* Spring documentation states that the @SessionAttributes annotation “list the names of model attributes which should be transparently stored in the session or some conversational storage.”

**Put Name is session :**

**@SessionAttribute in login Controller at top .**

After that add name in Model attribute

Model.addAttribute (“name”,name)

<HTML>

<BODY>

You have Logged in as ${name}

<hr>

<h2>

Welcome <b>${name}</b> to the System !!

</h2>

<br>

<a href=*"/logout"*> Click here to logout!!</a>

<br>

</BODY>

</HTML>

**Webjars :**

## 1. Introduction

Among the other features that Spring Boot simplifies, one of them is the webjars. Webjars are the static client-side dependencies (for eg. javascript libraries) packaged into JARs. Manually adding the client side libraries could be difficult to maintain, so with webjars (feature enhanced in Spring Boot), we can add these libraries just by making required pom configuration entries, and we are all-set to start using them instantly.

|  |
| --- |
| <dependency>  <groupId>org.webjars</groupId>  <artifactId>bootstrap</artifactId>  <version>3.3.6</version>  </dependency>    <dependency>  <groupId>org.webjars</groupId>  <artifactId>jquery</artifactId>  <version>1.9.1</version>  </dependency> |

We can now directly use the client-side libraries in our view pages like –

<link href=*"webjars/bootstrap/3.3.6/css/bootstrap.min.css"* rel=*"stylesheet"*>

<script src=*"webjars/jquery/1.9.1/jquery.min.js"*></script>

<script src=*"webjars/bootstrap/3.3.6/js/bootstrap.min.js"*></script>

Spring Boot automatically configures Spring to map requests for **/webjars** to the **/META-INF/resources/webjars** directory of all the JARs in the CLASSPATH.

Things can also work out even without specifying the WebJar (client-side library) versions while including them in HTML view files, like we mentioned in the previous snippet. For this to work, we need to additionally insert the below mentioned **webjar-locator**dependency into the pom file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <!-- webjars-locator -->  <dependency>  <groupId>org.webjars</groupId>  <artifactId>webjars-locator</artifactId>  <version>0.30</version>  </dependency> |

Spring Boot automatically detects the **webjars-locator**library in the classpath and uses it to automatically resolve the version of WebJars we are trying to insert into our HTML page.