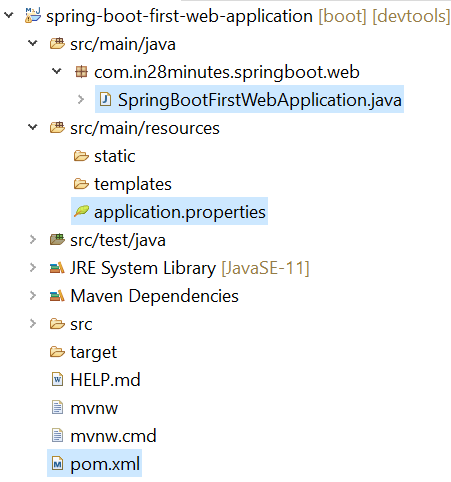
Web Application with Spring Boot

# Project Structure

When you initiate a new spring boot project with *web* and *dev tools* dependencies, or any other dependencies, the following structure is generated.



Let us discuss about each one of them.

## File pom.xml

POM stands for **Project Object Model**. This file contains **maven dependencies**.

The first tag that we look is *parent*. This is just like a parent class from which our project is inheriting some functionalities. The package artifact ID is **spring-boot-starter-parent**. Then we have dependencies, as shown below.

1. **Spring-boot-starter-web** is used for developing web applications.
2. **Spring-boot-starter-devtools** is used for making development tasks easier.
3. **Spring-boot-starter-test** is used for writing test cases.



POM file will also install **transitive dependencies** as well, i.e., dependencies which are required by the dependencies that we’ve specified in pom.xml. The last part is *<build>*.



This tag is used to help build us .jar file.

## File SpringBootFirstWebApplication.java

This file is the entry point (names can differ) to our spring boot project.



There are two things that this **@SpringBootApplication** does.

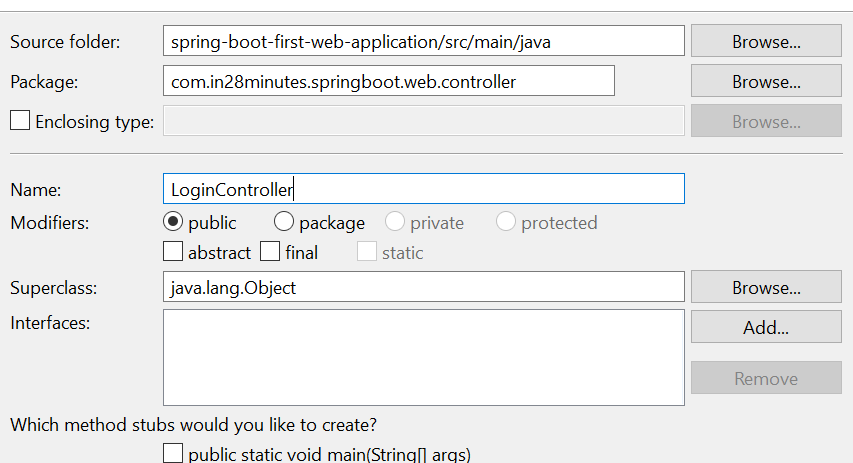
1. It initializes the Spring framework itself by running **@ComponentScan**.
2. It initializes the Spring Boot by running auto-configurations.

## File application.properties

This file is initially empty. This file can be used as a configuration file.

# MVC Controller

The controller is that class in Java to which the URL path is matched to. To define a new controller class, create a new class, and also add a package name **com.in28minutes.springboot.web.controller**.



Now let’s try to display a “Hello World” message using this controller through /login path. Add the following code in this controller.



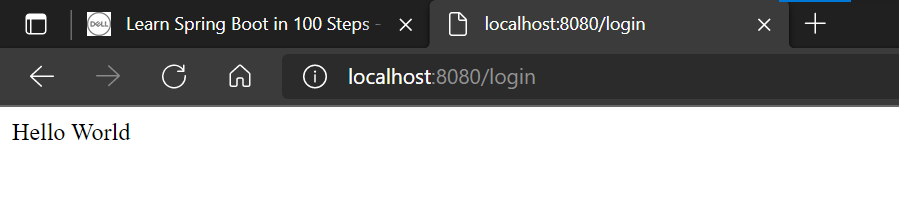
Note that we need **@Controller** for the controller to be defined and **@RequestMapping** for defining the path. However, this code will not work when you will try to access /login. To see the debug logs, set the following property in application.properties.



Now, what is happening is that Spring Boot expects a View to be in the return statement. To fix this issue add a **@ResponseBody** annotation to the controller method.



Now if you go to the browser, you will see the following screen.



# MVC View

To create a view in spring boot, add the following folder – **src/main/webapp/WEB-INF/jsp** and inside this folder our JSP views will go. Now for this to work, add the following in application.properties.



Note that src/main/webapp/ is default, so need to add it in the prefix.

Now, for JSP to work, add the following dependency in pom.xml.



Now, in the controller that we created above, remove @ResponseBody.



Now, this will return whatever there is in **login.jsp** which is located in **src/main/webapp/WEB-INF/jsp/login.jsp**. (Shown below).



# MVC Model – Passing values to View

To make application interactive, let’s pass the named parameters from the URL to the View. Change the controller method to the following.

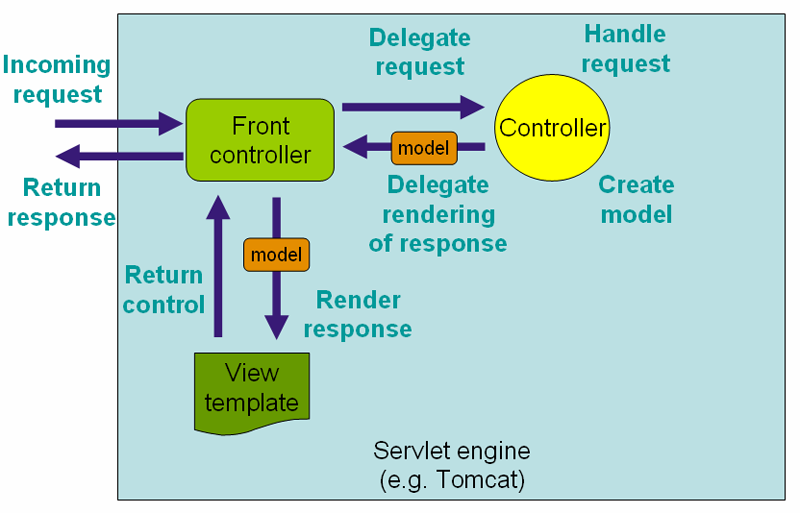


Please see the parameters in the loginMessage() method. **@RequestParam** is used to fetch the named parameter from the URL. It should be exactly same as specified in the URL. **ModelMap** is used to pass values to the view. In the view, we will be using the key *display\_name* to access this value.



To access value from model, use ${} notation.

# MVC Request Flow



When the user sends a request using a URL, it first goes to the Front Controller (**DispatcherServlet**) which forwards that request to the controller. The controller maps the request to a model and gives it back to the front controller. Then front controller sends that model to the view. View implements the model into HTML and sends it back to the front controller. Front controller then sends back the HTML template back to the user.

# Accessing Form Data

Till now we have seen how we pass values using the named parameters using URL. However, sending information like passwords is not a good idea in this approach. Let’s create a login form first. We will simply modify the existing login page only.



Now, we will create a JSP page to create the view for logged in user.



Now, we will create controllers for handling login page and welcome page. Login page will use GET method because using the URL we are just loading the form. Welcome page will display on the POST call when the form details are submitted.



Note that both paths are /login, but one is for handling form as GET and the other is for POST.

## Validating user credentials

Let’s now add a simple service to validate our users. Please add the following code in **LoginService.java** in **com.in28minutes.springboot.web.service**.



Here, we are just hardcoding the credentials to the service only. Now use this service as a dependency in the Login Controller.



Now, just add the following part in the login.jsp.



**Note – If you want to scan a dependency that is not in the directory of the main Java application, what you can do is that add *@ComponentScan(<other-path>)* after @SpringBootApplication.**

# Todos Application

We will now create a new Todos application. Add the following service.



Now, let us add a Todo controller.



Now add the business logic, i.e. model.



If you see the controller, it has hardcoded the user’s string to retrieve the todos. It would be better to maintain the session of logged in user which is put through by model and accessible to all the controllers. To do that, add **@SessionAttributes** annotation to the todo controller.