FPT Software

JAVA SPRING FRAMEWORK

Lab Guides

Document Code	25e-BM/HR/HDCV/FSOFT
Version	1.0
Effective Date	01/09/2024

Hanoi, 08/2024

RECORD OF CHANGES

No	Effective Date	Change Description	Reason	Reviewer	Approver
1	06/08/2024	Create a new Lab	Create new		VinhNV

Contents

CODE:	JSFW_Lab_03_Opt1
TYPE:	SHORT
LOC:	200
DURATION:	120 MINUTES

Java Spring Framework Introduction

Objectives:

- Understand how to use DAO (Data Access Object) pattern with Spring MVC.
- Learn to configure and use Spring MVC for managing entities with database interactions.

Lab Specifications:

In a University Management System, Employee entity will use DAO classes to interact with a PostgreSQL database. Students will learn to implement CRUD operations using Spring MVC and PostgreSQL.

Problem Description:

- Trainees must implement and test methods for managing employees using DAO patterns and PostgreSQL for persistence.

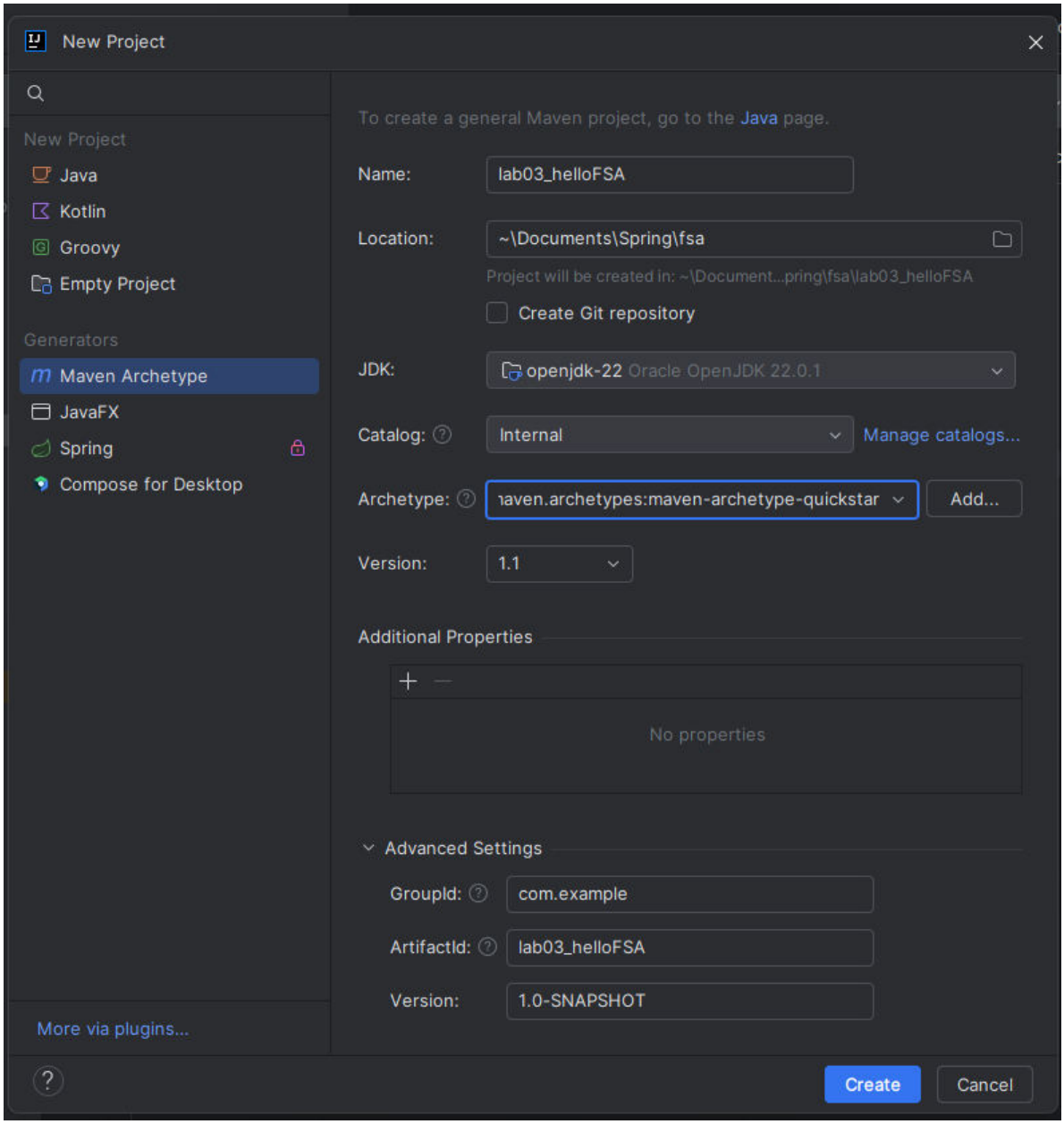
Prerequisites:

- Using Java SDK version 8.0 at least.
- Using Maven.
- Using Spring Framework 5.0 or higher version.

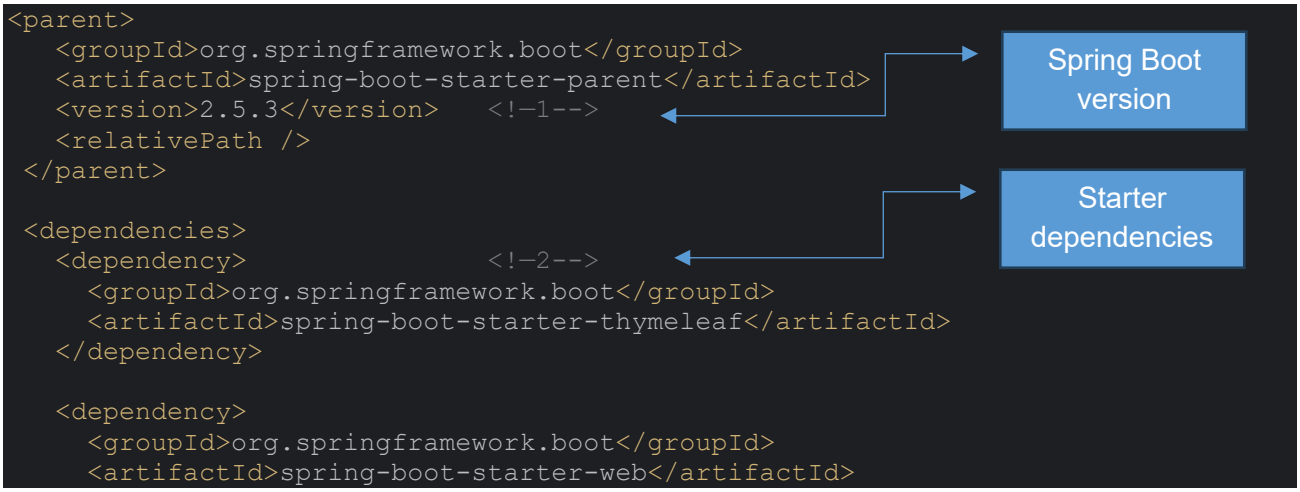
Guidelines:

Step 1: Extend the previous project to include dependency injection:

- Open IntelliJ IDEA.
- Click on File -> New -> Project....
- Select Maven from the project types, and then choose maven-archetype-webapp as the archetype.
- Click Next and set the project name to **lab03_helloFSA**
- Set the groupId to com.example and artifactId to **lab03_helloFSA**
- Click **Create**.



Step 2: Add dependencies and configuration into pom.xml file: Add the Spring Core dependency to your pom.xml file.



```
    </dependency>
</dependencies>
}
```

Step 3: Write a main class:

Because you'll be running the application from an executable JAR, it's important to have a main class that will be executed when that JAR file is run. You'll also need at least a minimal amount of Spring configuration to bootstrap the application. That's what you'll find in the HelloFSAApplication class, shown in the following listing

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // <1>
public class HelloFSAApplication
{
    public static void main( String[] args )
    {
        SpringApplication.run(HelloFSAApplication.class, args); // <2>
    }
}
```

Step 4: Write a controller:

You'll write a simple controller class that handles requests for the root path (for example, /) and forwards those requests to the homepage view without populating any model data. The following listing shows the simple controller class.

```
package com.example;


import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller             // <1>
public class HomeController {

    @GetMapping("/")      // <2>
    public String home() {
        return "home";      // <3>
    }

}
```

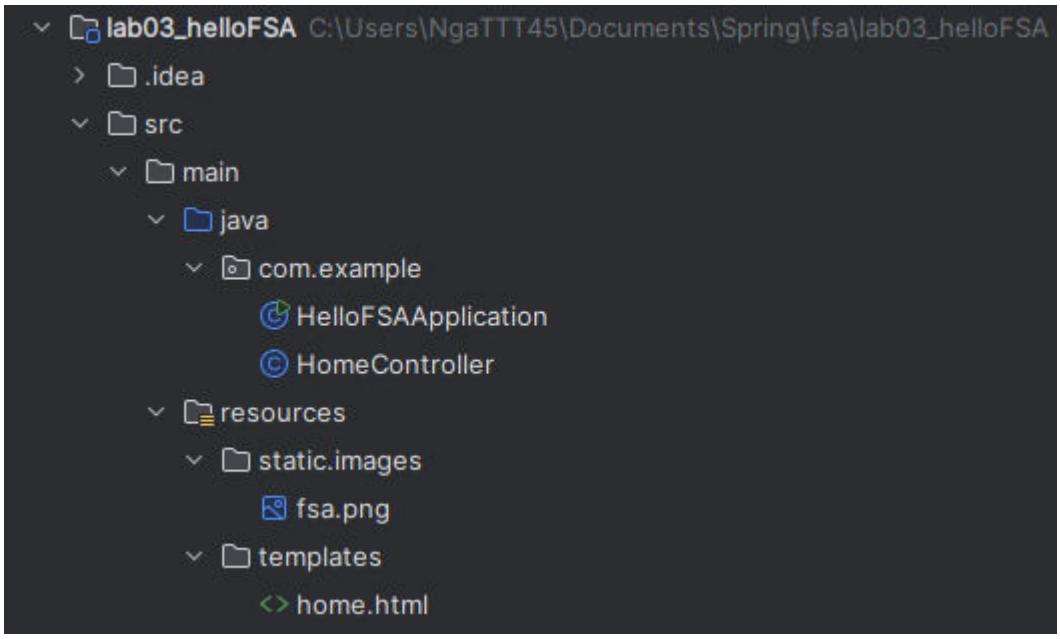

Step 5: Defining a view:

The next listing shows the basic Thymeleaf template that defines the Hello FSA homepage

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>FPT Software Academy</title>
</head>
```

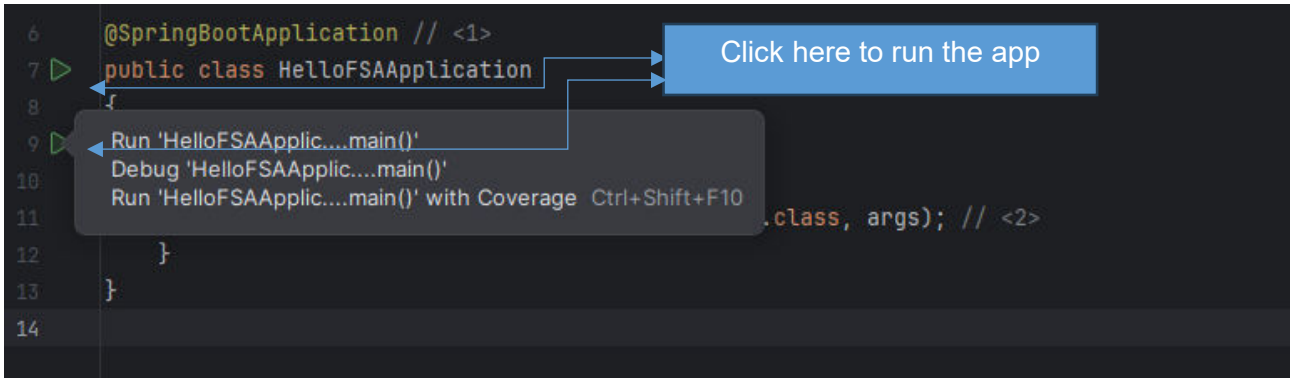
```
<body>
<h1>Welcome to FSA...</h1>
<img th:src="@{/images/fsa.png}"/>
</body>
</html>
```

Here is the structure of the application:

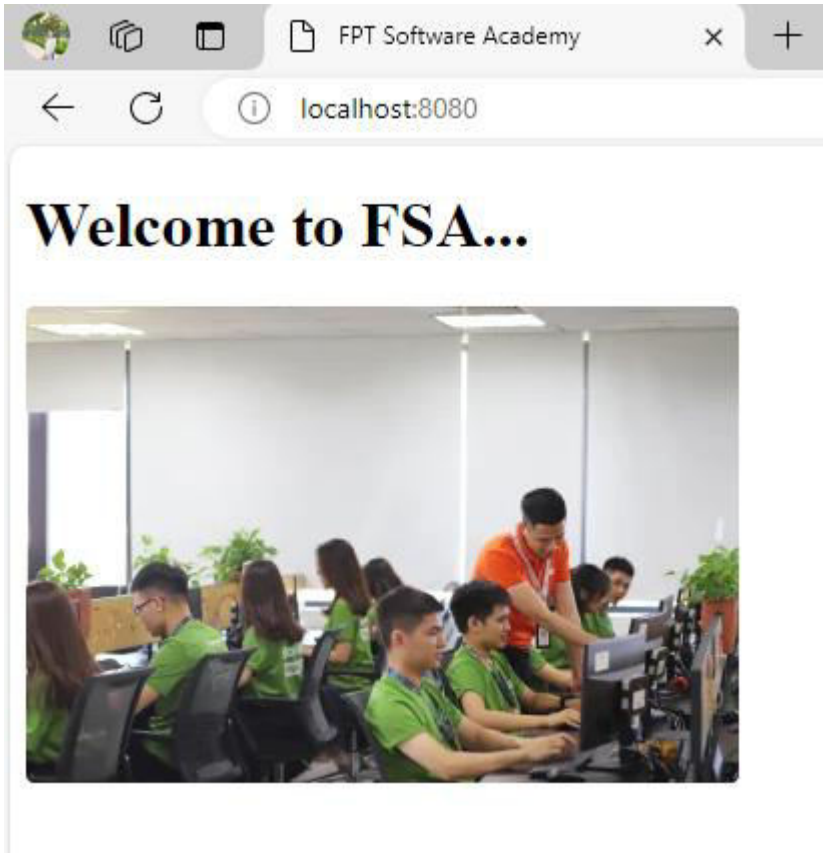


Step 6: Run the application:

In the HelloFSAApplication class, click to the icon in the left (in green color) before [public class…] or [main] method to run the application.

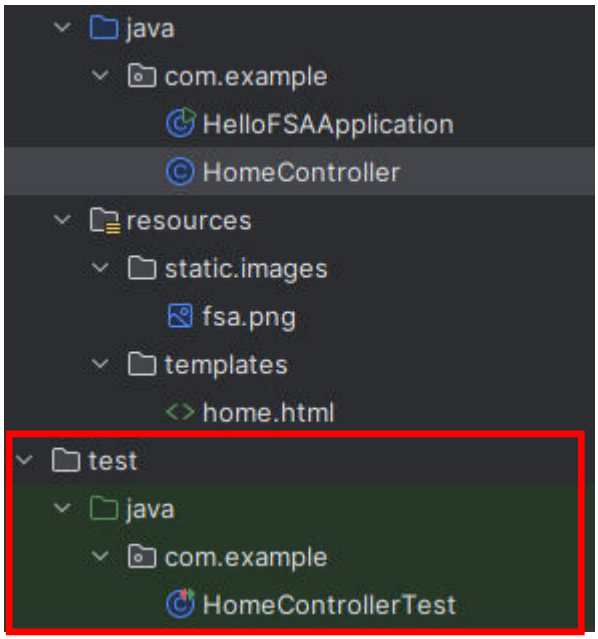


Now that the application has started, point your web browser to http://localhost:8080, you should see something like this:



Step 6: Testing the controller:

Right click on the **HomeController**, click on the [Generate], click on [Test], IntelliJ will create the folder [test] for you.



Add dependencies to the pom.xml file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <exclusions>
```

```
    <exclusion>
      <groupId>org.junit.vintage</groupId>
      <artifactId>junit-vintage-engine</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Here is the HomeControllerTest:

```
package com.example;

import static org.hamcrest.Matchers.containsString;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(HomeController.class)    // <1>
public class HomeControllerTest {

    @Autowired
    private MockMvc mockMvc;    // <2>

    @Test
    public void testHomePage() throws Exception {
        mockMvc.perform(get("/"))     // <3>
                .andExpect(status().isOk())   // <4>
                .andExpect(view().name("home"))   // <5>
                .andExpect(content().string(           // <6>
                        containsString("Welcome to FSA...")));
    }

}
```

Click to the icon on the left side of the file to run test:

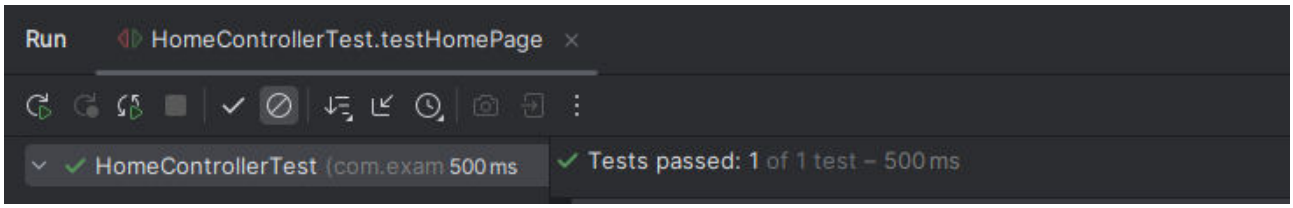
```java
@WebMvcTest(HomeController.class)   // <1>
Run Test Ctrl+Shift+F10 ollerTest {

    @Autowired 1 usage
    private MockMvc mockMvc;   // <2>

    @Test
    public void testHomePage() throws Exception {
        mockMvc.perform(get( urlTemplate: "/"))    // <3>
                .andExpect(status().isOk())  // <4>
                .andExpect(view().name( expectedViewName: "home"))  // <5>
                .andExpect(content().string(           // <6>
                        containsString( substring: "Welcome to FSA...")));
    }

}
```

Here is the result after running test:



This simple Spring Web MVC application demonstrates the basic setup and functionality of a Spring controller, view resolution, and serving static content.

Summary

- **Controller Class (HomeController)**:
    - The HomeController is annotated with @Controller, marking it as a Spring MVC controller.
    - It handles HTTP GET requests to the root URL ("/") using the @GetMapping annotation.
    - The home() method returns a view name "home", which is resolved by Spring's view resolver to the corresponding template file.
- **View (home.html)**:
    - The view is a Thymeleaf template named home.html, which is used to render the response.

- The HTML file contains a welcome message and an image. The image source is dynamically resolved by Thymeleaf using the th:src attribute.

- **Static Content**:
  - The image is referenced using the Thymeleaf expression @{/images/fsa.png}, which resolves to the image file located in the images directory under the static resources of the application.

Application Flow

1. When a user accesses the root URL ("/"), the HomeController handles the request.
2. The home() method returns "home", instructing Spring to render the home.html template.
3. The HTML page is displayed, showing a welcome message and an image.

- This example illustrates the basic components of a Spring MVC application: a controller, a view resolver, and a Thymeleaf template engine.

----oOo-----

THE END