**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Fall 2024, Lab Manual - 11**

|  |  |
| --- | --- |
| **Course Code: SL3001** | **Course: Software Development and construction** |
| **Instructor:** | **Yasir Arfat** |

**Lab # 11**

**What is Spring Boot?**

Spring Boot is a framework built on top of the Spring Framework, designed to simplify the creation and deployment of Java-based web and enterprise applications. It provides pre-configured templates and conventions, which allow developers to quickly set up applications without the need for extensive configuration. By using Spring Boot, developers can focus on building the application's functionality rather than spending time on setup and infrastructure.

**Why Use Spring Boot?**

Spring Boot automates much of the setup required in traditional Spring applications, which makes it ideal for rapid development. With Spring Boot, developers can start coding right away without having to worry about configuring a large number of files. It is especially beneficial for creating REST APIs and web applications, as it offers built-in support for essential features like embedded servers (Tomcat, Jetty), dependency management, and security. This ease of setup and development speeds up the overall workflow, making Spring Boot a popular choice for modern Java applications.

**Differences from Swing**:

Swing and Spring Boot serve different purposes in application development. Swing is a Java library used to build desktop applications that run on a user's local machine, providing a graphical user interface (GUI) with components like buttons, text fields, and tables. On the other hand, Spring Boot is designed for building web applications that can be accessed over a network, typically through a browser. While Swing applications run on the client side, Spring Boot applications run on the server side, handling data and processing requests over the internet or intranets.

**Differences from Maven**:

Maven and Spring Boot work together but serve different roles. Maven is a build automation and dependency management tool, which means it helps in compiling code, managing libraries (dependencies), and packaging applications. Spring Boot, on the other hand, is an application framework that provides a structured way to develop Java applications. Spring Boot leverages Maven to handle dependencies, manage the project structure, and ensure that all required libraries are available for the application to function.

1. **Set Up IntelliJ for Spring Boot Development**
2. Open IntelliJ IDEA and go to File > New > Project.
3. In the New Project window, select Spring Initializr on the left. Spring Initializr helps generate a Spring Boot project with essential configurations.
4. Configure the project:
5. Project SDK: Choose the Java SDK you have installed (e.g., Java 17).
6. Group: Enter a group ID (e.g., com.example).
7. Artifact: Enter an artifact ID (e.g., demo).
8. Click Next.
9. **Select Dependencies**
10. In the Dependencies section, add dependencies based on your project’s needs. For a basic setup, add:
11. Spring Web: To create REST APIs and web applications.
12. Click Next, then Finish to create the project.
13. **Understand the Project Structure**
14. src/main/java: Contains your Java source code.
15. src/main/resources: Holds resources like application.properties for configurations.
16. DemoApplication.java: This is the main class with the @SpringBootApplication annotation, which serves as the entry point for the application.
17. **Code the Application**
18. Open DemoApplication.java and check for the @SpringBootApplication annotation.
19. Create a simple REST controller by adding the following code to a new class named HelloController.java
20. package com.example.springbootlab01;  
      
    import org.springframework.web.bind.annotation.GetMapping;  
    import org.springframework.web.bind.annotation.RestController;  
      
    @RestController  
    public class HelloController {  
      
     @GetMapping("/hello")  
     public String sayHello() {  
     return "Hello, Welcome to the first lab on Spring Boot!";  
     }  
    }
21. Here, @RestController makes this class a web controller, and @GetMapping("/hello") maps HTTP GET requests to the sayHello() method.
22. **Run the Application**

start your Spring Boot application, go to the DemoApplication.java file, right-click, and select Run 'DemoApplication.main()'. When you see 'Started DemoApplication' in the Console, the application is running on http://localhost:8081.

1. **Test the Application**

Open a browser or a tool like Postman, navigate to http://localhost:8081/hello, and you should see the response: Hello, World!

1. **Customize the Application Properties (Optional)**

Open application.properties and add the following to change the port:  
```properties  
server.port=8081  
```  
This sets the application to run on port 8081. Restart to apply changes.

**common Spring Boot annotations**

The most common Spring Annotations are as follows

1. SpringBootApplication
2. @RestController
3. @GetMapping, @PostMapping, etc.
4. @Autowired
5. @RequestParam and @PathVariable

**@SpringBootApplication**

The main entry point for your Spring Boot application.

Combines three annotations:

@Configuration - Marks this class as a source of bean definitions.

@EnableAutoConfiguration - Automatically configures Spring based on the dependencies you have added.

@ComponentScan - Scans the package for Spring components (like controllers, services, etc.).

package com.example.springbootdemo;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class SpringBootDemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringBootDemoApplication.class, args);  
 }  
  
}

**@** **RestController**

* Combines @Controller and @ResponseBody.
* Used to create RESTful web services**.**
* To see Example of Rest Controller Create Class Rest Example
* package com.example.springbootdemo;  
    
  import org.springframework.web.bind.annotation.GetMapping;  
  import org.springframework.web.bind.annotation.RestController;  
    
  @RestController  
  public class RestExample {  
   @GetMapping("/Load")  
   public String sayHello() {  
   return "Working Of Restfull Annotations";  
   }  
  }
* **Access**: Run your application and go to <http://localhost:8080/Load>.

**@GetMapping, @PostMapping, @RequestMapping**

* @GetMapping: Handles GET requests.
* @PostMapping: Handles POST requests.
* @RequestMapping: Can handle all types of requests (GET, POST, etc.).
* Create Api controller.java class

package com.example.springbootdemo;  
  
import org.springframework.web.bind.annotation.\*;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
  
@Controller  
@RequestMapping("/api")  
public class ApiController {  
  
 // Handle GET request to /api (this should load the form)  
 @GetMapping("")  
 public String showApiHome() {  
 return "form"; // Return the form view directly for /api  
 }  
  
 // Render the HTML form  
 @GetMapping("/form")  
 public String showForm() {  
 return "form"; // Render the form view for /api/form  
 }  
  
 // Handle GET request from form submission  
 @GetMapping("/submit")  
 public String handleGetRequest(@RequestParam String name, Model model) {  
 model.addAttribute("message", "Hello, " + name);  
 return "response"; // Return the response view after GET submission  
 }  
  
 // Handle POST request from form submission  
 @PostMapping("/submit")  
 public String handlePostRequest(@RequestParam String data, Model model) {  
 model.addAttribute("message", "Data received: " + data);  
 return "response"; // Return the response view after POST submission  
 }  
}

Now we will add the Html file for form and response in src->main->resources->templates

Form.html code is gave as

<!DOCTYPE html>  
<html>  
<head>  
 <title>Spring Boot Form</title>  
</head>  
<body>  
<h2>Submit Data</h2>  
<form action="/api/submit" method="get">  
 <label for="name">Name (GET request):</label>  
 <input type="text" id="name" name="name" required>  
 <button type="submit">Submit (GET)</button>  
</form>  
<br>  
<form action="/api/submit" method="post">  
 <label for="data">Data (POST request):</label>  
 <input type="text" id="data" name="data" required>  
 <button type="submit">Submit (POST)</button>  
</form>  
</body>  
</html>

Response.html code is given as

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
 <title>Response</title>  
</head>  
<body>  
<h1>Response</h1>  
<p th:text="${message}"></p> <!-- This will display the message -->  
</body>  
</html>

Before you run the application make sure you add dependencies thymeleaf' if you are using maven then use

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-thymeleaf</artifactId>**

**</dependency>**

If you are using build.gradle (as used in this project ) use this line in dependencies

**implementation ‘org.springframework.boot:spring-boot-starter-thymeleaf’**

complete build.gradle file is given as

plugins **{** id 'java'  
 id 'org.springframework.boot' version '3.3.5'  
 id 'io.spring.dependency-management' version '1.1.6'  
**}**group = 'com.example'  
version = '0.0.1-SNAPSHOT'  
  
java **{** toolchain **{** languageVersion = JavaLanguageVersion.*of*(17)  
 **}  
}**repositories **{** mavenCentral()  
**}**dependencies **{** implementation 'org.springframework.boot:spring-boot-starter-web'  
 implementation 'org.springframework.boot:spring-boot-starter-thymeleaf' // Added Thymeleaf starter  
 testImplementation 'org.springframework.boot:spring-boot-starter-test'  
 testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
**}**tasks.named('test') **{** useJUnitPlatform()  
**}**

Now run the Application

http://localhost:8081/api

**@Autowired**

* Used for dependency injection.
* Automatically injects a bean (class instance) into your class.

**Practicle Example**

Create MyService add the following code

package com.example.springbootdemo;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class MyService {  
 public String greet() {  
 return "Hello from MyService!";  
 }  
}

create MyController class and add following code

package com.example.springbootdemo;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class MyController {  
  
 @Autowired  
 private MyService myService;  
  
 @GetMapping("/greet")  
 public String greet() {  
 return myService.greet();  
 }  
}

Here, @Autowired is used to inject the MyService bean into the MyController class. Spring automatically resolves the dependency by searching for a matching bean and injecting it into the controller.

Now Run the Application using <http://localhost:8081/greet>

**@RequestParam and @PathVariable**

* @RequestParam: Binds a query parameter from the URL.
* @PathVariable: Binds a path parameter from the URL.

We'll create a Democontroller class with methods that demonstrate both annotations.

package com.example.springbootdemo;  
  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class DemoController {  
  
 // Demonstrating @RequestParam  
 @GetMapping("/weclome")  
 public String greet(@RequestParam String name) {  
 return "Hello, " + name + "!";  
 }  
  
 // Demonstrating @PathVariable  
 @GetMapping("/weclome/{name}")  
 public String greetByPathVariable(@PathVariable String name) {  
 return "Hello, " + name + "!";  
 }  
  
 // Demonstrating both @RequestParam and @PathVariable together  
 @GetMapping("/weclome/{name}/age")  
 public String greetWithAge(@PathVariable String name, @RequestParam int age) {  
 return "Hello, " + name + "! You are " + age + " years old.";  
 }  
}

**Explanation**

1. **@RequestParam**:
   * The @RequestParam annotation is used to retrieve query parameters from the URL.
   * In this case, @RequestParam String name takes the name query parameter from the URL and returns a greeting.
   * Example URL: http://localhost:8081/weclome?name=John
2. **@PathVariable**:
   * The @PathVariable annotation is used to retrieve path variables from the URL path.
   * In this case, @PathVariable String name extracts the name from the URL path and returns a greeting.
   * Example URL: http://localhost:8080/weclome/John
3. **Both Together**:
   * You can also use @RequestParam and @PathVariable together, where @PathVariable is used for path-based parameters, and @RequestParam is used for query-based parameters.
   * Example URL: <http://localhost:8080/welcome/John/age?age=25>

**Test the Application**

For the @RequestParam example, visit:

http://localhost:8080/welcome?name=John

Output: Hello, John!

For the @PathVariable example, visit:

http://localhost:8080/welcome/John

Output: Hello, John!

For the @RequestParam and @PathVariable example, visit:

http://localhost:8080/greet/welcome/age?age=25

Output: Hello, John! You are 25 years old.