WEEK 9 COURSE CONTENT : Building WEB Application with Spring Boot

TOPICS:

- Building Web Applications with Spring Boot
- Creating Controllers and Handling Requests
- **Request Parameters and Path Variables**
- Explain how to perform request validations in Spring Boot using annotations and validation frameworks
- **Form Handling in Spring Boot**
- Handling Form Submissions and Redirects

LEARNING OUTCOME:

- Building RESTful Web Applications
- Creation and manipulation of data using the controller class
- Differentiate between Request parameters and Path variable
- Handling of form in Spring Boot and redirecting.

DAY 1

Building Web Applications with Spring Boot

Introduction: Web development with Spring Boot is a powerful and popular approach for creating web applications in Java. Spring Boot simplifies the development process by providing a set of conventions and tools that streamline the creation of web applications, making it easier and faster for developers. In this tutorial, we'll explore the basics of web development using Spring Boot, including its benefits, architecture, and components.

Benefits of Using Spring Boot for Web Application Development:

- **Rapid Development:** Spring Boot offers a range of pre-configured templates and tools that significantly speed up the development process, allowing developers to focus on application logic rather than infrastructure.

- **Microservices-Ready:** Spring Boot is well-suited for building microservices-based applications due to its modularity and support for building RESTful APIs.
- **Auto-Configuration:** It provides auto-configuration, which automatically configures application components based on the dependencies you include in your project.
- **Embedded Servers:** Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, making it easy to deploy web applications as standalone executables.
- **Production-Ready:** Spring Boot includes production-ready features like health checks, metrics, and externalized configuration, making it suitable for enterprise-level applications.

Architecture and Components: Spring Boot follows a layered architecture with various components:

- **Controller:** Handles incoming HTTP requests and manages routing to appropriate business logic.
- **Service:** Contains the application's business logic and serves as a bridge between controllers and repositories.
- **Repository:** Manages data storage and retrieval, often interfacing with databases.
- **Model:** Represents the data structures and entities used within the application.
- **View:** In the case of a web application, this represents the user interface (HTML, templates, or front-end frameworks).
- **Spring Boot Application Class:** This class serves as the entry point of the Spring Boot application and contains the **`main`** method.

Now, let's move on to creating controllers and handling requests.

Topic: Creating Controllers and Handling Requests

Role of Controllers in Spring Boot Web Applications: Controllers play a crucial role in Spring Boot web applications by handling incoming HTTP requests and defining how those requests should be processed. They are responsible for processing user input, invoking appropriate business logic, and returning responses.

Creating Controllers Using Annotations: In Spring Boot, you can create controllers easily using annotations. Here's a step-by-step guide:

- **Annotate a Class:** Create a Java class and annotate it with **`@RestController`** or **`@Controller`** to indicate that it's a controller.

```
@RestController
public class MyController {
   // Controller methods will go here
}
```

- **Define Controller Methods:** Inside the controller class, define methods that map to specific HTTP endpoints. Use annotations like **`@GetMapping`**, **`@PostMapping`**, **`@PutMapping`**, or **`@DeleteMapping`** to specify the HTTP methods.

```
@GetMapping("/hello")
public String sayHello() {
   return "Hello, World!";
}
```

Handling Incoming Requests and Defining Routes: Spring Boot uses annotations to map incoming requests to controller methods. You can specify the request path and HTTP method in these annotations.

- **`@GetMapping("/path")`**: Handles GET requests for the specified path.
- **`@PostMapping("/path")`**: Handles POST requests for the specified path.
- **`@PutMapping("/path")`**: Handles PUT requests for the specified path.
- **`@DeleteMapping("/path")`**: Handles DELETE requests for the specified path.

For example, if you have a controller method annotated with **`@GetMapping("/hello")`**, it will handle HTTP GET requests to the "/hello" path.

By following these steps, you can create controllers, define routes, and handle incoming requests effectively in Spring Boot web applications. Controllers serve as the entry point for processing user interactions and directing them to the appropriate parts of your application.

Creating Controllers and Handling Requests

Role of Controllers

What Are Controllers?

- Controllers are responsible for handling incoming HTTP requests.
- They determine what response to send back based on the request.

Controller Annotations

- Spring Boot uses annotations to define controllers.
- Commonly used annotations: **@Controller**, **@RestController**, **@RequestMapping**.

Creating Controllers

@Controller Annotation

- Use **@Controller** to mark a class as a controller.
- Define methods to handle specific request mappings.

```
@Controller
public class HomeController {

    @RequestMapping("/")
    public String home() {
        return "index"; // Returns the "index" template
    }
}
```

@RestController Annotation

- Use **@RestController** for RESTful web services.
- Controller methods return data directly as JSON or XML.

```
@RestController
public class ProductController {

    @GetMapping("/products")
    public List<Product> getAllProducts() {
```

```
        // Fetch and return a list of products
    }
}
```

Day 2

Handling Requests and Defining Routes

Request Parameters with @RequestParam

- Use **@RequestParam** to capture query parameters from the URL.
- Query parameters are usually added to the URL after the **?** symbol.

```
@GetMapping("/search")
public String searchProducts(@RequestParam String keyword) {
    // Use the 'keyword' parameter to search for products
    return "searchResults";
}
```

Path Variables with @PathVariable

- Use **@PathVariable** to capture dynamic values from the URL path.
- Path variables are parts of the URL enclosed in curly braces **{}**.

```
@GetMapping("/products/{id}")
public String getProductById(@PathVariable Long id) {
    // Retrieve and display product details by ID
    return "productDetails";
}
```

Key Differences:

- **@RequestParam** is used for query parameters in the URL, while **@PathVariable** is used for dynamic values in the URL path.
- Query parameters appear after the **?** symbol in the URL (e.g., **/search?keyword=shoes**), while path variables are part of the URL path (e.g., **/products/123**).
- **@RequestParam** is typically used for optional parameters, while **@PathVariable** is used when a value in the URL path is required for the request to be valid.

By understanding the distinction between **@RequestParam** and **@PathVariable**, you can handle various types of request data in your Spring Boot applications effectively. Practice creating controllers, handling different types of requests, and defining routes to become proficient in web application development.

Request Parameters and Path Variables

Different Ways to Pass Data

Request Parameters

- Request parameters are commonly used to pass data to a Spring Boot application.
- They are included in the URL query string after the **?** symbol.

Example URL: **http://example.com/search?query=spring**

Path Variables

- Path variables are parts of the URL path enclosed in curly braces **{}**.
- They are used to capture dynamic values from the URL.

Example URL: **http://example.com/users/123**

Retrieving and Processing Data

Retrieving Request Parameters

```
@GetMapping("/search")
public String search(@RequestParam String query) {
```

```
    // Process the 'query' parameter
    return "searchResults";
}
```

Retrieving Path Variables

```
@GetMapping("/users/{id}")
public String getUserById(@PathVariable Long id) {
    // Process the 'id' path variable
    return "userDetails";
}
```

Request Validations

Importance of Request Validations

Why Validate User Input?

- Validating user input is crucial to ensure data integrity and security.
- It prevents invalid or malicious data from entering the system.
- Proper validation enhances the user experience by providing meaningful error messages.

Performing Request Validations

Using Annotations

- Spring Boot provides validation annotations like **@NotBlank**, **@Min**, **@Max**, etc.
- Annotate fields in your DTO (Data Transfer Object) or model classes.

```
public class User {
    @NotBlank(message = "Username is required")
    private String username;
```

```
    @Min(value = 18, message = "Age must be at least 18")
    private int age;

    // Getters and setters
}
```

Handling Validation Errors

- Use the **BindingResult** object to access validation errors.

```
@PostMapping("/register")
public String registerUser(@Valid @ModelAttribute User user, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        // Handle validation errors
        return "registrationForm";
    }
    // Process the valid user data
    return "registrationSuccess";
}
```

Display error messages in the view using Thymeleaf or another templating engine.

```
<form th:object="${user}" th:action="@{/register}" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" th:field="*{username}" />
```

```
    <span th:if="${#fields.hasErrors('username')}"
th:errors="*{username}"></span>

    <label for="age">Age:</label>
    <input type="text" id="age" name="age" th:field="*{age}" />
    <span th:if="${#fields.hasErrors('age')}" th:errors="*{age}"></span>

    <input type="submit" value="Register" />
</form>
```

Incorporating request parameter and path variable handling, along with request validation, will enable you to create robust and secure Spring Boot web applications. Practice these concepts and exercises to become proficient in web application development and validation.

DAY 3:

Form Handling in Spring Boot

Understanding Form Handling

What is Form Handling?

- Form handling is a fundamental part of web applications.
- It involves creating HTML forms that allow users to input data and submit it to the server.

Why is Form Handling Important?

- Forms enable users to interact with your application by providing input.
- They are used for user registration, login, search, and more.
- Effective form handling is crucial for gathering and processing user data.

Creating Forms and Processing Submissions

Creating HTML Forms

- Use HTML **<form>** elements to create forms.
- Define form fields such as text inputs, checkboxes, and buttons.

Example:

```
<form action="/submit" method="post">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" required>

  <label for="password">Password:</label>
  <input type="password" id="password" name="password" required>

  <input type="submit" value="Submit">
</form>
```

Processing Form Submissions

- In Spring Boot, use controller methods to handle form submissions.
- Use **@PostMapping** to map a method to handle POST requests.

Example:

```
@Controller
public class FormController {

  @PostMapping("/submit")
  public String submitForm(@RequestParam String username, @RequestParam String password) {
    // Process form data
    return "success"; // Redirect to a success page
```

```
    }
}
```

Binding Form Data and Validation

Binding Form Data to Java Objects

- Create a Java class (DTO) with fields matching the form inputs.
- Use **@ModelAttribute** to bind form data to the Java object.

Example:

```
public class User {
    private String username;
    private String password;

    // Getters and setters
}

@Controller
public class FormController {

    @PostMapping("/submit")
    public String submitForm(@ModelAttribute User user) {
        // Process form data using the User object
        return "success"; // Redirect to a success page
    }
}
```

Form Validation

- Use validation annotations like **@NotBlank**, **@Email**, etc., to validate form data.
- Add **@Valid** before the **@ModelAttribute** to enable validation.

Example:

```
public class User {

   @NotBlank(message = "Username is required")

   private String username;


   @NotBlank(message = "Password is required")

   private String password;


   // Getters and setters

}
```

Handling Form Submissions and Redirects

Handling Form Submissions

Handling Form Submissions

- To handle form submissions, use controller methods annotated with **@PostMapping**.
- These methods process the form data submitted by the user.

Example:

```
@Controller

public class FormController {


   @PostMapping("/submit")

   public String submitForm(@ModelAttribute User user) {
```

```
    // Process form data using the User object
    return "success"; // Redirect to a success page
  }
}
```

Understanding Redirects

What are Redirects?

- Redirects are HTTP responses that instruct the browser to navigate to a different URL.
- They are used to send users to a new page after a form submission or other actions.

Redirect in Spring Boot

- Use the **redirect:** prefix to specify the redirect URL in Spring Boot.

Example:

```
@Controller
public class FormController {

  @PostMapping("/submit")
  public String submitForm(@ModelAttribute User user) {
    // Process form data using the User object
    return "redirect:/success"; // Redirect to the success page
  }
}
```

By mastering form handling and redirects in Spring Boot, you'll be able to create interactive web applications that gather and process user data effectively.

Practice creating forms, handling form submissions, and implementing redirects to enhance your web development skills.

DAY 4:

Building Web Applications with Spring Boot

Creating Controllers and Handling Requests

Role of Controllers

Controllers are responsible for handling incoming HTTP requests. They determine what response to send back based on the request.

Controller Annotations

Spring Boot uses annotations to define controllers, with commonly used ones being **@Controller**, **@RestController**, and **@RequestMapping**.

Creating Controllers

Use the **@Controller** annotation to mark a class as a controller. Define methods within the class to handle specific request mappings.

Example:

```
@Controller
public class HomeController {
    @RequestMapping("/")
    public String home() {
        return "index"; // Returns the "index" template
    }
}
```

Handling Requests and Defining Routes

Request Parameters with **@RequestParam**

Use **@RequestParam** to capture query parameters from the URL. Query parameters are usually added to the URL after the **?** symbol.

Example:

```
@GetMapping("/search")
public String searchProducts(@RequestParam String keyword) {
   // Use the 'keyword' parameter to search for products
   return "searchResults";
}
```

Path Variables with **@PathVariable**
Use **@PathVariable** to capture dynamic values from the URL path. Path variables are parts of the URL enclosed in curly braces **{}**.

Example:

```
@GetMapping("/products/{id}")
public String getProductById(@PathVariable Long id) {
   // Retrieve and display product details by ID
   return "productDetails";
}
```

Key Differences:

- **@RequestParam** is used for query parameters in the URL, while **@PathVariable** is used for dynamic values in the URL path.
- Query parameters appear after the **?** symbol in the URL (e.g., **/search?keyword=shoes**), while path variables are part of the URL path (e.g., **/products/123**).
- **@RequestParam** is typically used for optional parameters, while **@PathVariable** is used when a value in the URL path is required for the request to be valid.

Request Validations

Importance of Request Validations

Request validations are important to ensure data integrity and security in web applications. They prevent invalid or malicious data from entering the system and enhance the user experience by providing meaningful error messages.

Performing Request Validations

Using Annotations

Spring Boot provides validation annotations like **@NotBlank**, **@Min**, **@Max**, etc. You can annotate fields in your DTO (Data Transfer Object) or model classes to perform validation.

Example:

```
public class User {
    @NotBlank(message = "Username is required")
    private String username;

    @Min(value = 18, message = "Age must be at least 18")
    private int age;

    // Getters and setters
}
```

Handling Validation Errors

You can use the **BindingResult** object to access validation errors in your controller methods.

Example:

```
@PostMapping("/register")
public String registerUser(@Valid @ModelAttribute User user, BindingResult bindingResult) {
```

```
    if (bindingResult.hasErrors()) {
        // Handle validation errors
        return "registrationForm";
    }
    // Process the valid user data
    return "registrationSuccess";
}
```

This allows you to display error messages in the view using Thymeleaf or another templating engine.

Incorporating request parameter and path variable handling, along with request validation, will enable you to create robust and secure Spring Boot web applications. Practice these concepts and exercises to become proficient in web application development and validation.

External Links:

Spring Framework Documentation: Spring Framework Documentation

Baeldung offers in-depth tutorials and articles on Spring Boot : Baeldung

Spring Boot Tutorials on YouTube: Java Brains