14-Spring MVC and Layers

Introduction to Spring MVC

Spring MVC (Model-View-Controller) is a framework within the Spring ecosystem designed to build web applications. It follows the Model-View-Controller design pattern, which separates an application into three interconnected components, making it easier to manage and develop web applications.

Key Components of Spring MVC

1. Model

- The model represents the application's data and business logic.
- It is responsible for retrieving data from the database, processing it, and passing it to the View for presentation.
- The model encapsulates the application's state and exposes it through getter and setter methods.

2. View

- The View is responsible for rendering the user interface (UI).
- It takes data from the model and displays it to the user.
- In Spring MVC, the view can be a JSP (JavaServer Pages), Thymeleaf template, or other templating engines that support rendering UI components.

3. Controller

- The controller handles user requests and interactions.
- It acts as an intermediary between the model and the view.
- When a user sends a request, the controller processes it, calls the necessary model methods, and then decides which view to render.

How Spring MVC Works

1. **User Request**: The user sends a request to the server (e.g., by clicking a button or submitting a form).
2. **Front Controller (DispatcherServlet)**: Spring MVC uses a special servlet called **DispatcherServlet** as the front controller. It intercepts all incoming requests and routes them to the appropriate controller.
3. **Controller**: The DispatcherServlet forwards the request to the corresponding Controller, which processes the request, interacts with the Model to retrieve or update data, and then returns a ModelAndView object.
4. **ModelAndView**: This object contains both the data (Model) and the logical view name that should be rendered.
5. **View Resolver**: The view resolver maps the logical view name to a specific view technology (like JSP or Thymeleaf). It then renders the view using the data provided by the model.
6. **Response**: The rendered view is sent back to the user as a response.

Advantages of Using Spring MVC

- **Separation of Concerns**: By separating the Model, View, and Controller, Spring MVC ensures that each component handles a specific part of the application, leading to more modular and maintainable code.

- **Flexibility**: Spring MVC is highly flexible, supporting various view technologies (e.g., JSP, Thymeleaf) and integration with other frameworks.
- **Powerful Configuration**: It offers powerful configuration options through both XML and Java annotations, allowing developers to customize the framework to their needs.
- **Extensive Support**: Being part of the Spring ecosystem, Spring MVC benefits from extensive community support, detailed documentation, and a wide range of extensions.

Building a Basic E-Commerce Project with Spring Boot

In this section, we will build a basic e-commerce project using Spring Boot. In previous chapters, we created web projects and endpoint request pages that returned simple text, such as a login page, a home page, and an about page. Although these pages only returned plain text, Spring Boot processed this text as files before sending it back to the client.

Adding frontend components

Later, we will add frontend components such as HTML and CSS to improve the appearance of our project. However, our current focus is solely on the backend.

Project Overview

To gain a deeper understanding of Spring Boot, we will create a basic e-commerce application. The main features of this application will include:

- **Displaying a List of Products**: Users will be able to view a list of products.
- **Search Functionality**: Users will be able to search for specific products.
- **Category Filtering**: Users will be able to sort products by category.
- **Cart Option**: Users will be able to add products to their cart.

For each product, we will handle attributes like the product's price, name, and category.

Data handling and formatting

In this project, all entities (e.g., products) will be treated as objects. The object data will be transferred across the network, but it won't be displayed directly on the user interface (UI). Instead, the data will be formatted into either **JSON** or **XML**. JSON is widely used in the industry due to its readability and ease of use.

It's important to note that enterprise-level projects usually have a large number of classes. These classes are organized into packages based on their functionality. For example, we often use the **MVC pattern**:

- **M (Model)**: Contains all the entity's object or model data.

```java
package com.telusko.SimplewebApp.model;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class Product {

    public Product() {
        // TODO Auto-generated constructor stub
    }

    private int pid;
    private String Pname;
    private long price;

}
```

- **V (View)**: Responsible for the user interface (UI).
- **C (Controller)**: Manages incoming requests, processes data, and returns the appropriate response.

Implementing the Product Controller and Service

1. Creating the Product Controller

- We will create a `ProductController` that handles sending and receiving data between the client and server.
- The controller will include an endpoint like `('/products')` that allows us to access all products.
- This endpoint will provide a list of products, which we'll initially mock with fake data since we are not using a database at this stage.

```java
@RestController
public class ProductController {

    @Autowired
    ProductService service;

    @RequestMapping("/products")
    public List<Product> getProducts() {
        return service.getproducts();
    }}
```

2. Creating the Service Class

- The controller will interact with a service class method called `getProducts`, which returns a list of all products.
- To structure our project properly, we'll add a `@Service` annotation at the class level and place this service class within a `service` package.
- The `@Service` annotation indicates that the business logic resides in this class, which is part of the service layer. It is similar in function to the `@Component` annotation.

```
@Service
public class ProductService {

    List<Product> products = Arrays.asList(
            new Product( prodId: 101, prodName: "Iphone",  price: 50000),
            new Product( prodId: 102,  prodName: "Canon Camera",  price: 70000),
            new Product( prodId: 103,  prodName: "Shure Mic",  price: 10000));

    public List<Product> getProducts(){
        return products;
    }

}
```

Here are the endpoints we've created so far. In the future, we can use UI tools like React or Angular to visualize the data. Additionally, we can use Postman, a well-known tool for API testing, which we will explore further in the next chapters.

Running the Project


```
[
  {
    "prodId": 101,
    "prodName": "Iphone",
    "price": 50000
  },
  {
    "prodId": 102,
    "prodName": "Canon Camera",
    "price": 70000
  },
  {
    "prodId": 103,
    "prodName": "Shure Mic",
    "price": 10000
  }
]
```


"Yeah! We got the output, and we are ready to move on to learning the HTTP methods like GET, POST, PUT, and DELETE, which we will cover in the upcoming chapters. Until then..."

Happy learning & Coding…😊 👾