Controller Layer

December 22, 2024

1 Understanding the Controller Layer and Integration Testing in Spring Boot

1.1 Introduction to the Controller Layer

The *Controller Layer* is a crucial part of a Spring Boot application. It handles HTTP requests from clients and maps them to appropriate services or business logic. By using controllers, we define the *endpoints* of our REST API, enabling clients to perform operations like creating, reading, updating, or deleting resources.

1.1.1 Role of the Controller Layer

- Acts as the *gateway* to the application.
- Transforms client requests into appropriate service calls.
- Handles both incoming data (from request bodies) and outgoing responses (returned to the client).
- Ensures that the application adheres to the principles of RESTful architecture.

1.1.2 Key Annotations in Controllers

- `@RestController`: Marks a class as a controller where every method returns a response body.
- `@RequestMapping`: Maps HTTP requests to specific methods or classes.
- `@PostMapping`, `@GetMapping`, `@PutMapping`, `@DeleteMapping`: Map HTTP methods (POST, GET, PUT, DELETE) to specific operations.
- `@RequestBody`: Binds the body of a client's request to a method parameter.
- `@PathVariable`: Extracts a value from the URI for use in the method.
- `@Autowired`: Injects dependencies like services into the controller.

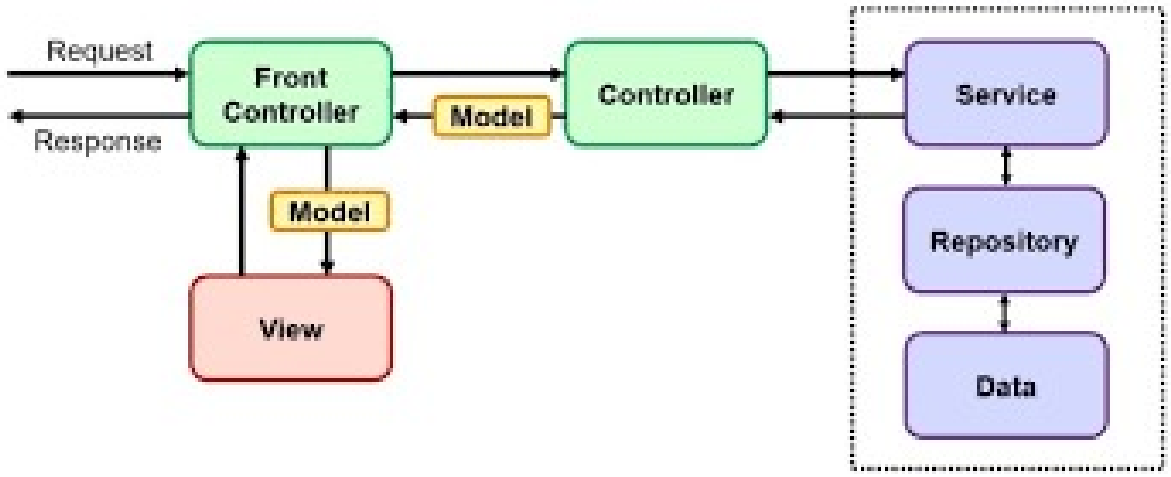

Figure 1: Diagram of the Controller Layer interacting with other components such as Services, Repositories, and Clients.

2 The Controllers We Built

2.1 Author Controller

Purpose: Manages authors with details like `name` and `age`.

- **Endpoints**:
  - **Create**: `POST /api/authors/save` - Adds a new author to the system.
  - **Read**:
    - `GET /api/authors/all`: Retrieves all authors.
    - `GET /api/authors/one/{id}`: Fetches a specific author by ID.
  - **Update**: `PUT /api/authors/update/{id}` - Modifies an author's details.
  - **Delete**: `DELETE /api/authors/delete/{id}` - Removes an author by ID.

Example of Code: Author Controller The following example demonstrates the implementation of a controller layer (Author Controller as example). It contains the annotations and all the features required for this layer.

The complete code is available on the project's GitHub repository:

`https://github.com/DOesNOtM1tter/spring_boot_project-/blob/master/supporting_project/supporting_project/src/main/java/com/example/demo/controllers/AuthorControlle java`.

Explanation:

- The class is annotated with `@RestController`, indicating that it handles HTTP requests and provides responses in JSON format.
- The `@PostMapping("/save")` annotation specifies the endpoint for saving an author.
- `AuthorService` is injected using `@Autowired` to handle business logic.

- The `create()` method takes an `AuthorDto` object, calls the service layer to save it, and returns a response with status `201 CREATED`.

2.2 Book Controller

Purpose: Handles books with attributes like `title`, `author`, `category`, and `publish date`.

- **Endpoints**:
  - **Create**: `POST /api/books/save` - Adds a new book.
  - **Read**:
    - `GET /api/books/all`: Retrieves all books.
    - `GET /api/books/one/{id}`: Fetches a specific book by ID.
  - **Update**: `PUT /api/books/update/{id}` - Updates book details.
  - **Delete**: `DELETE /api/books/delete/{id}` - Deletes a book.

Example of Code: Book Controller Integration Test

The following example demonstrates the implementation of one of the controller layer integration tests. It contains the annotations and all the JUnit and Mockmvc logic required for this layer.

The complete code is available on the project's GitHub repository:

`https://github.com/D0esN0tM1tter/spring_boot_project-/blob/master/supporting_project/supporting_project/src/test/java/com/example/demo/services/BookServicesIntegr` `java`.

2.3 Category Controller

Purpose: Manages categories, which classify books.

- **Endpoints**:
  - **Create**: `POST /api/category/save` - Adds a new category.
  - **Read**:
    - `GET /api/category/all`: Lists all categories.
    - `GET /api/category/one/{id}`: Fetches a specific category by ID.
  - **Update**: `PUT /api/category/update/{id}` - Updates category details.
  - **Delete**: `DELETE /api/category/delete/{id}` - Deletes a category.

3 Integration Testing: MockMvc and JSON-DTO-Entity Transformation

3.1 Why Testing is Important

Testing helps verify that:

- Each controller handles requests as expected.
- Data flows correctly between controllers and services.
- The system responds gracefully to errors or invalid inputs.

3.2 Testing Options and Why We Chose MockMvc

When developing a web application, several testing options are available. For instance, tests can be performed on the server side using technologies such as JPA, Thymeleaf, or by using frontend frameworks like Angular to test the interaction between the client and the server.

3.3 Possible Approaches to Testing

- **Server-side Testing (JPA, Thymeleaf)**:
  - Server-side tests involve testing the business logic in the backend. For example, JPA (Java Persistence API) can be used to test interactions with the database, ensuring that entities are correctly persisted and retrieved.
  - Thymeleaf, integrated with Tomcat, can be used to test the server-side generation of views using dynamic HTML templates. These tests check that data is correctly injected into HTML pages before rendering.
- **Testing with Front-End Frameworks (Angular)**:
  - Another type of test is to test the interactions between the frontend and backend. Frameworks like Angular allow simulating HTTP requests and verifying that the server responds correctly to user actions. These tests not only verify the integration of the frontend with the backend but also the user interface and interactions with the API.

3.4 Why We Chose MockMvc

While other testing methods could be used to test the application end-to-end, we chose to use `MockMvc` for our integration tests. This approach offers several advantages, including:

- **Simplified Backend Testing**: `MockMvc` allows us to simulate HTTP requests directly on the controller without requiring a real server like Tomcat. This lets us test the integration of the controller and service layers without the need for a full server setup or frontend client like Angular.
- **Business Logic Verification**: Using `MockMvc`, we can verify that the business logic, request routing, and data transformation (via DTOs, entities, etc.) work as expected, while avoiding the additional complexities involved with configuring a full web server or frontend framework.
- **Isolation and Focus on the API**: This method enables us to test only the API, isolating the backend without the need for a frontend client. This makes tests faster and more focused, ensuring that we concentrate on the essential server-side components.

In summary, while alternatives such as JPA, Thymeleaf, or tests with Angular exist to test interactions between different components, using `MockMvc` allows us to effectively test the backend in isolation and focus on backend logic, all while simplifying the testing process.

3.5 Role of MockMvc

3.5.1 What is MockMvc?

MockMvc is a Spring tool that allows us to test controllers without starting a server. It simulates HTTP requests and provides responses just like a real client-server interaction.

3.5.2 2. Why MockMvc?

- Avoids the need for a running server.
- Mocks the HTTP request and response process.
- Ideal for integration testing since it ensures that the controller interacts correctly with the service layer and other components.

3.5.3 3. Steps for Testing Using MockMvc:

- Configure MockMvc with `@AutoConfigureMockMvc` in your test class.
- Use MockMvc to simulate HTTP requests (e.g., POST, GET).
- Assert the expected status codes, headers, and response bodies.

3.6 MockMvc Example: BookController Tests

3.6.1 1. Test for Creating a Book

Simulates a POST request to create a new book. Converts a BookDto object to JSON, sends the JSON payload via MockMvc, and verifies the status code (201 Created) and response content (e.g., title, author).

```
@Test
public void testCreateBook() throws Exception {
    AuthorDto author = new AuthorDto("Halasami", 21);
    CategoryDto category = new CategoryDto("Info");
    BookDto book = new BookDto("Java", author, category,
       parseDate("2024-11-20"));

    String bookJson = objectMapper.writeValueAsString(book);

    mockMvc.perform(MockMvcRequestBuilders.post("/api/books/save"
       )
            .contentType(MediaType.APPLICATION_JSON)
            .content(bookJson))
            .andExpect(MockMvcResultMatchers.status().isCreated()
               )
            .andExpect(MockMvcResultMatchers.jsonPath("$.title").
               value("Java"))
            .andExpect(MockMvcResultMatchers.jsonPath("$.author.
               name").value("Halasami"));
}
```

3.6.2 Test for Finding a Book by ID

- Simulates a GET request to fetch a book by ID.
- Verifies the response status (200 OK) and content (e.g., book title, author).

3.6.3 Test for Finding All Books

- Simulates a GET request to fetch all books.
- Verifies the list of books in the response.

3.6.4 MockMvc Integration in Test Class

- `@SpringBootTest`: Loads the entire application context for testing.
- `@AutoConfigureMockMvc`: Enables MockMvc for integration tests.
- `MockMvc.perform`: Executes HTTP requests.
- `ObjectMapper`: Serializes Java objects to JSON and vice versa.

3.7 3.3.2 Why Integration Tests?

Integration tests validate the interaction between:

- **Controller and Service layers**.
- **DTOs, Entities, and JSON**.

4 Utility of JSON-DTO-Entity Transformation in a Layered Project

4.1 JSON-DTO-Entity Transformation

In our layered project, the **JSON-DTO-Entity transformation** is a critical design pattern that ensures clean data flow between different application layers.

- **JSON**:
  - Represents the data exchanged between the client and the server.
  - Used as the standard format for HTTP request and response bodies due to its simplicity and readability.
- **DTO (Data Transfer Object)**:
  - Acts as an intermediary between the client and the internal layers of the application.
  - Enables validation and shaping of incoming and outgoing data without exposing the database structure.
  - Prevents tight coupling between external APIs and internal domain models.
- **Entity**:
  - Represents the actual database structure used for persistence.
  - The Service Layer converts DTOs into Entities for database operations and vice versa.

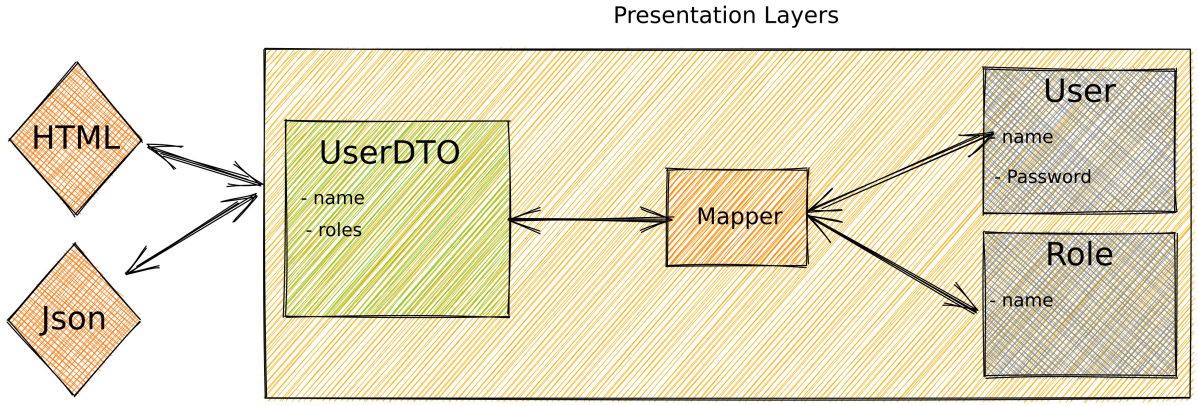

Figure 2: Add a flowchart showing the transformation from JSON → DTO → Entity.

5 Understanding REST Architecture

Representational State Transfer (REST) is an architectural style used for designing networked applications. REST is built around the principles of statelessness, resource-oriented interactions, and the use of standard HTTP methods.

RESTful applications expose their resources through Uniform Resource Identifiers (URIs) and allow interaction with these resources using HTTP verbs such as:

- **GET**: Retrieve a resource or a list of resources.
- **POST**: Create a new resource.
- **PUT**: Update an existing resource.
- **DELETE**: Remove a resource.

A RESTful system communicates using a stateless protocol, typically HTTP, which ensures scalability and simplicity. The data is commonly exchanged in formats like JSON or XML, making it lightweight and compatible with various client types.

REST plays a significant role in the development of modern web applications by enabling flexible and efficient interactions between clients and servers.

5.1 REST in the Project Context

In this project, REST was chosen to design APIs that facilitate communication between the client and server. These APIs manage resources such as user data, tasks, or transactions, following the REST principles. By adopting REST, the application benefits from improved modularity, easier integration with third-party systems, and simplified client-server interactions.

5.2 Testing RESTful APIs with MockMvc

The RESTful APIs implemented in this project were tested using `MockMvc`, a framework provided by Spring for testing web applications. By leveraging `MockMvc`, we ensured that the APIs adhered to REST principles and returned the expected results for all HTTP methods.

The tests focused on:

- Verifying correct implementation of HTTP methods (GET, POST, PUT, DELETE).
- Checking resource accessibility through URIs.
- Validating response formats (e.g., JSON) and HTTP status codes.
- Ensuring proper handling of edge cases, such as invalid inputs or unauthorized access.

This approach ensures the reliability and correctness of the RESTful APIs, which are integral to the application's architecture and functionality.

6 Conclusion

In this report, we explored the key aspects of the Controller Layer in a Spring Boot application, emphasizing its role in handling HTTP requests and serving as the entry point to the application's business logic. By detailing the construction and purpose of various controllers, including the Author, Book, and Category controllers, we demonstrated the practical implementation of REST principles.

The integration of annotations like `@RestController`, `@RequestMapping`, and `@Autowired` enables streamlined development and adherence to modern web standards. Additionally, the incorporation of DTOs ensures a clean separation between client requests and the application's data model.

To validate the functionality of these controllers, we leveraged `MockMvc` for integration testing. This approach allowed us to test API endpoints effectively without the overhead of a full server setup, providing rapid feedback on the correctness of controller logic and its interactions with the service layer.

By combining robust design principles with effective testing strategies, this report highlights a structured methodology for building and validating RESTful APIs in Spring Boot. This knowledge lays a strong foundation for developing scalable and maintainable web applications while ensuring reliable performance through comprehensive testing.