**// SUJATA KUMARI**

**// SUPERSET\_ID - 5004852**

**Exercise 1: Online Bookstore - Setting Up RESTful Services**

**Business Scenario:**

You are tasked with developing a RESTful service for an online bookstore. The service will manage books, authors, and customers.

**Instructions:**

1. **Setup Spring Boot Project:**
   * Initialize a new Spring Boot project named **BookstoreAPI**.

Initialized a new Spring Boot project named BookstoreAPI.

* + Add dependencies: **Spring Web, Spring Boot DevTools, Lombok**.

Added dependencies: Spring Web, Spring Boot DevTools, Lombok.

1. **Project Structure:**
   * Familiarize yourself with the generated project structure.

**src/main/java**: contains the Java source code for the project.

**src/main/resources**: contains configuration files, such as **application.properties.**

**src/test/java**: contains test classes for the project.

**pom.xml**: the Maven project file that defines the project's dependencies and build settings.

1. **What's New in Spring Boot 3:**
   * Explore and document the new features introduced in Spring Boot 3.

**Improved performance**: Spring Boot 3 includes several performance optimizations, such as improved startup time and reduced memory usage.

**Simplified configuration**: Spring Boot 3 introduces a new configuration mechanism that simplifies the process of configuring applications.

**Enhanced security**: Spring Boot 3 includes several security enhancements, such as improved password storage and authentication mechanisms.

**Support for Java 17**: Spring Boot 3 supports Java 17, which includes several new features and improvements.

**Improved support for cloud-native applications**: Spring Boot 3 includes several features that make it easier to build cloud-native applications, such as improved support for containerization and serverless deployment.

**Exercise 2: Online Bookstore - Creating Basic REST Controllers**

**Business Scenario:**

Implement RESTful endpoints to manage books.

**Instructions:**

1. **Create Book Controller:**
   * Define a **BookController** class with request mappings for /books.

Created a new Java class **BookController** in the **com.example.bookstoreapi.controller** package.

package com.example.bookstoreapi.controller;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.bind.annotation.RequestMapping;

@RestController

@RequestMapping("/books")

public class BookController {

}

The **@RestController** annotation indicates that this class is a controller where every method returns a domain object instead of a view. The **@RequestMapping("/books")** annotation maps the **/books** endpoint to this controller.

1. **Handle HTTP Methods:**
   * Implement methods to handle **GET**, **POST**, **PUT**, and **DELETE** requests.

Implemented methods to handle GET, POST, PUT, and DELETE requests.

package com.example.bookstoreapi.controller;

import org.springframework.web.bind.annotation.RestController;

import org.springframework.web.bind.annotation.RequestMapping;

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

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.PutMapping;

import org.springframework.web.bind.annotation.DeleteMapping;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import com.example.bookstoreapi.entity.Book;

@RestController

@RequestMapping("/books")

public class BookController {

@GetMapping

public ResponseEntity<List<Book>> getAllBooks() {

// TO DO: implement logic to retrieve all books

return ResponseEntity.ok(new ArrayList<>());

}

@GetMapping("/{id}")

public ResponseEntity<Book> getBook(@PathVariable Long id) {

// TO DO: implement logic to retrieve a book by ID

return ResponseEntity.ok(new Book());

}

@PostMapping

public ResponseEntity<Book> createBook(@RequestBody Book book) {

// TO DO: implement logic to create a new book

return ResponseEntity.status(HttpStatus.CREATED).body(book);

}

@PutMapping("/{id}")

public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book book) {

// TO DO: implement logic to update a book

return ResponseEntity.ok(book);

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteBook(@PathVariable Long id) {

// TO DO: implement logic to delete a book

return ResponseEntity.noContent().build();

}

}

The above code defines five methods:

* **getAllBooks()**: handles GET requests to retrieve all books
* **getBook()**: handles GET requests to retrieve a book by ID
* **createBook()**: handles POST requests to create a new book
* **updateBook()**: handles PUT requests to update a book
* **deleteBook()**: handles DELETE requests to delete a book

1. **Return JSON Responses:**
   * Ensure the controller returns JSON responses.

The controller methods return JSON responses using the **ResponseEntity** class. The **@RequestBody** annotation is used to convert the JSON request body to a **Book** object.

* + Define the Book entity with attributes like **id, title, author, price**, and **isbn**.

Created a new Java class **Book** in the **com.example.bookstoreapi.entity** package.

package com.example.bookstoreapi.entity;

import lombok.Data;

@Data

public class Book {

private Long id;

private String title;

private String author;

private Double price;

private String isbn;

}

The **@Data** annotation from Lombok is used to generate getters and setters for the **Book** entity.

**Exercise 3: Online Bookstore - Handling Path Variables and Query Parameters**

**Business Scenario:**

Enhance the book management endpoints to handle dynamic URLs and query parameters.

**Instructions:**

1. **Path Variables:**
   * Implement an endpoint to fetch a book by its ID using a path variable.

@GetMapping("/{id}")

public ResponseEntity<Book> getBook(@PathVariable Long id) {

// TO DO: implement logic to retrieve a book by ID

Book book = bookService.getBook(id);

if (book != null) {

return ResponseEntity.ok(book);

} else {

return ResponseEntity.notFound().build();

}

}

In the above code, the **@PathVariable** annotation is used to inject the **id** path variable into the **getBook** method. The **bookService** is a hypothetical service class that retrieves a book by its ID.

1. **Query Parameters:**
   * Implement an endpoint to filter books based on query parameters like title and author.

@GetMapping

public ResponseEntity<List<Book>> getBooks(@RequestParam(required = false) String title, @RequestParam(required = false) String author) {

// TO DO: implement logic to filter books based on title and author

List<Book> books = bookService.getBooks(title, author);

return ResponseEntity.ok(books);

}

In the above code, the **@RequestParam** annotation is used to inject the **title** and **author** query parameters into the **getBooks** method. The **required = false** parameter indicates that these query parameters are optional. The **bookService** is a hypothetical service class that filters books based on the provided title and author.

Example implementation of bookService class:

@Service

public class BookService {

@Autowired

private BookRepository bookRepository;

public Book getBook(Long id) {

return bookRepository.findById(id).orElse(null);

}

public List<Book> getBooks(String title, String author) {

if (title != null && author != null) {

return bookRepository.findByTitleAndAuthor(title, author);

} else if (title != null) {

return bookRepository.findByTitle(title);

} else if (author != null) {

return bookRepository.findByAuthor(author);

} else {

return bookRepository.findAll();

}

}

}

**Exercise 4: Online Bookstore - Processing Request Body and Form Data**

**Business Scenario:**

Create endpoints to accept and process JSON request bodies and form data for customer registrations.

**Instructions:**

1. **Request Body:**
   * Implement a POST endpoint to create a new customer by accepting a JSON request body.

@PostMapping("/customers")

public ResponseEntity<Customer> createCustomer(@RequestBody Customer customer) {

// TO DO: implement logic to create a new customer

Customer createdCustomer = customerService.createCustomer(customer);

return ResponseEntity.status(HttpStatus.CREATED).body(createdCustomer);

}

In the above code, the **@RequestBody** annotation is used to inject the JSON request body into the **createCustomer** method. The **Customer** object is a hypothetical entity class that represents a customer.

Example implementation of customer entity class:

@Data

public class Customer {

private Long id;

private String name;

private String email;

private String password;

}

1. **Form Data:**
   * Implement an endpoint to process form data for customer registrations.

@PostMapping("/customers/register")

public ResponseEntity<Customer> registerCustomer(@ModelAttribute Customer customer) {

// TO DO: implement logic to register a new customer

Customer registeredCustomer = customerService.registerCustomer(customer);

return ResponseEntity.status(HttpStatus.CREATED).body(registeredCustomer);

}

In the above code, the **@ModelAttribute** annotation is used to inject the form data into the **registerCustomer** method. The **Customer** object is the same entity class used in the previous step.

Example implementation of customerService class:

@Service

public class CustomerService {

@Autowired

private CustomerRepository customerRepository;

public Customer createCustomer(Customer customer) {

return customerRepository.save(customer);

}

public Customer registerCustomer(Customer customer) {

// TO DO: implement logic to register a new customer

// For example, you can send a confirmation email or generate a random password

return customerRepository.save(customer);

} }

In the above code, the **CustomerService** class uses a hypothetical **CustomerRepository** class to save the customer data to a database.

**Exercise 5: Online Bookstore - Customizing Response Status and Headers**

**Business Scenario:**

Customize the HTTP response status and headers for the book management endpoints.

**Instructions:**

1. **Response Status:**
   * Use **@ResponseStatus** to customize HTTP status codes for your endpoints.

@GetMapping("/{id}")

@ResponseStatus(HttpStatus.OK)

public Book getBook(@PathVariable Long id) {

// TO DO: implement logic to retrieve a book by ID

Book book = bookService.getBook(id);

return book;

}

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public Book createBook(@RequestBody Book book) {

// TO DO: implement logic to create a new book

Book createdBook = bookService.createBook(book);

return createdBook;

}

@PutMapping("/{id}")

@ResponseStatus(HttpStatus.OK)

public Book updateBook(@PathVariable Long id, @RequestBody Book book) {

// TO DO: implement logic to update a book

Book updatedBook = bookService.updateBook(id, book);

return updatedBook;

}

@DeleteMapping("/{id}")

@ResponseStatus(HttpStatus.NO\_CONTENT)

public void deleteBook(@PathVariable Long id) {

// TO DO: implement logic to delete a book

bookService.deleteBook(id);

}

1. **Custom Headers:**
   * Add custom headers to the response using **ResponseEntity**.

@GetMapping("/{id}")

public ResponseEntity<Book> getBook(@PathVariable Long id) {

// TO DO: implement logic to retrieve a book by ID

Book book = bookService.getBook(id);

return ResponseEntity.ok()

.header("X-Book-Id", String.valueOf(book.getId()))

.header("X-Book-Title", book.getTitle())

.body(book);

}

@PostMapping

public ResponseEntity<Book> createBook(@RequestBody Book book) {

// TO DO: implement logic to create a new book

Book createdBook = bookService.createBook(book);

return ResponseEntity.status(HttpStatus.CREATED)

.header("Location", "/books/" + createdBook.getId())

.body(createdBook);

}

In the above code, the **ResponseEntity** class is used to add custom headers to the response. The **header** method is used to add a single header, and the **headers** method can be used to add multiple header.

**Exercise 6: Online Bookstore - Exception Handling in REST Controllers**

**Business Scenario:**

Implement a global exception handling mechanism for the bookstore RESTful services.

**Instructions:**

1. **Global Exception Handler:**
   * Create a **GlobalExceptionHandler** class using **@ControllerAdvice**.

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<ErrorResponse> handleResourceNotFoundException(ResourceNotFoundException ex) {

ErrorResponse errorResponse = new ErrorResponse("Resource not found", ex.getMessage());

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(errorResponse);

}

@ExceptionHandler(BookAlreadyExistsException.class)

public ResponseEntity<ErrorResponse> handleBookAlreadyExistsException(BookAlreadyExistsException ex) {

ErrorResponse errorResponse = new ErrorResponse("Book already exists", ex.getMessage());

return ResponseEntity.status(HttpStatus.CONFLICT).body(errorResponse);

}

@ExceptionHandler(Exception.class)

public ResponseEntity<ErrorResponse> handleGenericException(Exception ex) {

ErrorResponse errorResponse = new ErrorResponse("Internal Server Error", ex.getMessage());

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body(errorResponse);

}

}

* + Define methods to handle various exceptions and return appropriate HTTP status codes.

In the above code, the **GlobalExceptionHandler** class is annotated with **@ControllerAdvice** to enable global exception handling. The class defines three methods to handle different types of exceptions:

* **handleResourceNotFoundException**: Handles **ResourceNotFoundException** and returns a 404 NOT FOUND response with an error message.
* **handleBookAlreadyExistsException**: Handles **BookAlreadyExistsException** and returns a 409 CONFLICT response with an error message.
* **handleGenericException**: Handles any other type of exception and returns a 500 INTERNAL SERVER ERROR response with an error message.

**Exercise: Online Bookstore - Introduction to Data Transfer Objects (DTOs)**

**Business Scenario:**

Use DTOs to transfer data between the client and server for books and customers.

**Instructions:**

1. **Create DTOs:**
   * Define BookDTO and CustomerDTO classes.

@Data

public class BookDTO {

private Long id;

private String title;

private String author;

private double price;

}

@Data

public class CustomerDTO {

private Long id;

private String name;

private String email;

private String address;

}

1. **Mapping Entities to DTOs:**
   * Use a library like **MapStruct** or **ModelMapper** to map entities to DTOs and vice versa.

Added the following dependencies to your **pom.xml** file (if you're using Maven):

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct</artifactId>

<version>1.4.2.Final</version>

</dependency>

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct-processor</artifactId>

<version>1.4.2.Final</version>

</dependency>

Created a mapper interface for each entity:

@Mapper(componentModel = "spring")

public interface BookMapper {

BookDTO bookToBookDTO(Book book);

Book bookDTOToBook(BookDTO bookDTO);

}

@Mapper(componentModel = "spring")

public interface CustomerMapper {

CustomerDTO customerToCustomerDTO(Customer customer);

Customer customerDTOToCustomer(CustomerDTO customerDTO);

}

MapStruct will generate the implementation for these interfaces at compile-time.

1. **Custom Serialization/Deserialization:**
   * Customize JSON serialization and deserialization using Jackson annotations.

Added the following dependencies to your **pom.xml** file (if you're using Maven):

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.12.3</version>

</dependency>

Use Jackson annotations to customize serialization and deserialization:

@Data

public class BookDTO {

@JsonProperty("book\_id")

private Long id;

@JsonProperty("book\_title")

private String title;

@JsonProperty("book\_author")

private String author;

@JsonProperty("book\_price")

private double price;

}

@Data

public class CustomerDTO {

@JsonProperty("customer\_id")

private Long id;

@JsonProperty("customer\_name")

private String name;

@JsonProperty("customer\_email")

private String email;

@JsonProperty("customer\_address")

private String address;

}