**Week-4 & 5**

**EXERCISE 1:**

#### **1. Setup Spring Boot Project**

**Initialize a New Spring Boot Project:**

* 1. Go to [Spring Initializer](https://start.spring.io/).
  2. Project Name: BookstoreAPI
  3. Choose the following options:
     + Maven Project
     + Language: Java
     + Project:Spring Boot Version**:** 3.x.x (Choose the latest stable version)
     + Packaging**:** Jar
     + Java Version: 17 (or the latest supported by Spring Boot 3)
  4. Add Dependencies:
     + Spring Web: For building web applications, including RESTful services.
     + Spring Boot DevTools: Provides fast application restarts, LiveReload, and configurations for a better development experience.
     + Lombok: A Java library to minimize boilerplate code by providing annotations to generate code like getters, setters, constructors, etc.
  5. Click on Generate to download the project.
  6. Extract the downloaded zip file and open it in your preferred IDE (e.g., IntelliJ IDEA, Eclipse, or VS Code).

#### **2. Project Structure**

**Familiarize Yourself with the Project Structure:**

* + src/main/java: Contains the main application code.
    - com.example.bookstoreapi: The root package for your application.
    - BookstoreApiApplication.java: The main class where the Spring Boot application is started.
  + src/main/resources: Contains configuration files and static resources.
    - application.properties: The main configuration file for your Spring Boot application.
  + src/test/java: Contains test cases for your application.
  + pom.xml: The Maven configuration file, where dependencies and plugins are defined.

#### **3. What's New in Spring Boot 3**

**=Explore and Document New Features in Spring Boot 3:**

* + **Java 17 Support:**
    - Spring Boot 3.x fully supports Java 17, taking advantage of its new language features and performance improvements.
  + **New Baseline:**
    - Spring Boot 3 requires Java 17 as a minimum and Jakarta EE 9. It moves from javax.\* to jakarta.\* namespace.
  + **Native Image Support with GraalVM:**
    - Spring Boot 3 provides first-class support for building native images using GraalVM, enabling faster startup times and reduced memory usage.
  + **Improved Observability:**
    - Enhancements in observability, including better support for Micrometer, which is the default instrumentation library in Spring Boot for monitoring and metrics collection.
  + **Security Enhancements:**
    - Updated Spring Security with support for OAuth 2.1, including better integration with JWT and OAuth2 client/server capabilities.
  + **Auto-Configuration Enhancements:**
    - Improved auto-configuration capabilities with more modular design, allowing more flexibility and customization.
  + **Spring Framework 6.0:**
    - Built on top of Spring Framework 6.0, which includes improvements in core container, new features for reactive programming, and enhanced Kotlin support.
  + **Declarative HTTP Clients:**
    - New support for declarative HTTP clients, making it easier to work with REST APIs.
  + **Native Executables:**
    - Support for creating native executables using GraalVM, which can significantly reduce startup time and memory footprint.

**EXERCISE 2:**

#### **1. Create Book Controller**

Define a BookController Class:

* 1. In your src/main/java/com/example/bookstoreapi package, create a new package named controller.
  2. Inside the controller package, create a new Java class named BookController.

package com.example.bookstoreapi.controller;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/books")

public class BookController {

// Several methods will go here to handle various HTTP requests

}

#### **2. Handle HTTP Methods**

Implement Methods to Handle GET, POST, PUT, and DELETE Requests:

* 1. In the BookControllerclass, implement the methods to handle the different HTTP methods:

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Book;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping("/books")

public class BookController {

private List<Book> bookList = new ArrayList<>();

// GET all books

@GetMapping

public List<Book> getAllBooks() {

return bookList;

}

// GET a book by ID

@GetMapping("/{id}")

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// POST a new book

@PostMapping

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

bookList.add(book);

return new ResponseEntity<>(book, HttpStatus.CREATED)

}

// PUT to update an existing book

@PutMapping("/{id}")

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> {

book.setTitle(updatedBook.getTitle());

book.setAuthor(updatedBook.getAuthor());

book.setPrice(updatedBook.getPrice());

book.setIsbn(updatedBook.getIsbn());

return new ResponseEntity<>(book, HttpStatus.OK);

})

.orElse(ResponseEntity.notFound().build());

}

@DeleteMapping("/{id}")

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

boolean removed = bookList.removeIf(book -> book.getId().equals(id));

return removed ? ResponseEntity.noContent().build() : ResponseEntity.notFound().build();

}

}

#### **3. Return JSON Responses**

Define the Book Entity:

* 1. In your src/main/java/com/example/bookstoreapi package, create a new package named model.
  2. Inside the model package, create a new Java class named Book with attributes id, title, author, price, and isbn.

package com.example.bookstoreapi.model;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Book {

private Long id;

private String title;

private String author;

private double price;

private String isbn;

}

**EXERCISE 3:**

#### **1. Handling Path Variables**

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

**Solution:**

In the BookController class, you will create a method that uses the @PathVariable annotation to map the id from the URL to the method parameter.

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Book;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/books")

public class BookController {

private List<Book> bookList = new ArrayList<>();

// GET all books with optional filtering by title and/or author

@GetMapping

public List<Book> getAllBooks(

@RequestParam(required = false) String title,

@RequestParam(required = false) String author) {

return bookList.stream()

.filter(book -> (title == null || book.getTitle().equalsIgnoreCase(title)) &&

(author == null || book.getAuthor().equalsIgnoreCase(author)))

.collect(Collectors.toList());

}

// GET a book by ID using Path Variable

@GetMapping("/{id}")

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// POST to create a new book

@PostMapping

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

bookList.add(book);

return new ResponseEntity<>(book, HttpStatus.CREATED);

}

// PUT to update an existing book

@PutMapping("/{id}")

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> {

book.setTitle(updatedBook.getTitle());

book.setAuthor(updatedBook.getAuthor());

book.setPrice(updatedBook.getPrice());

book.setIsbn(updatedBook.getIsbn());

return new ResponseEntity<>(book, HttpStatus.OK);

})

.orElse(ResponseEntity.notFound().build());

}

// DELETE a book by ID

@DeleteMapping("/{id}")

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

boolean removed = bookList.removeIf(book -> book.getId().equals(id));

return removed ? ResponseEntity.noContent().build() : ResponseEntity.notFound().build();

}

}

#### **2. Handling Query Parameters**

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

**Solution:**

In the same BookController class, add a method that uses @RequestParam to filter books by optional query parameters.

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Book;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/books")

public class BookController {

private List<Book> bookList = new ArrayList<>();

// GET all books with optional filtering by title and/or author

@GetMapping

public List<Book> getAllBooks(

@RequestParam(required = false) String title,

@RequestParam(required = false) String author) {

return bookList.stream()

.filter(book -> (title == null || book.getTitle().equalsIgnoreCase(title)) &&

(author == null || book.getAuthor().equalsIgnoreCase(author)))

.collect(Collectors.toList());

}

// GET a book by ID using Path Variable

@GetMapping("/{id}")

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(ResponseEntity::ok)

.orElse(ResponseEntity.notFound().build());

}

// POST to create a new book

@PostMapping

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

bookList.add(book);

return new ResponseEntity<>(book, HttpStatus.CREATED);

}

// PUT to update an existing book

@PutMapping("/{id}")

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> {

book.setTitle(updatedBook.getTitle());

book.setAuthor(updatedBook.getAuthor());

book.setPrice(updatedBook.getPrice());

book.setIsbn(updatedBook.getIsbn());

return new ResponseEntity<>(book, HttpStatus.OK);

})

.orElse(ResponseEntity.notFound().build());

}

// DELETE a book by ID

@DeleteMapping("/{id}")

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

boolean removed = bookList.removeIf(book -> book.getId().equals(id));

return removed ? ResponseEntity.noContent().build() : ResponseEntity.notFound().build();

}

}

**EXERCISE 4:**

#### **1. Processing JSON Request Body**

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

**Solution:**

First, create a Customer model:

package com.example.bookstoreapi.model;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Customer {

private Long id;

private String name;

private String email;

private String phoneNumber;

}

Then, implement the POST endpoint in a CustomerController class:

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Customer;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping("/customers")

public class CustomerController {

private List<Customer> customerList = new ArrayList<>();

// POST to create a new customer with JSON request body

@PostMapping

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

customerList.add(customer);

return new ResponseEntity<>(customer, HttpStatus.CREATED);

}

// Other methods can be added here...

}

#### **2. Processing Form Data**

**Objective:** Implement an endpoint to process form data for customer registrations.

**Solution:**

You can handle form data using @RequestParam or @ModelAttribute annotations:

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Customer;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping("/customers")

public class CustomerController {

private List<Customer> customerList = new ArrayList<>();

// POST to create a new customer with form data

@PostMapping("/register")

public ResponseEntity<Customer> registerCustomer(

@RequestParam String name,

@RequestParam String email,

@RequestParam String phoneNumber) {

Customer customer = new Customer(null, name, email, phoneNumber);

customerList.add(customer);

return new ResponseEntity<>(customer, HttpStatus.CREATED);

}

// Other methods can be added here...

}

**EXERCISE 5:**

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

#### **1. Response Status**

You can use the @ResponseStatus annotation to customize HTTP status codes for your endpoints. Here’s how to apply it to your existing BookController methods.

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Book;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/books")

public class BookController {

private List<Book> bookList = new ArrayList<>();

@GetMapping

public List<Book> getAllBooks(

@RequestParam(required = false) String title,

@RequestParam(required = false) String author) {

return bookList.stream()

.filter(book -> (title == null || book.getTitle().equalsIgnoreCase(title)) &&

(author == null || book.getAuthor().equalsIgnoreCase(author)))

.collect(Collectors.toList());

}

@GetMapping("/{id}")

@ResponseStatus(HttpStatus.OK)

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> ResponseEntity.ok().header("Custom-Header", "BookFound").body(book))

.orElse(ResponseEntity.notFound().build());

} @PostMapping

@ResponseStatus(HttpStatus.CREATED)

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

bookList.add(book);

return ResponseEntity.status(HttpStatus.CREATED).header("Custom-Header", "BookCreated").body(book);

}

@PutMapping("/{id}")

@ResponseStatus(HttpStatus.OK)

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

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> {

book.setTitle(updatedBook.getTitle());

book.setAuthor(updatedBook.getAuthor());

book.setPrice(updatedBook.getPrice());

book.setIsbn(updatedBook.getIsbn());

return ResponseEntity.ok().header("Custom-Header", "BookUpdated").body(book);

})

.orElse(ResponseEntity.notFound().build());

}

@DeleteMapping("/{id}")

@ResponseStatus(HttpStatus.NO\_CONTENT)

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

boolean removed = bookList.removeIf(book -> book.getId().equals(id));

return removed ? ResponseEntity.noContent().build() : ResponseEntity.notFound().build();

}

}

**EXERCISE 6:**

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

#### 1. Global Exception Handler

Create a GlobalExceptionHandler class using @ControllerAdvice to handle exceptions globally.

package com.example.bookstoreapi.exception;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

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

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

import org.springframework.web.server.ResponseStatusException;

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(ResponseStatusException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public ResponseEntity<String> handleNotFoundException(ResponseStatusException ex) {

return new ResponseEntity<>(ex.getReason(), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(Exception.class)

@ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

public ResponseEntity<String> handleGenericException(Exception ex) {

return new ResponseEntity<>("An error occurred: " + ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**EXERCISE 7:**

**Objective:** Use DTOs to transfer data between the client and server.

#### 1. Create DTOs

Define BookDTO and CustomerDTO classes.

package com.example.bookstoreapi.dto;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

public class BookDTO {

private Long id;

private String title;

private String author;

private double price;

private String isbn;

}

package com.example.bookstoreapi.dto;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@AllArgsConstructor

public class CustomerDTO {

private Long id;

private String name;

private String email;

private String phoneNumber;

}

#### **2. Mapping Entities to DTOs**

Use a library like ModelMapper or MapStruct. Below is an example using ModelMapper.

Add ModelMapper dependency to pom.xml:

<dependency>

<groupId>org.modelmapper</groupId>

<artifactId>modelmapper</artifactId>

<version>3.1.1</version>

</dependency>

Configure ModelMapper:

package com.example.bookstoreapi.config;

import org.modelmapper.ModelMapper;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration

public class AppConfig {

public ModelMapper modelMapper() {

return new ModelMapper();

}

}

Update BookController to use DTOs:

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.dto.BookDTO;

import com.example.bookstoreapi.model.Book;

import org.modelmapper.ModelMapper;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Collectors;

@RestController

@RequestMapping("/books")

public class BookController {

private List<Book> bookList = new ArrayList<>();

private final ModelMapper modelMapper;

public BookController(ModelMapper modelMapper) {

this.modelMapper = modelMapper;

}

@GetMapping

public List<BookDTO> getAllBooks(

@RequestParam(required = false) String title,

@RequestParam(required = false) String author) {

return bookList.stream()

.filter(book -> (title == null || book.getTitle().equalsIgnoreCase(title)) &&

(author == null || book.getAuthor().equalsIgnoreCase(author)))

.map(book -> modelMapper.map(book, BookDTO.class))

.collect(Collectors.toList());

}

@GetMapping("/{id}")

public ResponseEntity<BookDTO> getBookById(@PathVariable Long id) {

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> ResponseEntity.ok(modelMapper.map(book, BookDTO.class)))

.orElse(ResponseEntity.notFound().build());

}

public ResponseEntity<BookDTO> addBook(@RequestBody BookDTO bookDTO) {

Book book = modelMapper.map(bookDTO, Book.class);

bookList.add(book);

return ResponseEntity.status(HttpStatus.CREATED)

.body(modelMapper.map(book, BookDTO.class));

}

@PutMapping("/{id}")

public ResponseEntity<BookDTO> updateBook(@PathVariable Long id, @RequestBody BookDTO bookDTO) {

return bookList.stream()

.filter(book -> book.getId().equals(id))

.findFirst()

.map(book -> {

book.setTitle(bookDTO.getTitle());

book.setAuthor(bookDTO.getAuthor());

book.setPrice(bookDTO.getPrice());

book.setIsbn(bookDTO.getIsbn());

return ResponseEntity.ok(modelMapper.map(book, BookDTO.class));

})

.orElse(ResponseEntity.notFound().build());

}

@DeleteMapping("/{id}")

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

boolean removed = bookList.removeIf(book -> book.getId().equals(id));

return removed ? ResponseEntity.noContent().build() : ResponseEntity.notFound().build();

}

}