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

**Project Structure:**

* + Familiarize yourself with the generated project structure.

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

mvn archetype:generate -DgroupId=com.example -DartifactId=BookstoreAPI -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

**Add Dependencies**:

* Open the pom.xml file and add the following dependencies for Spring Web, Spring Boot DevTools, and Lombok:

<dependencies>

<!-- Spring Web -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- Spring Boot DevTools -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<!-- Lombok -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<scope>provided</scope>

</dependency>

<!-- Test dependencies -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

**Main Application Class**:

* Create the main application class BookstoreApiApplication.java:

package com.example.bookstoreapi;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class BookstoreApiApplication {

public static void main(String[] args) {

SpringApplication.run(BookstoreApiApplication.class, args);

}

}

**What’s New in Spring Boot 3**

Documenting new features introduced in Spring Boot 3 can be done by writing a short summary or creating a markdown file (SpringBoot3Features.md) within your project:

# What's New in Spring Boot 3

## 1. AOT (Ahead-of-Time) Compilation

Spring Boot 3 introduces AOT compilation, which optimizes the application startup time and memory footprint, especially when building native images with GraalVM.

## 2. Observability Improvements

Enhanced observability with better metrics and tracing support, using Micrometer and Spring Observability.

## 3. Java 17 Baseline

Java 17 is the baseline version, taking advantage of the latest features and improvements in the Java language.

## 4. Updated Dependency Versions

Various dependencies like Spring Framework, Hibernate, and Spring Data have been updated to their latest versions, providing improved performance and new features.

## 5. HTTP Interface Client

A new HTTP interface client allows for creating REST clients with less boilerplate code.

Refer to the official Spring Boot 3 documentation for a complete list of new features and changes.

**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.
2. **Handle HTTP Methods:**
   * Implement methods to handle **GET**, **POST**, **PUT**, and **DELETE** requests.
3. **Return JSON Responses:**
   * Ensure the controller returns JSON responses.
   * Define the Book entity with attributes like **id, title, author, price**, and **isbn**.

**// Create Book Controller**

Create a controller class to manage books.

package com.example.bookstoreapi.controller;

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

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping("/books")

public class BookController {

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

// GET /books - Retrieve all books

@GetMapping

public List<Book> getAllBooks() {

return books;

}

// POST /books - Add a new book

@PostMapping

public Book addBook(@RequestBody Book book) {

books.add(book);

return book;

}

// PUT /books/{id} - Update an existing book

@PutMapping("/{id}")

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

books.set(id, book);

return book;

}

// DELETE /books/{id} - Delete a book

@DeleteMapping("/{id}")

public void deleteBook(@PathVariable int id) {

books.remove(id);

}

}

**// Define Book Entity**

Create the Book entity class.

package com.example.bookstoreapi.controller;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Book {

private int id;

private String title;

private String author;

private double price;

private String isbn;

}

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

**Handling Path Variables**

Add an endpoint to fetch a book by its ID.

// GET /books/{id} - Retrieve a book by ID

@GetMapping("/{id}")

public Book getBookById(@PathVariable int id) {

return books.get(id);

}

**Handling Query Parameters**

Add an endpoint to filter books based on title and author

// GET /books/search - Filter books by title and author

@GetMapping("/search")

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

return books.stream()

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

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

.toList();

}

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

#### Processing Request Body

Create a POST endpoint for customer registration.

package com.example.bookstoreapi.controller;

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

@RestController

@RequestMapping("/customers")

public class CustomerController {

// POST /customers - Register a new customer

@PostMapping

public Customer registerCustomer(@RequestBody Customer customer)

// Add logic to save the customer

return customer;

}

}

**Processing Form Data**

Create an endpoint to handle form data.

// POST /customers/register - Register a new customer with form data

@PostMapping("/register")

public Customer registerCustomerWithFormData(@RequestParam String name, @RequestParam String email) {

Customer customer = new Customer(name, email);

// Add logic to save the customer

return customer;

}

**Customer Entity**

package com.example.bookstoreapi.controller;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Customer {

private String name;

private String email;

}

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

#### Customizing Response Status

Customize HTTP status codes using @ResponseStatus

import org.springframework.http.HttpStatus;

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

// Custom response status for a successful book creation

@ResponseStatus(HttpStatus.CREATED)

@PostMapping

public Book addBook(@RequestBody Book book) {

books.add(book);

return book;

}

**Adding Custom Headers**

Add custom headers to responses.

import org.springframework.http.HttpHeaders;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

@PostMapping

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

books.add(book);

HttpHeaders headers = new HttpHeaders();

headers.add("Custom-Header", "Book Created");

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

}

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

#### Global Exception Handler

Create a GlobalExceptionHandler class.

package com.example.bookstoreapi.exception;

import org.springframework.http.HttpStatus;

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

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

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

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(IndexOutOfBoundsException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public String handleIndexOutOfBoundsException() {

return "Book not found";

}

@ExceptionHandler(Exception.class)

@ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

public String handleGenericException() {

return "Internal Server Error";

}

}

**Exercise 7: 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.
2. **Mapping Entities to DTOs:**
   * Use a library like **MapStruct** or **ModelMapper** to map entities to DTOs and vice versa.
3. **Custom Serialization/Deserialization:**
   * Customize JSON serialization and deserialization using Jackson annotations.

#### Create DTOs

Define BookDTO and CustomerDTO classes.

package com.example.bookstoreapi.dto;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class BookDTO {

private String title;

private String author;

private double price;

private String isbn;

}

@Data

@AllArgsConstructor

@NoArgsConstructor

public class CustomerDTO {

private String name;

private String email;

}

**Mapping Entities to DTOs**

You can use a library like **MapStruct** or **ModelMapper** to map between entities and DTOs. Here’s an example using ModelMapper:

import org.modelmapper.ModelMapper;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class MapperConfig {

@Bean

public ModelMapper modelMapper() {

return new ModelMapper();

}

}

**Custom Serialization/Deserialization**

Customize JSON serialization/deserialization using Jackson annotations.

import com.fasterxml.jackson.annotation.JsonProperty;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class BookDTO {

@JsonProperty("book\_title")

private String title;

@JsonProperty("book\_author")

private String author;

private double price;

@JsonProperty("book\_isbn")

private String isbn;

}