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

**Step 1: Setup Spring Boot Project**

// Initialize a new Spring Boot project named BookstoreAPI

// Add dependencies: Spring Web, Spring Boot DevTools, Lombok

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

#### </dependency> ****Step 2: Project Structure****

* Familiarize yourself with the project structure generated by Spring Boot, which typically includes directories like src/main/java, src/main/resources, and src/test/java.

#### ****Step 3: What's New in Spring Boot 3****

* **New Features in Spring Boot 3:**
  + **AOT Compilation:** Introduces Ahead-Of-Time (AOT) processing, enhancing performance by optimizing application startup.
  + **Spring Native:** Provides support for compiling Spring applications into native executables.
  + **Improved HTTP/2 Support:** Enhanced support for HTTP/2 protocol for better performance in web applications.
  + **Micrometer Metrics:** Updated to support Micrometer 2.x for better observability and monitoring.

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

#### ****Step 1: Create Book Controller****

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.entity.Book;

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<>();

@GetMapping

public List<Book> getAllBooks() {

return bookList;

}

@PostMapping

public void addBook(@RequestBody Book book) {

bookList.add(book);

}

@PutMapping("/{id}")

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

bookList.set(id, book);

}

@DeleteMapping("/{id}")

public void deleteBook(@PathVariable int id) {

bookList.remove(id);

}

#### } Step 2: Handle HTTP Methods

* **GET:** Retrieve all books.
* **POST:** Add a new book.
* **PUT:** Update an existing book by ID.
* **DELETE:** Delete a book by ID.

**Step 3: Return JSON Responses**

package com.example.bookstoreapi.entity;

import lombok.Data;

@Data

public class Book {

private int id;

private String title;

private String author;

private double price;

private String isbn;

### } Ensure the controller returns JSON responses by default, thanks to the @RestController annotation. ****Exercise 3: Online Bookstore - Handling Path Variables and Query Parameters****

#### **Step 1: Path Variables**

#### @GetMapping("/{id}")

#### public Book getBookById(@PathVariable int id) {

#### return bookList.get(id);

#### } ****Step 2: Query Parameters****

#### @GetMapping("/search")

#### public List<Book> searchBooks(@RequestParam String title, @RequestParam String author) {

#### return bookList.stream()

#### .filter(book -> book.getTitle().equalsIgnoreCase(title) && book.getAuthor().equalsIgnoreCase(author))

#### .collect(Collectors.toList());

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

#### ****Step 1: Request Body****

#### @PostMapping("/customers")

#### public void createCustomer(@RequestBody Customer customer) {

#### customerList.add(customer);

#### } ****Step 2: Form Data****

#### @PostMapping("/register")

#### public void registerCustomer(@RequestParam String name, @RequestParam String email) {

#### Customer customer = new Customer(name, email);

#### customerList.add(customer);

### }

### 

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

#### ****Step 1: Response Status****

#### @PostMapping("/books")

#### @ResponseStatus(HttpStatus.CREATED)

#### public void addBook(@RequestBody Book book) {

#### bookList.add(book);

#### } ****Step 2: Custom Headers****

#### @GetMapping("/{id}")

#### public ResponseEntity<Book> getBookWithHeaders(@PathVariable int id) {

#### HttpHeaders headers = new HttpHeaders();

#### headers.add("Custom-Header", "CustomHeaderValue");

#### return new ResponseEntity<>(bookList.get(id), headers, HttpStatus.OK);

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

#### ****Step 1: Global Exception Handler****

#### 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(BookNotFoundException.class)

#### @ResponseStatus(HttpStatus.NOT\_FOUND)

#### public String handleBookNotFound(BookNotFoundException ex) {

#### return ex.getMessage();

#### }

#### @ExceptionHandler(Exception.class)

#### @ResponseStatus(HttpStatus.INTERNAL\_SERVER\_ERROR)

#### public String handleGeneralException(Exception ex) {

#### return "An error occurred: " + ex.getMessage();

#### }

### } ****Exercise 7: Online Bookstore - Introduction to Data Transfer Objects (DTOs)****

#### ****Step 1: Create DTOs****

#### package com.example.bookstoreapi.dto;

#### import lombok.Data;

#### @Data

#### public class BookDTO {

#### private int id;

#### private String title;

#### private String author;

#### private double price;

#### private String isbn;

#### }

#### @Data

#### public class CustomerDTO {

#### private int id;

#### private String name;

#### private String email;

#### } ****Step 2: Mapping Entities to DTOs****

#### import org.modelmapper.ModelMapper;

#### public BookDTO convertToDto(Book book) {

#### ModelMapper modelMapper = new ModelMapper();

#### return modelMapper.map(book, BookDTO.class);

#### } ****Step 3: Custom Serialization/Deserialization****

#### @JsonInclude(JsonInclude.Include.NON\_NULL)

#### public class BookDTO {

#### // Fields

### } ****Exercise 8: Online Bookstore - Implementing CRUD Operations****

#### ****Step 1: CRUD Endpoints****

#### @PostMapping("/books")

#### public void createBook(@RequestBody Book book) {

#### bookList.add(book);

#### }

#### @GetMapping("/books")

#### public List<Book> getBooks() {

#### return bookList;

#### }

#### @PutMapping("/books/{id}")

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

#### bookList.set(id, book);

#### }

#### @DeleteMapping("/books/{id}")

#### public void deleteBook(@PathVariable int id) {

#### bookList.remove(id);

#### } ****Step 2: Validating Input Data****

#### @Data

#### public class Book {

#### @NotNull

#### private int id;

#### @NotBlank

#### @Size(min = 1, max = 255)

#### private String title;

#### @NotBlank

#### private String author;

#### @Min(0)

#### private double price;

#### } ****Step 3: Optimistic Locking****

#### @Version

### private int version; ****Exercise 9: Online Bookstore - Understanding HATEOAS****

#### ****Step 1: Add Links to Resources****

#### @GetMapping("/{id}")

#### public EntityModel<Book> getBookById(@PathVariable int id) {

#### Book book = bookList.get(id);

#### return EntityModel.of(book,

#### linkTo(methodOn(BookController.class).getBookById(id)).withSelfRel(),

#### linkTo(methodOn(BookController.class).getAllBooks()).withRel("books"));

#### } Step 2: Hypermedia-Driven APIs

* Use EntityModel, CollectionModel, and Link classes from Spring HATEOAS to add navigation links.

**Exercise 10: Online Bookstore - Configuring Content Negotiation**

**Step 1: Content Negotiation**

#### @Bean

#### public ContentNegotiationManager contentNegotiationManager(ContentNegotiationConfigurer configurer) {

#### configurer.favorPathExtension(true)

#### .favorParameter(true)

#### .ignoreAcceptHeader(false)

#### .defaultContentType(MediaType.APPLICATION\_JSON)

#### .mediaType("xml", MediaType.APPLICATION\_XML)

#### .mediaType("json", MediaType.APPLICATION\_JSON);

#### return configurer.build();

#### } ****Step 2: Accept Header****

#### @GetMapping(value = "/books/{id}", produces = {MediaType.APPLICATION\_JSON\_VALUE, MediaType.APPLICATION\_XML\_VALUE})

#### public Book getBookById(@PathVariable int id) {

#### return bookList.get(id);

### } ****Exercise 11: Online Bookstore - Integrating Spring Boot Actuator****

#### ****Step 1: Add Actuator Dependency****

#### <dependency>

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

#### <artifactId>spring-boot-starter-actuator</artifactId>

#### </dependency> ****Step 2: Expose Actuator Endpoints****

management:

endpoints:

web:

exposure:

include: health, info, metrics  
  
Expose specific Actuator endpoints like health, info, and metrics by configuring application.yml.

#### ****Step 3: Custom Metrics and Health Indicators****

#### import io.micrometer.core.instrument.MeterRegistry;

#### import org.springframework.stereotype.Component;

#### @Component

#### public class CustomMetrics {

#### public CustomMetrics(MeterRegistry registry) {

#### registry.gauge("custom.metric", Math.random());

#### }

#### }

#### import org.springframework.boot.actuate.health.Health;

#### import org.springframework.boot.actuate.health.HealthIndicator;

#### import org.springframework.stereotype.Component;

#### @Component

#### public class CustomHealthIndicator implements HealthIndicator {

#### @Override

#### public Health health() {

#### return Health.up().withDetail("Custom Health", "All systems operational").build();

#### }

#### }

Define custom metrics and health indicators by implementing MeterRegistry and HealthIndicator interfaces.

### ****Exercise 12: Online Bookstore - Securing Endpoints with Spring Security****

#### ****Step 1: Add Spring Security Dependency****

#### <dependency>

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

#### <artifactId>spring-boot-starter-security</artifactId>

#### </dependency>

#### ****Step 2: Basic Authentication Setup****

#### import org.springframework.context.annotation.Bean;

#### import org.springframework.security.config.annotation.web.builders.HttpSecurity;

#### import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

#### import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

#### import org.springframework.security.core.userdetails.User;

#### import org.springframework.security.core.userdetails.UserDetailsService;

#### import org.springframework.security.provisioning.InMemoryUserDetailsManager;

#### @EnableWebSecurity

#### public class SecurityConfig extends WebSecurityConfigurerAdapter {

#### @Override

#### protected void configure(HttpSecurity http) throws Exception {

#### http

#### .csrf().disable()

#### .authorizeRequests()

#### .antMatchers("/public/\*\*").permitAll()

#### .anyRequest().authenticated()

#### .and()

#### .httpBasic();

#### }

#### @Bean

#### @Override

#### public UserDetailsService userDetailsService() {

#### return new InMemoryUserDetailsManager(

#### User.withDefaultPasswordEncoder()

#### .username("user")

#### .password("password")

#### .roles("USER")

#### .build());

#### }

#### }

* Implement basic authentication with in-memory user details service.

#### ****Step 3: Role-Based Access Control****

#### protected void configure(HttpSecurity http) throws Exception {

#### http

#### .csrf().disable()

#### .authorizeRequests()

#### .antMatchers("/admin/\*\*").hasRole("ADMIN")

#### .antMatchers("/user/\*\*").hasAnyRole("USER", "ADMIN")

#### .antMatchers("/public/\*\*").permitAll()

#### .anyRequest().authenticated()

#### .and()

#### .formLogin();

#### }

**Exercise 13: Online Bookstore - Unit Testing REST Controllers**

**Step 1: JUnit Setup**

#### <!-- pom.xml -->

#### <dependency>

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

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

#### <scope>test</scope>

#### </dependency>

#### <dependency>

#### <groupId>org.mockito</groupId>

#### <artifactId>mockito-core</artifactId>

#### <scope>test</scope>

#### </dependency>

**Step 2: MockMvc for Unit Tests**

#### package com.example.bookstoreapi;

#### import com.example.bookstoreapi.controller.BookController;

#### import com.example.bookstoreapi.service.BookService;

#### import org.junit.jupiter.api.BeforeEach;

#### import org.junit.jupiter.api.Test;

#### import org.mockito.InjectMocks;

#### import org.mockito.Mock;

#### import org.mockito.MockitoAnnotations;

#### import org.springframework.beans.factory.annotation.Autowired;

#### import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

#### import org.springframework.test.web.servlet.MockMvc;

#### import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

#### import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

#### @WebMvcTest(BookController.class)

#### public class BookControllerTest {

#### @Autowired

#### private MockMvc mockMvc;

#### @Mock

#### private BookService bookService;

#### @InjectMocks

#### private BookController bookController;

#### @BeforeEach

#### public void setup() {

#### MockitoAnnotations.openMocks(this);

#### }

#### @Test

#### public void testGetAllBooks() throws Exception {

#### mockMvc.perform(get("/books"))

#### .andExpect(status().isOk())

#### .andExpect(content().json("[]"));

#### }

#### }

**Step 3: Ensure Test Coverage**

* Write additional test cases to cover all CRUD operations and edge cases.
* Ensure each method in the controller is tested using different scenarios.

**Exercise 14: Online Bookstore - Integration Testing for REST Services**

**Step 1: Spring Test Setup**

#### <!-- pom.xml -->

#### <dependency>

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

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

#### <scope>test</scope>

#### </dependency>

#### <dependency>

#### <groupId>com.h2database</groupId>

#### <artifactId>h2</artifactId>

#### <scope>test</scope>

#### </dependency>

**Step 2: MockMvc Integration Testing**

#### package com.example.bookstoreapi;

#### import com.example.bookstoreapi.entity.Book;

#### import com.example.bookstoreapi.repository.BookRepository;

#### import org.junit.jupiter.api.Test;

#### import org.springframework.beans.factory.annotation.Autowired;

#### import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

#### import org.springframework.boot.test.context.SpringBootTest;

#### import org.springframework.test.web.servlet.MockMvc;

#### import org.springframework.transaction.annotation.Transactional;

#### import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

#### import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

#### @SpringBootTest

#### @AutoConfigureMockMvc

#### @Transactional

#### public class BookIntegrationTest {

#### @Autowired

#### private MockMvc mockMvc;

#### @Autowired

#### private BookRepository bookRepository;

#### @Test

#### public void testCreateBook() throws Exception {

#### String newBookJson = "{\"title\":\"Spring in Action\",\"author\":\"Craig Walls\",\"price\":40.0,\"isbn\":\"9781617294945\"}";

#### mockMvc.perform(post("/books")

#### .contentType("application/json")

#### .content(newBookJson))

#### .andExpect(status().isCreated())

#### .andExpect(jsonPath("$.id").isNumber());

#### }

#### }

#### package com.example.bookstoreapi;

#### import com.example.bookstoreapi.entity.Book;

#### import com.example.bookstoreapi.repository.BookRepository;

#### import org.junit.jupiter.api.Test;

#### import org.springframework.beans.factory.annotation.Autowired;

#### import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

#### import org.springframework.boot.test.context.SpringBootTest;

#### import org.springframework.test.web.servlet.MockMvc;

#### import org.springframework.transaction.annotation.Transactional;

#### import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

#### import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

#### @SpringBootTest

#### @AutoConfigureMockMvc

#### @Transactional

#### public class BookIntegrationTest {

#### @Autowired

#### private MockMvc mockMvc;

#### @Autowired

#### private BookRepository bookRepository;

#### @Test

#### public void testCreateBook() throws Exception {

#### String newBookJson = "{\"title\":\"Spring in Action\",\"author\":\"Craig Walls\",\"price\":40.0,\"isbn\":\"9781617294945\"}";

#### mockMvc.perform(post("/books")

#### .contentType("application/json")

#### .content(newBookJson))

#### .andExpect(status().isCreated())

#### .andExpect(jsonPath("$.id").isNumber());

#### }

#### }

**Step 3: Database Integration with H2**

* Configure H2 database for testing in application-test.properties.
* Ensure that tests include database operations like saving and retrieving records.

#### # application-test.properties

#### spring.datasource.url=jdbc:h2:mem:testdb

#### spring.datasource.driverClassName=org.h2.Driver

#### spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

#### spring.h2.console.enabled=true

**Exercise 15: Online Bookstore - API Documentation with Swagger**

**Step 1: Add Swagger Dependency**

#### <!-- pom.xml -->

#### <dependency>

#### <groupId>org.springdoc</groupId>

#### <artifactId>springdoc-openapi-ui</artifactId>

#### <version>1.7.0</version>

#### </dependency>

**Step 2: Annotate Controllers for Documentation**

#### package com.example.bookstoreapi.controller;

#### import com.example.bookstoreapi.entity.Book;

#### import io.swagger.v3.oas.annotations.Operation;

#### import io.swagger.v3.oas.annotations.tags.Tag;

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

#### import java.util.List;

#### @RestController

#### @RequestMapping("/books")

#### @Tag(name = "Books", description = "Operations related to Books")

#### public class BookController {

#### @GetMapping

#### @Operation(summary = "Get all books", description = "Retrieve a list of all books")

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

#### // ...

#### }

#### @PostMapping

#### @Operation(summary = "Create a new book", description = "Add a new book to the store")

#### public Book createBook(@RequestBody Book book) {

#### // ...

#### }

#### }

**Step 3: Generate and Review API Documentation**

* Start the application and navigate to http://localhost:8080/swagger-ui.html to review the generated API documentation.