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

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

**a. Initialize a New Spring Boot Project:**

You can create a new Spring Boot project using Spring Initializr:

* **Website:** Go to [Spring Initializr](https://start.spring.io/)
* **Project Name:** BookstoreAPI
* **Project Options:**
  + Project: Maven Project
  + Language: Java
  + Spring Boot: 3.x.x (latest stable version)
  + Group: com.example
  + Artifact: BookstoreAPI
  + Packaging: Jar
  + Java Version: 17 or the latest LTS

**b. Add Dependencies:**

Add the following dependencies:

* **Spring Web:** For building RESTful web services using Spring MVC.
* **Spring Boot DevTools:** Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
* **Lombok:** Reduces boilerplate code for model classes by using annotations.

After selecting these dependencies, click on the **Generate** button to download the project.

**c. Import the Project into Your IDE:**

* Extract the downloaded project and open it in your favorite IDE (e.g., IntelliJ IDEA, Eclipse, VSCode).

### **2. Project Structure**

Familiarize yourself with the generated project structure:

* **src/main/java**: Contains the application code.
  + **com.example.bookstoreapi**: The main package where all your classes will reside.
    - **BookstoreApiApplication.java**: The main entry point of the Spring Boot application.
* **src/main/resources**: Contains configuration files.
  + **application.properties**: Configuration file for the application.
* **src/test/java**: Contains test cases.
* **pom.xml**: Project Object Model file for managing dependencies.

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

Spring Boot 3 introduces several new features and improvements, including:

* **Jakarta EE 9 Migration:** Spring Boot 3 supports Jakarta EE 9, where package names have been changed from javax.\* to jakarta.\*. This migration is crucial for compatibility with future versions of the Jakarta EE platform.
* **Enhanced AOT Processing:** Spring Boot 3 includes Ahead-of-Time (AOT) processing to optimize application startup time and reduce memory consumption, especially in native images.
* **Improved Observability:** New features for tracing, metrics, and logging have been added, making it easier to monitor and observe applications.
* **Enhanced Security Configurations:** Spring Boot 3 comes with improved security defaults and configurations, particularly around Spring Security, to help developers secure their applications more easily.
* **Kotlin Support:** Kotlin support has been further enhanced with updated APIs and improved compatibility with Spring Framework 6.
* **Native Executable Support:** With the growing trend towards native applications, Spring Boot 3 offers support for building native executables using GraalVM.
* **Updated Dependency Versions:** Most dependencies in Spring Boot 3 have been updated to their latest stable versions, improving performance, security, and compatibility.

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

### **1. Create BookController Class**

Define a BookController class that will manage all the HTTP requests for books.

**BookController.java:**

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Book;

import com.example.bookstoreapi.repository.BookRepository;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/api/books")

public class BookController {

@Autowired

private BookRepository bookRepository;

// GET: Retrieve all books

@GetMapping

public List<Book> getAllBooks() {

return bookRepository.findAll();

}

// GET: Retrieve a book by ID

@GetMapping("/{id}")

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

Optional<Book> book = bookRepository.findById(id);

return book.map(ResponseEntity::ok)

.orElse(new ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

// POST: Create a new book

@PostMapping

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

Book savedBook = bookRepository.save(book);

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

}

// PUT: Update an existing book by ID

@PutMapping("/{id}")

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

return bookRepository.findById(id)

.map(book -> {

book.setTitle(bookDetails.getTitle());

book.setAuthor(bookDetails.getAuthor());

book.setPrice(bookDetails.getPrice());

book.setIsbn(bookDetails.getIsbn());

Book updatedBook = bookRepository.save(book);

return ResponseEntity.ok(updatedBook);

})

.orElse(new ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

// DELETE: Delete a book by ID

@DeleteMapping("/{id}")

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

return bookRepository.findById(id)

.map(book -> {

bookRepository.delete(book);

return new ResponseEntity<Void>(HttpStatus.NO\_CONTENT);

})

.orElse(new ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

}

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

The BookController class handles the following HTTP methods:

* **GET** /api/books: Retrieves a list of all books.
* **GET** /api/books/{id}: Retrieves a specific book by its ID.
* **POST** /api/books: Creates a new book.
* **PUT** /api/books/{id}: Updates an existing book by its ID.
* **DELETE** /api/books/{id}: Deletes a book by its ID.

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

Spring Boot automatically returns responses in JSON format when you use the @RestController annotation and include methods like @GetMapping, @PostMapping, etc. The Book entity is automatically serialized to JSON by Spring’s built-in Jackson library.

### **4. Define the Book Entity**

Here’s how to define the Book entity with attributes like id, title, author, price, and isbn.

package com.example.bookstoreapi.model;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Data

@NoArgsConstructor

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

private double price;

private String isbn;

public Book(String title, String author, double price, String isbn) {

this.title = title;

this.author = author;

this.price = price;

this.isbn = isbn;

}

}

### **Summary**

* **BookController**: Manages HTTP requests and maps them to corresponding methods.
* **Book Entity**: Represents a book with attributes id, title, author, price, and isbn.
* **Response in JSON**: Spring Boot handles JSON serialization and deserialization automatically for you.

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

### **1. Path Variables: Fetch a Book by Its ID**

To fetch a book by its ID using a path variable, you can use the @PathVariable annotation in the controller.

@GetMapping("/{id}")

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

Optional<Book> book = bookRepository.findById(id);

return book.map(ResponseEntity::ok)

.orElse(new ResponseEntity<>(HttpStatus.NOT\_FOUND));

}

* URL: /api/books/{id}
* Method: GET
* Example: GET /api/books/1 would fetch the book with ID 1.

### **2. Query Parameters: Filter Books by Title and Author**

To filter books based on query parameters like title and author, you can use the @RequestParam annotation in the controller.

@GetMapping("/search")

public ResponseEntity<List<Book>> searchBooks(

@RequestParam(required = false) String title,

@RequestParam(required = false) String author) {

List<Book> books;

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

books = bookRepository.findByTitleContainingAndAuthorContaining(title, author);

} else if (title != null) {

books = bookRepository.findByTitleContaining(title);

} else if (author != null) {

books = bookRepository.findByAuthorContaining(author);

} else {

books = bookRepository.findAll();

}

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

}

* URL: /api/books/search
* Method: GET
* Example:
  + GET /api/books/search?title=Spring would fetch books with "Spring" in the title.
  + GET /api/books/search?author=John would fetch books written by "John."
  + GET /api/books/search?title=Spring&author=John would fetch books with "Spring" in the title and written by "John."

### **3. Update the BookRepository for Custom Queries**

To support the query parameters, you need to define custom methods in your BookRepository.

package com.example.bookstoreapi.repository;

import com.example.bookstoreapi.model.Book;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface BookRepository extends JpaRepository<Book, Long> {

List<Book> findByTitleContaining(String title);

List<Book> findByAuthorContaining(String author);

List<Book> findByTitleContainingAndAuthorContaining(String title, String author);

}

### **Summary**

* Path Variables: Implemented to fetch a book by its ID using @PathVariable.
* Query Parameters: Implemented to filter books by title and author using @RequestParam.

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

### **1. Request Body: Implement a POST Endpoint to Create a New Customer**

To accept a JSON request body for creating a new customer, use the @RequestBody annotation in the controller.

**1.Create the Customer Entity:**

package com.example.bookstoreapi.model;

import lombok.Data;

import lombok.NoArgsConstructor;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

@Data

@NoArgsConstructor

@Entity

public class Customer {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private String password;

public Customer(String name, String email, String password) {

this.name = name;

this.email = email;

this.password = password;

}

}

**2. Create the CustomerRepository:**

package com.example.bookstoreapi.repository;

import com.example.bookstoreapi.model.Customer;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Long> {

}

**3.Create the CustomerController:**

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Customer;

import com.example.bookstoreapi.repository.CustomerRepository;

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

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

@RestController

@RequestMapping("/api/customers")

public class CustomerController {

@Autowired

private CustomerRepository customerRepository;

// POST: Create a new customer with JSON request body

@PostMapping

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

Customer savedCustomer = customerRepository.save(customer);

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

}

}

* URL: /api/customers
* Method: POST
* Request Body: JSON
* Example Request:

{

"name": "John Doe",

"email": "john.doe@example.com",

"password": "securepassword"

}

This endpoint accepts a JSON request body and creates a new customer in the database.

### **2. Form Data: Implement an Endpoint to Process Form Data for Customer Registrations**

To process form data, you can use the @RequestParam or @ModelAttribute annotations in the controller.

1. Process Form Data Using @RequestParam:

@PostMapping("/register")

public ResponseEntity<Customer> registerCustomer(

@RequestParam String name,

@RequestParam String email,

@RequestParam String password) {

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

Customer savedCustomer = customerRepository.save(customer);

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

}

* URL: /api/customers/register
* Method: POST
* Request Body: Form Data
* Example Form Data:
  + name=John Doe
  + email=john.doe@example.com
  + password=securepassword

This endpoint accepts form data and registers a new customer.

**2. Alternatively, Using @ModelAttribute:**

You can also use @ModelAttribute to bind form data directly to a model object:

@PostMapping("/register")

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

Customer savedCustomer = customerRepository.save(customer);

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

}

With @ModelAttribute, the form fields must match the attributes in the Customer entity.

### **Summary**

* Request Body: Implemented a POST endpoint to create a new customer by accepting a JSON request body.
* Form Data: Implemented an endpoint to process form data for customer registrations.

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

### **1. Response Status: Customizing HTTP Status Codes with @ResponseStatus:**

The @ResponseStatus annotation allows you to specify the HTTP status code that should be returned for a particular endpoint.

#### Example: Customizing Status for Creating a Book

Let's customize the HTTP status code for the endpoint that creates a new book.

**1. Create the BookController:**

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.model.Book;

import com.example.bookstoreapi.repository.BookRepository;

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

import org.springframework.http.HttpStatus;

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

@RestController

@RequestMapping("/api/books")

public class BookController {

@Autowired

private BookRepository bookRepository;

// POST: Create a new book

@PostMapping

@ResponseStatus(HttpStatus.CREATED) // Set HTTP status to 201 Created

public Book createBook(@RequestBody Book book) {

return bookRepository.save(book);

}

// Other methods...

}

### **2. Custom Headers: Adding Custom Headers to the Response**

To add custom headers to the response, you can use the ResponseEntity class, which allows you to set headers, status, and the body of the response.

#### Example: Adding Custom Headers When Fetching a Book

**1.Fetching a Book with Custom Headers:**

@GetMapping("/{id}")

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

Optional<Book> book = bookRepository.findById(id);

if (book.isPresent()) {

HttpHeaders headers = new HttpHeaders();

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

headers.add("Another-Header", "AnotherHeaderValue");

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

} else {

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

}

}

* Custom Headers:
  + Custom-Header: CustomHeaderValue
  + Another-Header: AnotherHeaderValue

### **Summary:**

* Response Status: Used @ResponseStatus to customize HTTP status codes, such as 201 Created for a successful creation.
* Custom Headers: Added custom headers to the response using ResponseEntity.

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

### **1. Global Exception Handler: Creating a GlobalExceptionHandler Class**

#### **Step 1: Create the GlobalExceptionHandler Class**

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.context.request.WebRequest;

import java.time.LocalDateTime;

import java.util.HashMap;

import java.util.Map;

@ControllerAdvice

public class GlobalExceptionHandler {

// Handle specific exception (e.g., Resource Not Found)

@ExceptionHandler(ResourceNotFoundException.class)

public ResponseEntity<Map<String, Object>> handleResourceNotFoundException(

ResourceNotFoundException ex, WebRequest request) {

Map<String, Object> body = new HashMap<>();

body.put("timestamp", LocalDateTime.now());

body.put("status", HttpStatus.NOT\_FOUND.value());

body.put("error", "Resource Not Found");

body.put("message", ex.getMessage());

body.put("path", request.getDescription(false));

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

}

// Handle generic exceptions

@ExceptionHandler(Exception.class)

public ResponseEntity<Map<String, Object>> handleGlobalException(

Exception ex, WebRequest request) {

Map<String, Object> body = new HashMap<>();

body.put("timestamp", LocalDateTime.now());

body.put("status", HttpStatus.INTERNAL\_SERVER\_ERROR.value());

body.put("error", "Internal Server Error");

body.put("message", ex.getMessage());

body.put("path", request.getDescription(false));

return new ResponseEntity<>(body, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

// Add more exception handlers as needed

}

#### **Step 2: Define Custom Exceptions (Optional)**

If you need custom exceptions, you can define them. For example, a ResourceNotFoundException:

**1. Create the ResourceNotFoundException class:**

package com.example.bookstoreapi.exception;

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

### **2. Using the Global Exception Handler in Controllers**

In your controllers, throw exceptions when something goes wrong (e.g., when a resource is not found):

@GetMapping("/{id}")

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

Book book = bookRepository.findById(id)

.orElseThrow(() -> new ResourceNotFoundException("Book not found with id: " + id));

return ResponseEntity.ok(book);

}

**7. Online Bookstore - Introduction to Data Transfer Objects (DTOs)**

### **1. Create DTOs: Define BookDTO and CustomerDTO Classes**

DTOs (Data Transfer Objects) are used to transfer data between the client and server while keeping the data format consistent and optimized.

#### **Step 1: Define BookDTO Class**

package com.example.bookstoreapi.dto;

import lombok.Data;

@Data

public class BookDTO {

private Long id;

private String title;

private String author;

private double price;

private String isbn;

}

#### **Step 2: Define CustomerDTO Class**

package com.example.bookstoreapi.dto;

import lombok.Data;

@Data

public class CustomerDTO {

private Long id;

private String name;

private String email;

private String address;

}

* **Lombok's @Data** annotation is used to automatically generate getters, setters, toString(), equals(), and hashCode() methods.

### **2. Mapping Entities to DTOs: Use MapStruct or ModelMapper**

MapStruct and ModelMapper are popular libraries used to map entities to DTOs and vice versa.

#### **Using MapStruct**

MapStruct generates the mapping code during compilation.

**1. Add the MapStruct dependency** in your pom.xml:

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct</artifactId>

<version>1.5.3.Final</version>

</dependency>

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct-processor</artifactId>

<version>1.5.3.Final</version>

</dependency>

**2. Create a Mapper Interface for Book and Customer:**

package com.example.bookstoreapi.mapper;

import com.example.bookstoreapi.dto.BookDTO;

import com.example.bookstoreapi.dto.CustomerDTO;

import com.example.bookstoreapi.model.Book;

import com.example.bookstoreapi.model.Customer;

import org.mapstruct.Mapper;

import org.mapstruct.factory.Mappers;

@Mapper

public interface BookstoreMapper {

BookstoreMapper INSTANCE = Mappers.getMapper(BookstoreMapper.class);

// Book mapping

BookDTO bookToBookDTO(Book book);

Book bookDTOToBook(BookDTO bookDTO);

// Customer mapping

CustomerDTO customerToCustomerDTO(Customer customer);

Customer customerDTOToCustomer(CustomerDTO customerDTO);

}

3. **Use the Mapper in Your Service or Controller:**

import com.example.bookstoreapi.mapper.BookstoreMapper;

BookDTO bookDTO = BookstoreMapper.INSTANCE.bookToBookDTO(book);

#### **Using ModelMapper**

ModelMapper is another option if you prefer dynamic mapping.

**1. Add the ModelMapper dependency** in pom.xml:

<dependency>

<groupId>org.modelmapper</groupId>

<artifactId>modelmapper</artifactId>

<version>3.1.0</version>

</dependency>

**2. Use ModelMapper to Map Entities to DTOs:**

import org.modelmapper.ModelMapper;

ModelMapper modelMapper = new ModelMapper();

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

CustomerDTO customerDTO = modelMapper.map(customer, CustomerDTO.class);

### **3. Custom Serialization/Deserialization: Customize JSON Using Jackson Annotations**

Jackson allows you to customize how JSON is serialized and deserialized.

#### **Example: Customize BookDTO Serialization**

**1. Add @JsonProperty to customize field names:**

package com.example.bookstoreapi.dto;

import com.fasterxml.jackson.annotation.JsonProperty;

import lombok.Data;

@Data

public class BookDTO {

@JsonProperty("book\_id")

private Long id;

@JsonProperty("book\_title")

private String title;

private String author;

@JsonProperty("book\_price")

private double price;

@JsonProperty("book\_isbn")

private String isbn;

}

2. **Exclude Fields from Serialization with @JsonIgnore:**

import com.fasterxml.jackson.annotation.JsonIgnore;

@Data

public class CustomerDTO {

private Long id;

private String name;

@JsonIgnore

private String email;

private String address;

}

### **Summary**

* **DTOs**: Created BookDTO and CustomerDTO classes to transfer data between the client and server.
* **Mapping**: Used MapStruct or ModelMapper to map entities to DTOs and vice versa.
* **Custom Serialization/Deserialization**: Used Jackson annotations to customize JSON serialization and deserialization.

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

### **1. CRUD Endpoints**

#### **Step 1: Create the Book and Customer Repositories**

package com.example.bookstoreapi.repository;

import com.example.bookstoreapi.model.Book;

import com.example.bookstoreapi.model.Customer;

import org.springframework.data.jpa.repository.JpaRepository;

public interface BookRepository extends JpaRepository<Book, Long> {

}

public interface CustomerRepository extends JpaRepository<Customer, Long> {

}

#### **Step 2: Implement CRUD Endpoints in the Book and Customer Controllers**

**BookController.java**

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.dto.BookDTO;

import com.example.bookstoreapi.model.Book;

import com.example.bookstoreapi.repository.BookRepository;

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

import org.springframework.http.ResponseEntity;

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

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/books")

public class BookController {

@Autowired

private BookRepository bookRepository;

// Create a new book

@PostMapping

public Book createBook(@RequestBody BookDTO bookDTO) {

Book book = new Book(bookDTO);

return bookRepository.save(book);

}

// Read all books

@GetMapping

public List<Book> getAllBooks() {

return bookRepository.findAll();

}

// Read a book by ID

@GetMapping("/{id}")

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

Optional<Book> book = bookRepository.findById(id);

return book.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());

}

// Update a book

@PutMapping("/{id}")

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

Optional<Book> bookOptional = bookRepository.findById(id);

if (!bookOptional.isPresent()) {

return ResponseEntity.notFound().build();

}

Book book = bookOptional.get();

book.updateFromDTO(bookDTO); // Assuming updateFromDTO is a method to update the entity from DTO

bookRepository.save(book);

return ResponseEntity.ok(book);

}

// Delete a book

@DeleteMapping("/{id}")

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

if (!bookRepository.existsById(id)) {

return ResponseEntity.notFound().build();

}

bookRepository.deleteById(id);

return ResponseEntity.noContent().build();

}

}

**CustomerController.java**

package com.example.bookstoreapi.controller;

import com.example.bookstoreapi.dto.CustomerDTO;

import com.example.bookstoreapi.model.Customer;

import com.example.bookstoreapi.repository.CustomerRepository;

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

import org.springframework.http.ResponseEntity;

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

import java.util.List;

import java.util.Optional;

@RestController

@RequestMapping("/customers")

public class CustomerController {

@Autowired

private CustomerRepository customerRepository;

// Create a new customer

@PostMapping

public Customer createCustomer(@RequestBody CustomerDTO customerDTO) {

Customer customer = new Customer(customerDTO);

return customerRepository.save(customer);

}

// Read all customers

@GetMapping

public List<Customer> getAllCustomers() {

return customerRepository.findAll();

}

// Read a customer by ID

@GetMapping("/{id}")

public ResponseEntity<Customer> getCustomerById(@PathVariable Long id) {

Optional<Customer> customer = customerRepository.findById(id);

return customer.map(ResponseEntity::ok).orElseGet(() -> ResponseEntity.notFound().build());

}

// Update a customer

@PutMapping("/{id}")

public ResponseEntity<Customer> updateCustomer(@PathVariable Long id, @RequestBody CustomerDTO customerDTO) {

Optional<Customer> customerOptional = customerRepository.findById(id);

if (!customerOptional.isPresent()) {

return ResponseEntity.notFound().build();

}

Customer customer = customerOptional.get();

customer.updateFromDTO(customerDTO); // Assuming updateFromDTO is a method to update the entity from DTO

customerRepository.save(customer);

return ResponseEntity.ok(customer);

}

// Delete a customer

@DeleteMapping("/{id}")

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

if (!customerRepository.existsById(id)) {

return ResponseEntity.notFound().build();

}

customerRepository.deleteById(id);

return ResponseEntity.noContent().build();

}

}

### **2. Validating Input Data**

To ensure data integrity and enforce business rules, you can use validation annotations such as @NotNull, @Size, and @Min.

**BookDTO.java**

package com.example.bookstoreapi.dto;

import lombok.Data;

import javax.validation.constraints.Min;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

@Data

public class BookDTO {

private Long id;

@NotNull(message = "Title is required")

@Size(min = 2, max = 100, message = "Title must be between 2 and 100 characters")

private String title;

@NotNull(message = "Author is required")

private String author;

@Min(value = 0, message = "Price must be a positive number")

private double price;

@NotNull(message = "ISBN is required")

private String isbn;

}

**CustomerDTO.java**

package com.example.bookstoreapi.dto;

import lombok.Data;

import javax.validation.constraints.Email;

import javax.validation.constraints.NotNull;

import javax.validation.constraints.Size;

@Data

public class CustomerDTO {

private Long id;

@NotNull(message = "Name is required")

@Size(min = 2, max = 100, message = "Name must be between 2 and 100 characters")

private String name;

@NotNull(message = "Email is required")

@Email(message = "Email should be valid")

private String email;

@NotNull(message = "Address is required")

private String address;

}

@PostMapping

public Customer createCustomer(@Valid @RequestBody CustomerDTO customerDTO) {

Customer customer = new Customer(customerDTO);

return customerRepository.save(customer);

}

### 3. Optimistic Locking

Optimistic locking helps to handle concurrent updates by using a version field in the entity. If the version in the database does not match the version in the update request, the update is rejected.

#### Step 1: Add a @Version field to the Entity

**Book.java:**

package com.example.bookstoreapi.model;

import lombok.Data;

import javax.persistence.\*;

@Entity

@Data

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

private double price;

private String isbn;

@Version

private Integer version;

// Constructor, getters, setters, and methods like updateFromDTO

}

**Customer.java**

package com.example.bookstoreapi.model;

import lombok.Data;

import javax.persistence.\*;

@Entity

@Data

public class Customer {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String email;

private String address;

@Version

private Integer version;

// Constructor, getters, setters, and methods like updateFromDTO

}

With optimistic locking, if a client tries to update a book or customer, and the version field doesn't match the version in the database, a OptimisticLockingFailureException is thrown. This helps in managing concurrent updates without data loss.

* CRUD Operations: Implemented endpoints for creating, reading, updating, and deleting books and customers.
* Data Validation: Used validation annotations to ensure data integrity.
* Optimistic Locking: Implemented optimistic locking using a @Version field to handle concurrent updates.