**Case Study: Library Book Management System**

**Problem Statement**

You are tasked with building a simple RESTful API to manage books in a library. Each book has an id, title, author, and yearPublished. The API should allow users to:

* Retrieve all books
* Retrieve a specific book by ID
* Add a new book
* Update a book
* Delete a book

**1. Setup Spring Boot Project**

You can create a Spring Boot project using **Spring Initializr** or your IDE with dependencies like:

* Spring Web (for RESTful services)
* Spring Data JPA (for interacting with the database)
* H2 Database (in-memory database for simplicity)

**2. Create the Book Entity**

This class will represent the Book entity in the system.

java

Copy code

package com.example.library.model;

import jakarta.persistence.Entity;

import jakarta.persistence.GeneratedValue;

import jakarta.persistence.GenerationType;

import jakarta.persistence.Id;

@Entity

public class Book {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String title;

private String author;

private int yearPublished;

// Getters and Setters

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

public int getYearPublished() {

return yearPublished;

}

public void setYearPublished(int yearPublished) {

this.yearPublished = yearPublished;

}

}

**3. Create the Repository**

The repository will handle all the data operations for Book. We can use Spring Data JPA for this.

java

Copy code

package com.example.library.repository;

import com.example.library.model.Book;

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

public interface BookRepository extends JpaRepository<Book, Long> {

}

**4. Create the Service Layer**

The service layer is responsible for the business logic. It communicates with the repository to retrieve or manipulate the data.

java

Copy code

package com.example.library.service;

import com.example.library.model.Book;

import com.example.library.repository.BookRepository;

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

import org.springframework.stereotype.Service;

import java.util.List;

import java.util.Optional;

@Service

public class BookService {

@Autowired

private BookRepository bookRepository;

public List<Book> getAllBooks() {

return bookRepository.findAll();

}

public Optional<Book> getBookById(Long id) {

return bookRepository.findById(id);

}

public Book addBook(Book book) {

return bookRepository.save(book);

}

public Book updateBook(Long id, Book bookDetails) {

Book book = bookRepository.findById(id).orElseThrow();

book.setTitle(bookDetails.getTitle());

book.setAuthor(bookDetails.getAuthor());

book.setYearPublished(bookDetails.getYearPublished());

return bookRepository.save(book);

}

public void deleteBook(Long id) {

bookRepository.deleteById(id);

}

}

**5. Create the Controller Layer**

The controller exposes the API endpoints. It interacts with the service layer to handle incoming HTTP requests.

java

Copy code

package com.example.library.controller;

import com.example.library.model.Book;

import com.example.library.service.BookService;

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("/api/books")

public class BookController {

@Autowired

private BookService bookService;

@GetMapping

public List<Book> getAllBooks() {

return bookService.getAllBooks();

}

@GetMapping("/{id}")

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

Optional<Book> book = bookService.getBookById(id);

return book.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.notFound().build());

}

@PostMapping

public Book addBook(@RequestBody Book book) {

return bookService.addBook(book);

}

@PutMapping("/{id}")

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

return ResponseEntity.ok(bookService.updateBook(id, bookDetails));

}

@DeleteMapping("/{id}")

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

bookService.deleteBook(id);

return ResponseEntity.noContent().build();

}

}

**6. Configure H2 Database**

Add the following configuration to application.properties for using an in-memory H2 database.

properties

Copy code

spring.h2.console.enabled=true

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

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

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

spring.jpa.hibernate.ddl-auto=update

**7. Test the API using Postman or Curl**

You can now test the endpoints using a tool like Postman or cURL.

* **Get all books:**

bash

Copy code

GET /api/books

* **Get a specific book by ID:**

bash

Copy code

GET /api/books/{id}

* **Add a new book:**

bash

Copy code

POST /api/books

{

"title": "Spring Boot in Action",

"author": "Craig Walls",

"yearPublished": 2015

}

* **Update a book:**

bash

Copy code

PUT /api/books/{id}

{

"title": "Spring Boot Updated",

"author": "Craig Walls",

"yearPublished": 2016

}

* **Delete a book:**

bash

Copy code

DELETE /api/books/{id}

**Conclusion**

This case study provides a simple overview of creating a Spring Boot REST application with basic CRUD operations, using layers like entity, repository, service, and controller. You can expand this application by adding features such as validation, exception handling, security, etc.

Top of Form

Bottom of Form