**Case Study: Product Management System**

**Application Overview:**

We will build a Spring Boot application that performs CRUD operations on a **Product** entity stored in a MySQL database. The application will expose RESTful APIs to allow creating, updating, deleting, and retrieving products.

**Steps:**

**1. Setting Up the Spring Boot Project**

1. Go to [Spring Initializr](https://start.spring.io/) and create a project:
   * **Project**: Maven
   * **Language**: Java
   * **Spring Boot Version**: 3.x
   * **Dependencies**: Spring Web, Spring Data JPA, MySQL Driver, Spring Boot DevTools, Lombok (optional)
2. Download and extract the project, then import it into your IDE (IntelliJ, Eclipse, etc.).

**2. MySQL Database Setup**

* Create a database in MySQL:

sql

Copy code

CREATE DATABASE product\_db;

**3. Configure Spring Boot Application**

Edit the application.properties or application.yml file to configure the database connection:

For **application.properties**:

properties

Copy code

spring.datasource.url=jdbc:mysql://localhost:3306/product\_db

spring.datasource.username=root

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

For **application.yml**:

yaml

Copy code

spring:

datasource:

url: jdbc:mysql://localhost:3306/product\_db

username: root

password: your\_password

jpa:

hibernate:

ddl-auto: update

show-sql: true

**4. Creating the Product Entity**

Create a Product entity class that maps to the database.

java

Copy code

package com.example.productapp.model;

import jakarta.persistence.\*;

import lombok.\*;

@Entity

@Data // Lombok to reduce boilerplate code (optional)

@NoArgsConstructor

@AllArgsConstructor

@Table(name = "products")

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private String description;

private double price;

}

**5. Creating the Repository**

Define a ProductRepository interface that extends JpaRepository to interact with the database.

java

Copy code

package com.example.productapp.repository;

import com.example.productapp.model.Product;

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

public interface ProductRepository extends JpaRepository<Product, Long> {

}

**6. Creating the Service Layer**

The service layer contains business logic. Create a ProductService interface and an implementation class.

**ProductService.java**

java

Copy code

package com.example.productapp.service;

import com.example.productapp.model.Product;

import java.util.List;

public interface ProductService {

Product saveProduct(Product product);

List<Product> getAllProducts();

Product getProductById(Long id);

Product updateProduct(Product product, Long id);

void deleteProduct(Long id);

}

**ProductServiceImpl.java**

java

Copy code

package com.example.productapp.service;

import com.example.productapp.exception.ResourceNotFoundException;

import com.example.productapp.model.Product;

import com.example.productapp.repository.ProductRepository;

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

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class ProductServiceImpl implements ProductService {

@Autowired

private ProductRepository productRepository;

@Override

public Product saveProduct(Product product) {

return productRepository.save(product);

}

@Override

public List<Product> getAllProducts() {

return productRepository.findAll();

}

@Override

public Product getProductById(Long id) {

return productRepository.findById(id)

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

}

@Override

public Product updateProduct(Product product, Long id) {

Product existingProduct = getProductById(id);

existingProduct.setName(product.getName());

existingProduct.setDescription(product.getDescription());

existingProduct.setPrice(product.getPrice());

return productRepository.save(existingProduct);

}

@Override

public void deleteProduct(Long id) {

productRepository.deleteById(id);

}

}

**7. Creating the Controller Layer**

Create a REST controller to expose the API endpoints for CRUD operations.

**ProductController.java**

java

Copy code

package com.example.productapp.controller;

import com.example.productapp.model.Product;

import com.example.productapp.service.ProductService;

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

import org.springframework.http.ResponseEntity;

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

import java.util.List;

@RestController

@RequestMapping("/api/products")

public class ProductController {

@Autowired

private ProductService productService;

@PostMapping

public Product createProduct(@RequestBody Product product) {

return productService.saveProduct(product);

}

@GetMapping

public List<Product> getAllProducts() {

return productService.getAllProducts();

}

@GetMapping("/{id}")

public ResponseEntity<Product> getProductById(@PathVariable Long id) {

Product product = productService.getProductById(id);

return ResponseEntity.ok(product);

}

@PutMapping("/{id}")

public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product productDetails) {

Product updatedProduct = productService.updateProduct(productDetails, id);

return ResponseEntity.ok(updatedProduct);

}

@DeleteMapping("/{id}")

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

productService.deleteProduct(id);

return ResponseEntity.noContent().build();

}

}

**8. Handling Exceptions**

Create a custom exception for handling cases where a product is not found.

**ResourceNotFoundException.java**

java

Copy code

package com.example.productapp.exception;

import org.springframework.http.HttpStatus;

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

@ResponseStatus(HttpStatus.NOT\_FOUND)

public class ResourceNotFoundException extends RuntimeException {

public ResourceNotFoundException(String message) {

super(message);

}

}

**9. Running the Application**

* Run the Spring Boot application from your IDE or using Maven:

bash

Copy code

mvn spring-boot:run

* The application will be accessible at http://localhost:8080/api/products.

**10. Testing the REST API**

You can use **Postman** or **cURL** to test the REST API endpoints:

* **Create Product (POST)**:

bash

Copy code

POST http://localhost:8080/api/products

Body (JSON):

{

"name": "Laptop",

"description": "High-end gaming laptop",

"price": 1200

}

* **Get All Products (GET)**:

bash

Copy code

GET http://localhost:8080/api/products

* **Get Product by ID (GET)**:

bash

Copy code

GET http://localhost:8080/api/products/1

* **Update Product (PUT)**:

bash

Copy code

PUT http://localhost:8080/api/products/1

Body (JSON):

{

"name": "Gaming Laptop",

"description": "Updated description",

"price": 1500

}

* **Delete Product (DELETE)**:

bash

Copy code

DELETE http://localhost:8080/api/products/1

This simple Spring Boot and MySQL project demonstrates how to set up a RESTful API for managing products. You can expand this case study by adding more features, such as filtering, pagination, or even a front-end interface.