Here’s a simple case study for a **Spring Boot REST Application**. This example showcases a basic **CRUD** (Create, Read, Update, Delete) operation for managing Product data using a RESTful API.

**1. Overview:**

The case study involves developing a REST API for managing products in an inventory system. Each product will have:

* id: Unique identifier
* name: Product name
* price: Product price

**2. Tools and Technologies:**

* Spring Boot
* Spring Data JPA (for database interaction)
* H2 Database (in-memory database for simplicity)
* Spring Web (for REST API)
* Lombok (for reducing boilerplate code)

**3. Project Structure:**

css

Copy code

├── src

│ └── main

│ ├── java

│ │ └── com.example.productapp

│ │ ├── controller

│ │ ├── model

│ │ ├── repository

│ │ └── service

│ └── resources

│ ├── application.properties

│ └── data.sql

**4. Step-by-Step Implementation:**

**4.1 Model Layer:**

Define the Product entity representing a product in the inventory.

java

Copy code

package com.example.productapp.model;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import lombok.Data;

@Entity

@Data

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

private double price;

}

**4.2 Repository Layer:**

Create a ProductRepository interface that extends JpaRepository.

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> {

}

**4.3 Service Layer:**

The service layer contains the business logic. Here, we define the ProductService to manage product operations.

java

Copy code

package com.example.productapp.service;

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 ProductService {

@Autowired

private ProductRepository repository;

public List<Product> getAllProducts() {

return repository.findAll();

}

public Product getProductById(Long id) {

return repository.findById(id).orElse(null);

}

public Product addProduct(Product product) {

return repository.save(product);

}

public Product updateProduct(Product product) {

return repository.save(product);

}

public void deleteProduct(Long id) {

repository.deleteById(id);

}

}

**4.4 Controller Layer:**

The ProductController exposes the REST API endpoints for CRUD operations.

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.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/products")

public class ProductController {

@Autowired

private ProductService productService;

@GetMapping

public List<Product> getAllProducts() {

return productService.getAllProducts();

}

@GetMapping("/{id}")

public Product getProductById(@PathVariable Long id) {

return productService.getProductById(id);

}

@PostMapping

public Product addProduct(@RequestBody Product product) {

return productService.addProduct(product);

}

@PutMapping("/{id}")

public Product updateProduct(@PathVariable Long id, @RequestBody Product product) {

product.setId(id);

return productService.updateProduct(product);

}

@DeleteMapping("/{id}")

public void deleteProduct(@PathVariable Long id) {

productService.deleteProduct(id);

}

}

**4.5 Application Configuration:**

In application.properties, we configure the H2 database and enable the H2 console.

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.hibernate.ddl-auto=update

**4.6 Database Initialization (Optional):**

In data.sql, we can add some initial data for testing purposes.

sql

Copy code

INSERT INTO PRODUCT (name, price) VALUES ('Product 1', 100.0);

INSERT INTO PRODUCT (name, price) VALUES ('Product 2', 150.0);

**5. Testing the API:**

Once the application is up and running, you can test the API using Postman or cURL:

* **GET** /api/products: Fetch all products
* **GET** /api/products/{id}: Fetch product by ID
* **POST** /api/products: Add a new product
* **PUT** /api/products/{id}: Update a product by ID
* **DELETE** /api/products/{id}: Delete a product by ID

**6. Conclusion:**

This simple case study provides a foundation for building REST APIs using **Spring Boot**. It includes layers for data management, business logic, and controller handling, giving a modular structure to the application. You can expand this example by adding more features like **validation**, **exception handling**, or **authentication** using **Spring Security**.