Spring Boot E-Commerce Project Notes

1. Project Overview

This Spring Boot application is designed for an e-commerce system, managing product-related operations. The provided code includes:

- **Controller (ProductController)**: Handles HTTP requests for product operations.
- **Model (Product)**: Represents a product entity for database persistence.
- **Configuration (application.properties)**: Sets up the application and H2 in-memory database.
- **Technologies**: Spring Boot, Lombok, JPA (Hibernate), and H2 database.

2. Code Breakdown

A. ProductController (controller/ProductController.java)

Code:

```
package codex_rishi.ecom_spring.controller;


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

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


@RestController

@RequestMapping("/api")

public class ProductController {

}
```

What's Happening:

- A REST controller class to handle HTTP requests (e.g., GET, POST) for product operations.
- Currently empty but serves as the entry point for API requests.

Annotations:

- **@RestController**:
    - **Purpose**: Marks the class as a RESTful controller, handling HTTP requests and returning data (e.g., JSON) directly.
    - **Function**: Combines @Controller (MVC) and @ResponseBody (serializes output to JSON/XML).
    - **Role**: Enables ProductController to process API requests (e.g., /api/products) and return JSON responses.
- **@RequestMapping("/api")**:

- **Purpose**: Sets a base URL path for all endpoints in the controller.
- **Function**: Prefixes all method URLs with /api (e.g., a /products mapping becomes /api/products).
- **Role**: Organizes API endpoints under /api for a clean structure.

Workflow Connection:

- Acts as the entry point for client requests (e.g., via browser or Postman).
- Will interact with services/repositories to process requests and return responses.
- Example: A GET request to /api/products fetches product data from the database.

B. Product (model/Product.java)

Code:

```
package codex_rishi.ecom_spring.model;


import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;


import java.math.BigDecimal;
import java.util.Date;


@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Product {
    @Id
    private int id;
    private String name;
    private BigDecimal price;
    private String description;
```

```
    private String brand;
    private String category;
    private Date releaseDate;
    private int quantity;
}
```

What's Happening:

- A model class representing a Product entity, mapped to a database table via JPA.
- Defines product attributes (e.g., id, name, price).

Annotations:

- **@Entity**:
    - **Purpose**: Marks the class as a JPA entity, corresponding to a database table.
    - **Function**: Hibernate creates a Product table with columns matching class fields.
    - **Role**: Enables persistence of Product objects in the H2 database.
- **@Id**:
    - **Purpose**: Specifies the primary key field.
    - **Function**: Marks id as the unique identifier for each Product record.
    - **Role**: Ensures unique identification for querying products.
- **@Data (Lombok)**:
    - **Purpose**: Generates boilerplate code (getters, setters, toString(), equals(), hashCode()).
    - **Function**: Adds these methods at compile time, reducing manual coding.
    - **Role**: Simplifies working with Product objects.
- **@AllArgsConstructor (Lombok)**:
    - **Purpose**: Generates a constructor with all fields as parameters.
    - **Function**: Creates a constructor like Product(int id, String name, ...).
    - **Role**: Allows creating fully initialized Product objects.
- **@NoArgsConstructor (Lombok)**:
    - **Purpose**: Generates a no-argument constructor.
    - **Function**: Creates Product() constructor, required by JPA for entity instantiation.
    - **Role**: Enables JPA to create Product objects when fetching from the database.

Fields:

- id (int): Unique product identifier.
- name (String): Product name.
- price (BigDecimal): Precise decimal for prices (e.g., 19.99).
- description (String): Product details.
- brand (String): Product brand.
- category (String): Product category (e.g., electronics).
- releaseDate (Date): Release date.
- quantity (int): Stock quantity.

Workflow Connection:

- Maps to a database table for storing product data.
- Used by the controller to create, retrieve, or update products via API requests.
- Example: A POST request to /api/products creates a new Product and saves it to the database.

C. application.properties

Code:

```
spring.application.name=E-com_spring

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

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update
```

What's Happening:

- Configures the Spring Boot application and H2 in-memory database for development.

Properties:

- **spring.application.name=E-com_spring**:
    - **Purpose**: Names the application.
    - **Function**: Identifies the app in logs or microservices.
    - **Role**: Names the e-commerce app E-com_spring.
- **spring.datasource.url=jdbc:h2:mem:Rishi**:
    - **Purpose**: Specifies the database connection URL.
    - **Function**: Configures an in-memory H2 database named Rishi.
    - **Role**: Provides a lightweight database for testing.

- **spring.datasource.driver-class-name=org.h2.Driver**:
    - **Purpose**: Specifies the JDBC driver for H2.
    - **Function**: Enables Spring Boot to connect to H2.
    - **Role**: Facilitates database communication.
- **spring.jpa.show-sql=true**:
    - **Purpose**: Logs Hibernate SQL queries.
    - **Function**: Prints SQL statements to the console.
    - **Role**: Aids debugging and learning.
- **spring.jpa.hibernate.ddl-auto=update**:
    - **Purpose**: Controls database schema management.
    - **Function**: Updates the schema based on entities (e.g., creates Product table).
    - **Role**: Ensures the database is ready for Product persistence.

Workflow Connection:

- Configures the H2 database and JPA for Product entity persistence.
- Enables the controller to interact with the database via repositories.
- Example: Saving a Product uses the H2 database configured here.

3. How Components Link Together

- **Controller (ProductController)**:
    - Handles HTTP requests (e.g., /api/products).
    - Interacts with services/repositories to manage Product entities.
    - Returns JSON responses to clients.
- **Model (Product)**:
    - Defines product data structure.
    - Maps to a database table via JPA.
    - Used by the controller for CRUD operations.
- **Configuration (application.properties)**:
    - Sets up H2 database and JPA.
    - Enables persistence of Product entities.

Workflow Example:

- A client sends a POST request to /api/products with product data.
- ProductController creates a Product object and passes it to a repository.

- The repository saves it to the H2 database (via application.properties).
- Hibernate generates SQL to insert the product.
- The controller returns a JSON response.

4. Key Technologies

- **Spring Boot**:
    - Java framework for building applications with minimal configuration.
    - Supports REST APIs (@RestController), dependency injection, and JPA.
- **Lombok**:
    - Reduces boilerplate code with annotations.
    - Simplifies Product class with generated methods.
- **H2 Database**:
    - In-memory database for development.
    - Lightweight and temporary for testing.
- **JPA (Hibernate)**:
    - Maps Java objects to database tables.
    - Handles CRUD operations via annotations.

5. Understanding Lombok

What is Lombok?

- A Java library that eliminates boilerplate code (e.g., getters, setters) using annotations.
- Used in Product class with @Data, @AllArgsConstructor, and @NoArgsConstructor.

Usage in Project:

- Simplifies Product class by generating:
    - Getters/setters for fields (via @Data).
    - A full constructor (via @AllArgsConstructor).
    - A no-arg constructor (via @NoArgsConstructor) for JPA.
- Reduces code length and maintenance.

Annotations:

- **@Data**:
    - **Generates**: Getters, setters, toString(), equals(), hashCode().
    - **Role**: Simplifies accessing/modifying Product fields (e.g., product.getName()).
- **@AllArgsConstructor**:

    - **Generates**: Constructor with all fields (e.g., Product(int id, String name, ...)).
    - **Role**: Allows creating fully initialized Product objects.
- **@NoArgsConstructor**:
    - **Generates**: Empty constructor (Product()).
    - **Role**: Required by JPA for entity instantiation.

Behind the Scenes:

- Lombok modifies bytecode during compilation using annotation processing.
- Adds generated methods to the .class file, keeping source code clean.
- Example: Product.class includes getters/setters, but Product.java remains concise.

Benefits:

- Reduces repetitive code.
- Improves readability and maintainability.
- Ensures JPA compatibility (e.g., no-arg constructor).
- Supports controller operations with easy field access.

Workflow Connection:

- @NoArgsConstructor enables JPA to create Product objects from database data.
- @Data provides getters/setters for controller/repository operations.
- @AllArgsConstructor simplifies object creation in code.

Differences Between HTTP Methods (GET, POST, PUT, DELETE)

These are HTTP methods used in REST APIs to perform CRUD operations (Create, Read, Update, Delete):

1. **GET**:
    - **What Happens**: Retrieves data from the server (e.g., list of products).
    - **In Your Project**: GET /api/products would fetch Product objects from the database via the controller.
    - **Example**: Returns [{id: 1, name: "Laptop", ...}].
2. **POST**:
    - **What Happens**: Creates a new resource (e.g., a new product).
    - **In Your Project**: POST /api/products with a JSON body (e.g., {name: "Phone", price: 499.99}) would create a new Product in the database.

- **Example**: Saves a new Product and returns the created object.

3. **PUT**:
    - **What Happens**: Updates an existing resource (e.g., updates a product's details).
    - **In Your Project**: PUT /api/products/1 with a JSON body would update the Product with id=1.
    - **Example**: Modifies fields like price or quantity in the database.
4. **DELETE**:
    - **What Happens**: Removes a resource (e.g., deletes a product).
    - **In Your Project**: DELETE /api/products/1 would delete the Product with id=1 from the database.
    - **Example**: Removes the record and returns a success status.

Method	Purpose	Data Flow	Common Use
`GET`	Get data	Server → Client	Show products, details
`POST`	Send data	Client → Server	Add product, submit form
`PUT`	Update data	Client → Server	Edit product
`DELETE`	Delete data	Client → Server	Remove product

SECOND PHASE->

Spring Boot Product Components Explanation

This document explains the new components added to the e-commerce Spring Boot project for fetching products, including the ProductController, ProductService, and ProductRepo. It covers what each component does, the role of annotations, and how they work together.

1. ProductController (controller/ProductController.java)

Code:

```
public class ProductController {
    @Autowired
    private ProductService service;

    @RequestMapping("/products")
```

```
    public List<Product> getAllProducts() {
        return service.getAllProducts();
    }
}
```

What's Happening:

- A REST controller handling HTTP requests to /api/products (assuming class-level @RequestMapping("/api")).
- The getAllProducts() method returns a list of all Product objects as JSON.
- Calls ProductService's getAllProducts() method to fetch data.

Annotations:

- **@Autowired**: Injects a ProductService bean into the service field. Spring's IoC container creates and assigns the ProductService instance during startup.
- **@RequestMapping("/products")**: Maps HTTP requests to /products to the getAllProducts() method, returning a List<Product> as JSON.

Purpose:

- Provides an API endpoint to fetch all products.
- Acts as the entry point for client requests, delegating logic to the service.

2. ProductService (service/ProductService.java)

Code:

```
import java.util.List;


@Service
public class ProductService {
    @Autowired
    private ProductRepo repo;


    public List<Product> getAllProducts() {
        return repo.findAll();
    }
}
```

What's Happening:

- A service class handling business logic for product operations.

- The getAllProducts() method calls repo.findAll() to retrieve all Product entities from the database.

Annotations:

- **@Service**: Marks the class as a Spring service component, registering it as a bean in the IoC container for dependency injection.
- **@Autowired**: Injects a ProductRepo bean into the repo field, allowing the service to access database operations.

Purpose:

- Encapsulates business logic, separating it from the controller and repository.
- Calls the repository to fetch data, with room for additional logic (e.g., validation) in the future.

3. ProductRepo (repository/ProductRepo.java)

Code:

```
package codex_rishi.ecom_spring.repository;


import codex_rishi.ecom_spring.model.Product;

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

import org.springframework.stereotype.Repository;


@Repository

public interface ProductRepo extends JpaRepository<Product, Integer> {

}
```

What's Happening:

- A repository interface for database operations on the Product entity.
- Extends JpaRepository<Product, Integer>, inheriting methods like findAll() for CRUD operations.
- Empty because JpaRepository provides all necessary methods.

Annotations:

- **@Repository**: Marks the interface as a Spring repository component. Spring Data JPA generates an implementation at runtime.
- **JpaRepository<Product, Integer>**: Provides CRUD methods for the Product entity, with Integer as the primary key type.

Purpose:

- Abstracts database operations, allowing the service to fetch/save products without writing SQL.
- Simplifies data access with built-in methods like findAll().

Workflow

1. A client sends a GET request to /api/products.
2. The ProductController's getAllProducts() method is invoked, calling service.getAllProducts().
3. The ProductService calls repo.findAll() to query the H2 database.
4. The repository returns a List<Product>, which is passed back to the service, then the controller.
5. The controller returns the list as JSON to the client.

Role of @Autowired

- **Purpose**: Enables dependency injection, allowing Spring to automatically provide instances of ProductService and ProductRepo.
- **How it works**: Spring's IoC container creates singleton beans for ProductService (@Service) and ProductRepo (@Repository) and injects them into the service and repo fields, respectively.
- **Benefit**: Eliminates manual instantiation, ensuring loose coupling and reusability.

Why These Components?

- **Controller**: Handles HTTP requests, providing a REST API for clients.
- **Service**: Separates business logic from presentation and data access.
- **Repository**: Abstracts database operations, leveraging Spring Data JPA for simplicity.
- **Layered Architecture**: Promotes modularity, making the code easier to maintain and extend.

Q & A REGARDING NOW

Spring Boot Product Components Q&A

This document answers specific questions about the ProductRepo, ProductService, and ProductController in the e-commerce Spring Boot project, focusing on annotations, the repository interface, and the service layer's purpose.

ProductRepo (repository/ProductRepo.java)

Code:

@Repository

public interface ProductRepo extends JpaRepository<Product, Integer> {}

1. What does @Repository mean, and what does Spring understand?

- **Meaning**: Marks the interface as a data access component.
- **Spring's understanding**: Registers ProductRepo as a bean. Spring Data JPA generates an implementation for database operations.
- **Purpose**: Enables methods like findAll() to interact with the database.

2. Why create an interface extending JpaRepository<Product, Integer>?

- **Why an interface?**: Defines a contract; Spring Data JPA provides the implementation at runtime, abstracting database logic.
- **Why JpaRepository<Product, Integer>?**: Provides CRUD methods for the Product entity, with Integer as the primary key type.
- **Use**: Simplifies database operations (e.g., findAll() retrieves all products) without writing SQL.

3. Why this structure?

- **Purpose**: Reduces boilerplate, abstracts data access, and allows easy extension with custom queries.

ProductService (service/ProductService.java)

Code:

```
@Service
public class ProductService {
   @Autowired
   private ProductRepo repo;

   public List<Product> getAllProducts() {
      return repo.findAll();
   }
}
```

1. What does @Service mean, and what does Spring understand?

- **Meaning**: Marks the class as a business logic component.
- **Spring's understanding**: Registers ProductService as a bean for dependency injection, recognizing it as the business logic layer.
- **Purpose**: Separates business logic from controllers and repositories.

2. What does repo.findAll() do, given the blank interface?

- **What it does**: Inherited from JpaRepository, findAll() queries the H2 database to retrieve all Product records.
- **How it works**: Spring Data JPA's proxy implementation translates findAll() into a SQL query (e.g., SELECT * FROM Product).
- **Why it works**: JpaRepository provides the method; the interface doesn't need implementation.

3. How does repo (an interface) work with @Autowired?

- **How**: @Autowired injects Spring's proxy implementation of ProductRepo.
- **Purpose**: Allows ProductService to call database methods without manual instantiation.
- **Benefit**: Promotes loose coupling.

ProductController and Why Use ProductService

Code:

```
public class ProductController {
   @Autowired
   private ProductService service;

   @RequestMapping("/products")
   public List<Product> getAllProducts() {
      return service.getAllProducts();
   }
}
```

Why create ProductService instead of calling repo.findAll() directly?

- **Why not direct call**:
    - Tight coupling to the repository.
    - Mixes presentation and data access logic.
    - Hard to add business logic or test.
- **Why use ProductService**:
    - **Layered architecture**: Separates concerns (controller: HTTP, service: logic, repository: data).
    - **Modularity**: Allows adding logic (e.g., validation) in the service.
    - **Testability**: Easier to mock the service.
    - **Reusability**: Service can be used across controllers.

Workflow

1. Client sends GET /api/products to ProductController.
2. Controller calls service.getAllProducts().
3. Service calls repo.findAll().
4. Repository queries the H2 database, returning List<Product>.
5. Controller returns the list as JSON.

Third – phase

Spring Boot Backend Explanation for E-Commerce REST API

This document explains the backend code for a Spring Boot e-commerce application, focusing on three REST API endpoints: retrieving a product by ID, adding a product with an image, and serving a product image. It's designed for beginners to understand the flow, data storage, and frontend interaction.

1. The Product Entity

Code

```
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private BigDecimal price;
    private String description;
    private String brand;
    private String category;
    private Date releaseDate;
    private int quantity;
    private String image;
    private String imagetype;
    @Lob
```

```
    private byte[] imagedata;
}
```

What It Does

- **Purpose**: Represents a product in the e-commerce system, mapped to a database table.
- **Annotations**:
    - @Id: Marks id as the primary key.
    - @GeneratedValue(strategy = GenerationType.IDENTITY): Auto-generates unique IDs (e.g., 1, 2, 3).
    - @Lob: Allows imagedata to store large binary data (image bytes).
- **Fields**: Stores product details like name, price, description, brand, category, releaseDate, quantity, image (filename), imagetype (e.g., image/jpeg), and imagedata (image bytes).

Storage

- **Database**: H2 (in-memory or file-based, configured in application.properties).
- **Table**: product with columns matching the fields.
- **How It's Saved**: Hibernate (via JPA) creates the table and stores data using SQL INSERT statements.

Frontend Interaction

- The frontend receives Product as JSON (e.g., {"id": 1, "name": "Realme Narzo", "price": 18999.99, ...}) and displays it in index.html or product-details.html.

---

2. Service Layer: ProductService

Code

```
public Product getProductByID(int id) {
    return repo.findById(id).get();
}

public Product addproduct(Product product, MultipartFile imagefile) throws IOException {
    product.setImage(imagefile.getOriginalFilename());
    product.setImagetype(imagefile.getContentType());
    product.setImagedata(imagefile.getBytes());
    return repo.save(product);
}
```

What It Does

- **Purpose**: Handles business logic, bridging the controller (HTTP requests) and repository (database operations).
- **Why Use It?**: Keeps code modular by separating HTTP handling from data processing.

Method 1: getProductByID

- **Function**: Fetches a product by ID from the database.
- **How It Works**:
    - repo.findById(id): Queries the database (SELECT * FROM product WHERE id = ?).
    - .get(): Extracts the Product (assumes it exists; risky if not found).
- **Storage**: Data comes from the product table.
- **Frontend**: Used by GET /api/product/{id} to display product details.

Method 2: addproduct

- **Function**: Saves a new product with an image.
- **How It Works**:
    - Sets image (filename), imagetype (e.g., image/jpeg), and imagedata (image bytes) from the uploaded file.
    - repo.save(product): Saves to the database via an INSERT statement.
- **Storage**: Data is stored in the product table.
- **Frontend**: Triggered by POST /api/addproduct from add-product.html.

---

3. Controller Layer: ProductController

Code

```
@GetMapping("/product/{id}")
public Product getProduct(@PathVariable int id) {
   return service.getProductByID(id);
}


@PostMapping("/addproduct")
public ResponseEntity<?> addproduct(@RequestPart Product product, @RequestPart MultipartFile imagefile) {
   try {
      Product product1 = service.addproduct(product, imagefile);
```

```
        return new ResponseEntity<>(product1, HttpStatus.CREATED);
    } catch (Exception e) {
        e.printStackTrace();
        return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
    }
}


@GetMapping("/product/{productid}/image")
public ResponseEntity<byte[]> getimagebyproductid(@PathVariable int productid) {
    Product product = service.getProductByID(productid);
    byte[] imagefile = product.getImagedata();
    return
ResponseEntity.ok().contentType(MediaType.valueOf(product.getImagetype())).body(imagefile);
}
```

What It Does

- **Purpose**: Handles HTTP requests and responses for the REST API.
- **REST API**: Uses HTTP methods (GET, POST) to interact with clients (frontend) via URLs.

Endpoint 1: GET /api/product/{id}

- **Function**: Returns a product's details by ID.
- **How It Works**:
    - @PathVariable int id: Extracts the ID from the URL (e.g., /api/product/1).
    - Calls service.getProductByID(id) to fetch the product.
    - Returns JSON (e.g., {"id": 1, "name": "Realme Narzo", ...}).
- **Storage**: Data is retrieved from the product table.
- **Frontend**: Used by product-details.html to show details like name, price, and description.

Endpoint 2: POST /api/addproduct

- **Function**: Creates a new product with an image.
- **How It Works**:
    - @RequestPart Product product: Parses JSON from the multipart/form-data request.
    - @RequestPart MultipartFile imagefile: Extracts the uploaded image.
    - Calls service.addproduct to save the product.

    - Returns 201 Created with the saved product or 400 Bad Request on error.
- **Storage**: Saves to the product table.
- **Frontend**: Triggered by add-product.html form submission, redirects to index.html on success.

Endpoint 3: GET /api/product/{productid}/image

- **Function**: Serves the product's image.
- **How It Works**:
    - @PathVariable int productid: Extracts the ID.
    - Fetches the product and returns its imagedata as a byte[] with Content-Type (e.g., image/jpeg).
- **Storage**: Image data is in the product table's imagedata column.
- **Frontend**: Used in <img src="/api/product/{id}/image"> in index.html and product-details.html.

---

4. How the Frontend Interacts

- **Fetch API**:
    - index.html: Calls GET /api/products to list products, uses /api/product/{id}/image for images.
    - product-details.html: Calls GET /api/product/{id} for details and /api/product/{id}/image for the image.
    - add-product.html: Sends POST /api/addproduct with FormData (JSON + image).
- **JSON and Images**:
    - JSON is parsed to update HTML elements (e.g., productName.textContent = product.name).
    - Images are displayed via <img> tags pointing to the image endpoint.

---

5. Data Storage

- **Database**: H2, configured in application.properties (e.g., spring.datasource.url=jdbc:h2:mem:testdb).
- **Table**: product with columns for all fields.
- **Check Data**: Use H2 console (http://localhost:8080/h2-console) to query:
- SELECT * FROM product;

---

6. Key REST API Concepts

- **REST**: Uses HTTP to perform CRUD (Create, Read, Update, Delete) operations.
- **Endpoints**: URLs like /api/product/{id} mapped to controller methods.
- **HTTP Methods**:
    - GET: Retrieve data.
    - POST: Create data.
- **Status Codes**:
    - 200 OK: Successful GET.
    - 201 Created: Successful POST.
    - 400 Bad Request: Error in request.
- **Content-Type**:
    - application/json: For product data.
    - multipart/form-data: For JSON + image uploads.
    - image/jpeg: For image responses.

---

7. Flow Summary

1. **Frontend Request**:
    - Sends HTTP requests via Fetch API.
2. **Controller**:
    - Maps requests to methods, calls service.
3. **Service**:
    - Processes logic, interacts with repository.
4. **Repository**:
    - Executes SQL via JPA/Hibernate.
5. **Response**:
    - JSON or image data sent to frontend for display.

---

8. Recommendations

- **Error Handling**: Improve getProductByID to avoid crashes:
- public Product getProductByID(int id) {
- return repo.findById(id).orElseThrow(() -> new RuntimeException("Product not found"));

- }
- **CORS**: Enable for frontend-backend communication:
- @Configuration
- public class WebConfig implements WebMvcConfigurer {
- @Override
- public void addCorsMappings(CorsRegistry registry) {
- registry.addMapping("/**").allowedOrigins("http://localhost:8080").allowedMethods("GET", "POST");
- }
- }
- **Scalability**: Consider storing images in a file system or cloud (e.g., AWS S3) instead of byte[].

Questions to Explore

- Why do we separate controllers, services, and repositories?
- What happens if the frontend sends an invalid id to GET /api/product/{id}?
- How would you add a new field like color to the Product entity?

FINAL PHASE ENDS HERE

Backend Explanation: Update, Delete, and Search Functionalities in Spring Boot E-Commerce

This document explains the backend code for updating, deleting, and searching products in a Spring Boot e-commerce application. It covers the controller, service, and repository layers, their interactions, and how they align with the frontend. Designed for beginners, it includes key concepts, minor details, and questions to deepen your understanding.

1. Overview of Components

Controller (ProductController)

- **Purpose**: Handles HTTP requests from the frontend (e.g., index.html, update-product.html) and returns responses.
- **Annotations**:
  - @RestController: Marks the class as a REST API controller, converting responses to JSON.
  - @RequestMapping("/api"): Sets the base URL path for all endpoints (e.g., /api/product/{id}).

- **Role**: Acts as the entry point, receiving requests, calling the service layer, and sending responses (e.g., JSON or HTTP status codes).

Service (ProductService)

- **Purpose**: Contains business logic, processing data between the controller and repository.
- **Role**: Ensures modularity by separating HTTP handling from data operations, making code reusable and testable.

Repository (ProductRepo)

- **Purpose**: Interacts with the H2 database using Spring Data JPA.
- **Extends**: JpaRepository<Product, Integer>, providing built-in CRUD methods (e.g., findById, save, deleteById).
- **Custom Queries**: Uses @Query for the search functionality.
- **Role**: Executes SQL queries via Hibernate, abstracting database operations.

Database

- **H2 Database**: Configured in application.properties (e.g., spring.datasource.url=jdbc:h2:mem:testdb).
- **Table**: product, with columns matching Product entity fields (e.g., id, brand, imagedata).
- **Access**: Use H2 console (http://localhost:8080/h2-console) to inspect data.

2. Update Functionality

Code

Controller:

```
@PutMapping("/product/{id}")
public ResponseEntity<String> updateproduct(@PathVariable int id, @RequestPart Product product, @RequestPart(required = false) MultipartFile imagefile) throws IOException {
    Product product1 = service.updateproduct(id, product, imagefile);
    if (product1 != null) {
        return new ResponseEntity<>("Product updated successfully", HttpStatus.OK);
    } else {
        return new ResponseEntity<>("Product not found", HttpStatus.NOT_FOUND);
    }
}
```

Service:

```
public Product updateproduct(int id, Product product, MultipartFile imagefile) throws IOException {
    Product existingProduct = repo.findById(id).orElse(null);
```

```
    if (existingProduct == null) {
        return null;
    }
    // Update fields
    existingProduct.setBrand(product.getBrand());
    existingProduct.setCategory(product.getCategory());
    existingProduct.setName(product.getName());
    existingProduct.setDescription(product.getDescription());
    existingProduct.setPrice(product.getPrice());
    existingProduct.setQuantity(product.getQuantity());
    existingProduct.setReleaseDate(product.getReleaseDate());
    // Update image only if a new file is provided
    if (imagefile != null && !imagefile.isEmpty()) {
        existingProduct.setImage(imagefile.getOriginalFilename());
        existingProduct.setImagetype(imagefile.getContentType());
        existingProduct.setImagedata(imagefile.getBytes());
    }
    return repo.save(existingProduct);
}
```

How It Works

1. **HTTP Request**:
    - Frontend (update-product.html) sends a PUT /api/product/{id} request with multipart/form-data containing:
        - product: JSON object (e.g., {"id": 1, "name": "Realme Narzo", "quantity": 10, ...}).
        - imagefile: Optional image file (e.g., phone.jpg).
    - @PathVariable int id: Extracts the product ID from the URL (e.g., /api/product/1).
    - @RequestPart Product product: Parses the JSON into a Product object.
    - @RequestPart(required = false) MultipartFile imagefile: Accepts an optional image file.
2. **Controller**:
    - Calls service.updateproduct(id, product, imagefile).

- Returns 200 OK with “Product updated successfully” if the product is updated, or 404 Not Found if not found.

3. **Service**:
    - Fetches the existing product using repo.findById(id).
    - If not found, returns null.
    - Copies fields from the incoming product to existingProduct.
    - Updates image fields (image, imagetype, imagedata) only if a new imagefile is provided.
    - Saves the updated product using repo.save(existingProduct).
4. **Repository**:
    - Hibernate generates an UPDATE SQL query:
    - UPDATE product SET brand = ?, category = ?, name = ?, ..., imagedata = ? WHERE id = ?
    - Only updates imagedata if a new file is provided.
5. **Frontend Alignment**:
    - update-product.html sends the PUT request via a FormData object.
    - Displays the existing image (/api/product/{id}/image) and allows optional image uploads.
    - Redirects to index.html on success.

Key Details

- **Optional Image**: The required = false on imagefile prevents errors when no image is sent, fixing the earlier MissingServletRequestPartException.
- **Image Preservation**: The service checks imagefile != null && !imagefile.isEmpty() to avoid overwriting the existing image.
- **Commented Code**: The existingProduct.setAvailable(product.isAvailable()) is commented out, suggesting the Product entity may not have an available field. If needed, add it to the entity:
- private boolean available;
- **Error Handling**: Returns 404 if the product isn’t found, but could throw an exception for invalid data (e.g., negative price).

3. Delete Functionality

Code

Controller:

@DeleteMapping("/product/{id}")

```
public ResponseEntity<String> deleteproduct(@PathVariable int id) {
   Product product = service.getProductByID(id);
   if (product != null) {
      service.deleteproduct(id);
      return new ResponseEntity<>("Product deleted successfully", HttpStatus.OK);
   } else {
      return new ResponseEntity<>("Product not found", HttpStatus.NOT_FOUND);
   }
}
```

Service:

```
public void deleteproduct(int id) {
   repo.deleteById(id);
}
```

How It Works

1. **HTTP Request**:
    - Frontend (product-details.html) sends a DELETE /api/product/{id} request (e.g., /api/product/1).
    - @PathVariable int id: Extracts the ID from the URL.
2. **Controller**:
    - Calls service.getProductByID(id) to check if the product exists.
    - If found, calls service.deleteproduct(id) and returns 200 OK with “Product deleted successfully.”
    - If not found, returns `404 Not Found.”
3. **Service**:
    - Calls repo.deleteById(id), which deletes the product from the database.
4. **Repository**:
    - Hibernate generates a DELETE SQL query:
    - DELETE FROM product WHERE id = ?
5. **Frontend Alignment**:
    - product-details.html triggers the DELETE request on clicking the “Delete” button.
    - Shows a confirmation prompt and redirects to index.html on success.

Key Details

- **Redundant Check**: The controller checks getProductByID(id) before calling deleteproduct(id). Since deleteById is idempotent (no error if ID doesn't exist), you could simplify:
- @DeleteMapping("/product/{id}")
- public ResponseEntity<String> deleteproduct(@PathVariable int id) {
- if (repo.existsById(id)) {
- service.deleteproduct(id);
- return new ResponseEntity<>("Product deleted successfully", HttpStatus.OK);
- } else {
- return new ResponseEntity<>("Product not found", HttpStatus.NOT_FOUND);
- }
- }
- **No Exception Handling**: If deleteById fails (e.g., database issue), it could throw an exception. Consider adding try-catch:
- public void deleteproduct(int id) {
- try {
- repo.deleteById(id);
- } catch (Exception e) {
- throw new RuntimeException("Failed to delete product: " + e.getMessage());
- }
- }

4. Search Functionality

Code

Controller:

```
@RequestMapping("/produts/search")
public ResponseEntity<List<Product>> searchproducts(@RequestParam String keyword) {
   List<Product> products = service.searchproduct(keyword);
   System.out.println("searching with" + keyword);
   return new ResponseEntity<>(products, HttpStatus.OK);
}
```

Service:

```
public List<Product> searchproduct(String keyword) {
   return repo.searchproducts(keyword);
```

}

Repository:

```
@Repository
public interface ProductRepo extends JpaRepository<Product, Integer> {
   @Query("SELECT p from Product p where LOWER(p.brand) LIKE LOWER(CONCAT('%', :keyword ,'%'))")
   List<Product> searchproducts(String keyword);
}
```

How It Works

1. **HTTP Request**:
    - Frontend (index.html) sends a GET /api/produts/search?keyword={term} request (e.g., /api/produts/search?keyword=Realme).
    - @RequestParam String keyword: Extracts the keyword query parameter.
2. **Controller**:
    - Calls service.searchproduct(keyword) and returns the product list as JSON with 200 OK.
    - Logs the keyword to the console for debugging.
3. **Service**:
    - Forwards the keyword to repo.searchproducts(keyword).
4. **Repository**:
    - Executes the JPQL query:
    - SELECT p FROM Product p WHERE LOWER(p.brand) LIKE LOWER(CONCAT('%', :keyword, '%'))
    - Hibernate converts it to SQL:
    - SELECT * FROM product WHERE LOWER(brand) LIKE LOWER('%keyword%')
    - Searches for products where the brand contains the keyword (case-insensitive).
5. **Frontend Alignment**:
    - index.html sends the search request via a search bar and displays results in the product grid.
    - Shows “No products found” if the response is empty.

Key Details

- **Typo in URL**: The endpoint is /produts/search (should be /products/search). Fix it for clarity:
- @GetMapping("/products/search")

- **Search Scope**: Only searches brand. To search name and category, extend the query:
- @Query("SELECT p FROM Product p WHERE " +
- "LOWER(p.brand) LIKE LOWER(CONCAT('%', :keyword, '%')) OR " +
- "LOWER(p.name) LIKE LOWER(CONCAT('%', :keyword, '%')) OR " +
- "LOWER(p.category) LIKE LOWER(CONCAT('%', :keyword, '%'))")
- **Empty Keyword**: No validation for empty or null keywords. Add handling:
- if (keyword == null || keyword.trim().isEmpty()) {
- return new ResponseEntity<>(repo.findAll(), HttpStatus.OK);
- }
- **Logging**: Replace System.out.println with a logger:
- private static final Logger logger = LoggerFactory.getLogger(ProductController.class);
- logger.info("Searching with: {}", keyword);

5. How Components Align

- **Controller ↔ Service**:
  - The controller receives HTTP requests and delegates to the service for logic.
  - Example: updateproduct calls service.updateproduct, passing id, product, and imagefile.
- **Service ↔ Repository**:
  - The service calls repository methods to interact with the database.
  - Example: repo.findById(id) in updateproduct fetches the product, and repo.save updates it.
- **Repository ↔ Database**:
  - JPA/Hibernate translates repository methods to SQL queries.
  - Example: repo.searchproducts(keyword) generates a SELECT query with a LIKE clause.
- **Frontend ↔ Backend**:
  - index.html: Sends GET /api/produts/search?keyword={term} to display search results.
  - product-details.html: Sends DELETE /api/product/{id} to delete a product.
  - update-product.html: Sends PUT /api/product/{id} with FormData to update a product.

6. Frontend Alignment

- **Update**:

    - update-product.html fetches product data (GET /api/product/{id}) to pre-fill the form and display the image (/api/product/{id}/image).
    - Submits multipart/form-data to PUT /api/product/{id}, optionally including an image.
- **Delete**:
    - product-details.html sends DELETE /api/product/{id} and redirects to index.html on success.
- **Search**:
    - index.html sends GET /api/produts/search?keyword={term} and updates the product grid dynamically.
    - Handles empty results and errors gracefully.

7. Key Spring Boot Concepts

- **REST API**:
    - Uses HTTP methods (GET, PUT, DELETE) to perform CRUD operations.
    - Returns JSON responses wrapped in ResponseEntity for status codes and data.
- **Annotations**:
    - @PutMapping, @DeleteMapping, @RequestMapping: Map HTTP requests to methods.
    - @RequestPart: Parses multipart/form-data parts (JSON or files).
    - @PathVariable: Extracts URL parameters (e.g., id).
    - @RequestParam: Extracts query parameters (e.g., keyword).
    - @Query: Defines custom JPQL queries for JPA.
- **JPA/Hibernate**:
    - Maps Product entity to the product table.
    - Converts repository methods to SQL (e.g., save → INSERT/UPDATE, deleteById → DELETE).
- **MultipartFile**:
    - Handles file uploads (e.g., images) in multipart/form-data requests.
    - required = false makes file uploads optional.

8. Minor Details and Best Practices

- **Typo in Endpoint**: Fix /produts/search to /products/search for consistency.
- **Error Handling**:
    - Add try-catch in updateproduct to handle file I/O errors:
    - try {

- if (imagefile != null && !imagefile.isEmpty()) {
- existingProduct.setImage(imagefile.getOriginalFilename());
- existingProduct.setImagetype(imagefile.getContentType());
- existingProduct.setImagedata(imagefile.getBytes());
- }
- } catch (IOException e) {
- throw new RuntimeException("Failed to process image: " + e.getMessage());
- }

- **Validation**:
  - Validate input data (e.g., non-negative price, non-empty name):
  - if (product.getPrice().compareTo(BigDecimal.ZERO) < 0) {
  - throw new IllegalArgumentException("Price cannot be negative");
  - }
- **Search Improvements**:
  - Extend search to include name and category.
  - Handle empty keywords to return all products.
- **Logging**: Use SLF4J instead of System.out.println.
- **CORS**: Ensure CORS is enabled for frontend requests:
- @Configuration
- public class WebConfig implements WebMvcConfigurer {
- @Override
- public void addCorsMappings(CorsRegistry registry) {
- registry.addMapping("/**").allowedOrigins("http://localhost:8080").allowedMethods("GET", "POST", "PUT", "DELETE");
- }
- }
- **Database Scalability**: Storing images in imagedata (@Lob) can bloat the database. Consider using a file system or cloud storage (e.g., AWS S3) for images.

9. Flow Summary

1. **Update**:
   - Frontend sends PUT /api/product/{id} with FormData.

    - Controller parses product and optional imagefile.
    - Service updates fields, preserves existing image if none provided.
    - Repository saves to the database.
    - Frontend redirects to index.html.
2. **Delete**:
    - Frontend sends DELETE /api/product/{id}.
    - Controller checks existence and deletes via service.
    - Repository executes DELETE.
    - Frontend redirects to index.html.
3. **Search**:
    - Frontend sends GET /api/produts/search?keyword={term}.
    - Controller calls service, which queries the repository.
    - Repository executes a LIKE query on brand.
    - Frontend displays results in the product grid.

10. Questions to Explore

- Why is it important to make imagefile optional in the update endpoint?
- How does the JPQL query in searchproducts achieve case-insensitive search?
- What would happen if you removed the existingProduct check in updateproduct?
- How could you improve the search to include more fields or handle special characters?
- Why might you want to use a logger instead of System.out.println?

11. Recommendations

- **Fix Typo**: Rename /produts/search to /products/search.
- **Add Validation**: Check for valid input data in the service layer.
- **Enhance Search**: Include name and category in the search query.
- **Error Handling**: Add try-catch for database operations.
- **Testing**: Use Postman to test endpoints and H2 console to verify database changes:
- SELECT * FROM product;