### **Product.java (Model Layer)**

* **Purpose**: Declares the Product class, which will be mapped to the database table.
* **Why it's used**: Represents a product entity with its attributes, which can be stored in and retrieved from a database.
* **How it's used**: Defines a Java class for the product entity. This class will interact with other layers of the application, such as repositories and services.

private Long id;

* **Purpose**: Defines the id attribute, which uniquely identifies each product.
* **Why it's used**: Every product in the database must have a unique identifier.
* **How it's used**: This field will be used as a primary key when interacting with the database to store or retrieve a product.

private Long categoryId;

* **Purpose**: Represents the ID of the category to which the product belongs.
* **Why it's used**: It connects the product to a category, allowing products to be organized and categorized.
* **How it's used**: This is a foreign key reference that links the product to a category in the database, ensuring that products are properly categorized.

public Product() {}

* **Purpose**: Defines a default constructor.
* **Why it's used**: The default constructor is needed by frameworks like **JPA** (Java Persistence API) for entity instantiation.
* **How it's used**: This constructor allows for the creation of a Product object without initializing any attributes. It's often used by JPA during entity creation.

public Product(String name, int quantity, double price) {  
 this.name = name;  
 this.quantity = quantity;  
 this.price = price;  
 }

* **Purpose**: Defines a parameterized constructor to initialize a Product object with its essential attributes.
* **Why it's used**: This constructor is useful for creating a product object when you don't have an id yet or when you want to specify only the basic attributes (name, quantity, price) for a new product.
* **How it's used**: This constructor will be called when creating a new product in the service layer or from the controller.

getters & setters

* **Purpose**: Defines the getter and setter for the field.
* **Why it's used**: These methods allow other classes to access and modify the field of a product.
* **How it's used**: The getter retrieves the field, while the setter assigns a new value to field.

### **ProductRepository.java (Repository Layer)**

@Repository

* **Purpose**: Marks the ProductRepository class as a **Spring-managed bean** with the @Repository annotation.
* **Why it's used**: The @Repository annotation is part of **Spring Data JPA** and tells Spring to automatically register the repository as a bean in the **ApplicationContext**, making it available for dependency injection.
* **How it's used**: The ProductRepository class is marked with @Repository to indicate it's a repository that handles database interactions for Product entities.

private final JdbcTemplate jdbcTemplate;

* **Purpose**: Declares the JdbcTemplate object, which is used to perform database operations.
* **Why it's used**: JdbcTemplate simplifies working with a database by handling SQL queries and exception handling, providing a higher-level abstraction over raw JDBC calls.
* **How it's used**: This field will be injected via the constructor, and it will be used for executing SQL queries and updates.

public ProductRepository(JdbcTemplate jdbcTemplate) {  
 this.jdbcTemplate = jdbcTemplate;  
 }

* **Purpose**: Constructor to inject the JdbcTemplate dependency.
* **Why it's used**: Using constructor injection ensures that jdbcTemplate is provided by the Spring container, and it allows easy unit testing by mocking the JdbcTemplate if needed.
* **How it's used**: The JdbcTemplate instance is passed into the constructor when Spring creates an instance of ProductRepository.

private final RowMapper<Product> productRowMapper = (rs, rowNum) -> {  
 Product product = new Product();  
 product.set(...);  
 return product;  
 };

* **Purpose**: Defines a RowMapper for Product to map rows from a SQL result set into Product objects.
* **Why it's used**: A RowMapper is needed to map the rows of the SQL query result into Java objects (Product). This is particularly useful when using JdbcTemplate to query the database.
* **How it's used**: The productRowMapper is defined using a lambda expression to create a new Product object from each row returned by the query. This RowMapper will be passed to JdbcTemplate when performing database queries.

**NOTE:-** The results are mapped into a list of Product objects. If no product is found, Optional.empty() is returned; otherwise, the first product is returned wrapped in an Optional.

### **ProductServiceImpl.java (Service Layer)**

@Service

* **Purpose**: Marks the ProductServiceImpl class as a **Spring service** and implements the ProductService interface.
* **Why it's used**: Tells Spring to treat this class as a service, so it can be injected into other components like controllers.
* **How it's used**: This is a Spring-specific annotation to register the class as a service. The ProductService interface is implemented, ensuring that this class provides the required business logic for products.

private final ProductRepository productRepository;

* **Purpose**: Declares the ProductRepository as a dependency.
* **Why it's used**: The service layer needs to interact with the repository to fetch, save, update, or delete product data in the database.
* **How it's used**: The repository is injected into the service class to enable the service to perform CRUD operations on the Product model.

public ProductServiceImpl(ProductRepository productRepository) {  
 this.productRepository = productRepository;  
 }

* **Purpose**: Constructor for **dependency injection**.
* **Why it's used**: Constructor injection is the preferred method for dependency injection in Spring because it ensures that the dependency (in this case, ProductRepository) is provided when the ProductServiceImpl instance is created.
* **How it's used**: Spring will automatically inject an instance of ProductRepository into the constructor when creating a ProductServiceImpl object.

@Override  
 public List<Product> getAllProducts() {  
 return productRepository.getAllProducts();  
 }

* **Purpose**: Implements the getAllProducts() method from the ProductService interface.
* **Why it's used**: This method retrieves a list of all products from the database.
* **How it's used**: It calls the getAllProducts() method from the ProductRepository, which interacts with the database and returns all Product records.

@Override  
 public Product getProductById(Long id) {  
 return productRepository.getProductById(id)  
 .orElseThrow(() -> new RuntimeException("Product not found with ID: " + id));  
 }

* **Purpose**: Implements the getProductById() method from the ProductService interface.
* **Why it's used**: This method retrieves a product by its unique id. If the product is not found, it throws a RuntimeException.
* **How it's used**: It calls the getProductById() method from the ProductRepository. If the product is not found, it throws an exception with a descriptive message.

### **ProductValidator.java (Validation Layer)**

@Component

* **Purpose**: Marks the ProductValidator class as a **Spring component**.
* **Why it's used**: The @Component annotation tells Spring to manage this class as a Spring bean, making it available for **dependency injection** in other components, such as the service or controller.
* **How it's used**: By using @Component, Spring will automatically detect this class during component scanning and inject it wherever needed.

The **ProductValidator** class is responsible for validating the product attributes before they are processed in the application.

1. **Purpose**: The class validates Product objects to ensure that they meet certain business rules and criteria.
2. **Why it's used**: Validation is necessary to ensure that only valid data is persisted in the database or processed by the business logic. It prevents errors and data integrity issues.
3. **How it's used**: The validateProduct() method is invoked in the service layer (such as ProductServiceImpl) before any product-related operations (e.g., adding, updating) are performed. If validation fails, a ValidationException is thrown, preventing further actions with invalid data.

### **ProductController.java (Controller Layer)**

@RestController  
@RequestMapping("/products")

* **Purpose**: Marks the class as a **REST controller** that handles HTTP requests for product-related operations.
* **Why it's used**:
  + **@RestController**: Tells Spring that this class is a REST controller, meaning it will handle HTTP requests and return data directly (usually in JSON format).
  + **@RequestMapping("/products")**: Specifies the base URL for all product-related API endpoints. All methods in this controller will be mapped under /products.
* **How it's used**: Spring will route all HTTP requests to /products to the methods in this controller.

@PathVariable Long id:

* **Purpose**: This annotation is used to extract the value from the **URL** path and bind it to the method parameter (id in this case).
* **Why it's used**: In RESTful web services, it's common to pass data as part of the URL path. For example, /products/{id} allows the client to request a product with a specific ID. The @PathVariable annotation automatically extracts that ID from the URL and passes it to the controller method.
* **How it's used**: When a request is made to /products/1, the @PathVariable extracts the value 1 from the path, and assigns it to the id parameter in the method. This ID is then passed to the productService.getProductById(id) method to fetch the product with that ID.

ResponseEntity<Product>:

* **Purpose**: ResponseEntity is a generic wrapper for an HTTP response. It contains the status code, headers, and body of the response. In this case, it's wrapping a Product object that will be returned as the response body.
* **Why it's used**: ResponseEntity provides more control over the HTTP response. It allows you to specify not just the data (in this case, the product) but also the HTTP status code, headers, and other properties. It's very useful for customizing the response based on the outcome of the method.
* **How it's used**: In this case, ResponseEntity.ok(product) means the HTTP response will have a 200 OK status, and the Product object will be serialized into JSON (because of @RestController).