Repository Instances, JPA Repositories, Persisting Entities, and Transactions

1 Project Abstract

In this project, you will build and enhance a transactional product ordering system using Spring Boot and Spring Data JPA. The primary goal is to help you understand how to work with **JPA repositories**, **persist entities**, and handle **transactions** effectively in a real-world application.

This assignment simulates a basic e-commerce backend, where products are managed and orders are placed with automatic stock adjustments. You will implement full CRUD operations for the product catalog and ensure transactional integrity when placing orders using custom repository logic.

The key learning areas include:

- Defining and managing JPA entity classes (`Product`, `Order`)
- Using Spring Data JPA repositories for data access
- Implementing custom repository methods using `EntityManager`
- Performing transactional operations using `@Transactional`
- Designing RESTful endpoints to interact with entities via HTTP

You are provided with a project template that includes controller classes, entity classes, and service/repository layers. Your task is to ensure that the product stock is updated reliably when an order is placed, using both standard and custom repository mechanisms.

Following is the requirement specifications:

	Repository Instances, JPA Repositories, Persisting Entities, and Transactions
Modules	
1	Product
2	Order
Product Module Functionalities	
1	Create a product
2	Get all products
3	Update a product by ID
4	Delete a product by ID
Order Module Functionalities	
1	Place a new order (Fully transactional)

2 ASSUMPTIONS, DEPENDENCIES, RISKS / CONSTRAINTS

2.1 ORDER CONSTRAINTS (TransactionService.java)

- When placing an order, if the **product ID does not exist**, the system must throw a `RuntimeException` with the message: **"Product not found"**.

2.2 COMMON CONSTRAINTS

- For all rest endpoints receiving @RequestBody, validation checks must be done and must throw custom exceptions if data is invalid.
- All the business validations must be implemented in Entity classes.
- Do not change, add, remove any existing methods in the service layer.
- In Repository interfaces, custom methods can be added as per requirements.
- All RestEndpoint methods and Exception Handlers must return data wrapped in **ResponseEntity.**

3 Business Validations

3.1 Business Validations - Product

- Product class must be treated as an entity using appropriate annotation.
- id field must be treated as Id generated through IDENTITY technique.

3.2 Business Validations - Order

- Order class must be treated as an entity using appropriate annotation with table name as “orders”.
- id field must be treated as Id generated through IDENTITY technique.

4 Rest Endpoints

Rest End-points to be exposed in the controller along with method details for the same to be created

4.1 Product Controller

URL Exposed		Purpose
1. /api/products		Creates a new product
Http Method	POST	
Parameter	**The product data to be created must be received in the controller using @RequestBody.**	
Return	Product	
2. /api/products		Gets list of all products
Http Method	GET	
Parameter 1	-	

Return	List<Product>	
3. /api/products		Updates a product by ID
Http Method	PUT	
Parameter 1	Long (id) (as @PathVariable)	
Parameter 2	Product data via `@RequestBody` **The product data to be updated must be received in the controller using @RequestBody.**	
Return	Product	
4. /api/products/{id}		Deletes a product by ID
Http Method	DELETE	
Parameter 1	Long (id) (as @PathVariable)	
Return	-	

4.2 Order Controller

URL Exposed		Purpose
1. /api/orders		Places an order and updates stock
Http Method	POST	
Parameter	**The order data to be placed must be received in the controller using @RequestBody.**	
Return	-	

5 Template Code Structure

5.1 Package: com.yaksha.assignment

Resources

ECommerceApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

5.2 Package: com.yaksha.assignment.repository

Resources

Class/Interface	**Description**	**Status**
ProductRepository (interface)	● Repository interface exposing CRUD functionality for Product Entity. ● You can go ahead and add any custom methods as per requirements.	Already implemented.
OrderRepository (interface)	● Repository interface exposing CRUD functionality for Order Entity. ● You can go ahead and add any custom methods as per requirements.	Already implemented.
CustomProductRepository (interface)	● Repository class using `EntityManager` for custom logic. ● You can go ahead and add any custom methods as per requirements.	Partially implemented.

	• Method `updateProductStock(Long productId, Integer quantitySold)` should update stocks manually for transactions.	

5.3 PACKAGE: COM.YAKSHA.ASSIGNMENT.SERVICE

Resources

Class/Interface	Description	Status
ProductService (Class)	• Exposes methods to manage product-related operations such as create, delete, and get list. • Implements all business logic for products. • Do not modify, add or delete any method.	To be implemented
TransactionService (Class)	• Implements the transactional flow for placing an order. • Contains template method implementation. • Coordinates product stock updates using a custom repository. • Do not modify, add or delete any method signature.	To be implemented

5.4 PACKAGE: COM.YAKSHA.ASSIGNMENT.CONTROLLER

Resources

Class/Interface	Description	Status

ProductController (Class)	• Controller class to expose all REST endpoints for product-related activities. • Handles creation, update, retrieval, and deletion of products.	To be implemented
OrderController (Class)	• Controller class to expose order-placement endpoint. • Delegates business logic to `TransactionService`.	To be implemented

5.5 PACKAGE: COM.YAKSHA.ASSIGNMENT.ENTITY

Resources

Class/Interface	**Description**	**Status**
Product (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with `id` as primary key. • Generate the id using the IDENTITY strategy • Map this class with a `product` table.	Partially implemented.

Order (Class)	<ul><li>This class is partially implemented.</li><li>Annotate this class with proper annotation to declare it as an entity class with `id` as primary key.</li><li>Generate the id using the IDENTITY strategy</li><li>Map this class with an `orders` table.</li></ul>	

Method Descriptions:

1. Service Class– Method Descriptions

a. ProductService class

- Declare a private final variable with name `productRepository` of type `ProductRepository` interface.

Method	Task	Implementation Details
`@Autowired public ProductService(ProductRepository productRepository)`	Constructor-based dependency injection	- Annotated with `@Autowired`. - Injects the repository dependency through constructor. - Assigns to the `productRepository` field.

Method	Task	Implementation Details
`getAllProducts`	To implement logic to retrieve all product records from the database	- Uses the JPA repository method productRepository.findAll(). - No parameters required. - Returns a list of all Product entities from the database.

		- Automatically uses Spring's transactional read-only behavior for repository methods.
`saveProduct`	To implement logic for creating or updating a product entity	- Accepts a Product object as input. - Calls productRepository.save(product) to persist the entity. - If the product ID exists, it updates; otherwise, it creates a new entry. - Involves transaction management implicitly via Spring Data JPA. - Returns the saved or updated Product entity.
`deleteProduct`	To implement logic for deleting a product by its ID	- Accepts a Long ID as input. - Calls productRepository.deleteById(id) to delete the product from the database. - No return value (void). - Handled within a transactional context by Spring.

b. TransactionService class

- Declare a private final variable with name `orderRepository` of type `OrderRepository` interface.
- Declare a private final variable with name `productRepository` of type `ProductRepository` interface.
- Declare a private final variable with name `customProductRepository` of type `CustomProductRepository` interface.

Method	Task	Implementation Details
`@Autowired public TransactionService(OrderRepository orderRepository, ProductRepository productRepository, CustomProductRepository customProductRepository)`	Constructor-based dependency injection	- Annotated with `@Autowired`. - Injects the repository dependency through constructor. - Assigns to the `orderRepository`, `productRepository`, and `customProductRepository` fields.

Method	Task	Implementation Details
`placeOrder`	To implement logic to place an order and update product stock in a transactional manner	- Annotated with @Transactional to ensure atomicity. - Accepts an Order object as input. - Calls orderRepository.save(order) to persist the order. - Retrieves the Product entity using productRepository.findById(order.getProductId()). - If product not found, throws RuntimeException("Product not found"). - Calls customProductRepository.updateProductStock(product.getId(), order.getQuantity()) to update stock. - Saves the updated product using productRepository.save(product). - All operations occur within a single transaction to maintain consistency.

2. Controller Class– Method Descriptions

a. ProductController

- Declare a private final variable with name `productService` of type `ProductService` class.

Method	Task	Implementation Details
`@Autowired public ProductController(ProductService productService)`	Constructor-based dependency injection	- Annotated with `@Autowired`. - Injects the repository dependency through constructor. - Assigns to the `productService` field.

Method	Task	Implementation Details

`getAllProducts`	To implement logic to fetch all product entities	- Request type: GET, URL: /api/products - Method name: getAllProducts - Returns List<Product> - Calls productService.getAllProducts()
`createProduct`	To implement logic to create a new product	- Request type: POST, URL: /api/products - Method name: createProduct - Accepts @RequestBody Product - Returns the created Product - Calls productService.saveProduct(product)
`updateProduct`	To implement logic to update an existing product by ID	- Request type: PUT, URL: /api/products/{id} - Method name: updateProduct - Accepts @PathVariable for ID and @RequestBody Product - Sets the ID on the product before saving - Returns the updated Product - Calls productService.saveProduct(product)
`deleteProduct`	To implement logic to delete a product by ID	- Request type: DELETE, URL: /api/products/{id} - Method name: deleteProduct - Accepts @PathVariable for ID - No return value (void) - Calls productService.deleteProduct(id)

b. OrderController

- Declare a private final variable with name `transactionService` of type `TransactionService` class.

Method	Task	Implementation Details
`@Autowired public OrderController(TransactionService transactionService)`	Constructor-based dependency injection	- Annotated with `@Autowired`. - Injects the repository dependency through constructor. - Assigns to the `transactionService` field.

Method	Task	Implementation Details

`placeOrder`	To implement logic for placing a new order	- Request type: POST, URL: /api/orders - Method name: placeOrder - Accepts @RequestBody Order - Calls transactionService.placeOrder(order) - No return value (void)

Execution Steps to Follow

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.**
3. **cd into your backend project folder.**
4. **To build your project use command:**

   mvn clean package -Dmaven.test.skip
5. **To launch your application, move into the target folder (cd target). Run the following command to run the application:**

   java -jar <your application jar file name>
6. **To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN. Please use 127.0.0.1 instead of localhost to test rest endpoints.**
7. **Mandatory: Before final submission run the following command:**

   mvn test