Repository Instances, JPA Repositories, Persisting Entities, and Transactions

Assignment: Implement AppController to Fetch Data from API

Objective

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.

Project Setup

Your project structure is:
- **Main Application Class**: The entry point of the Spring Boot application that bootstraps and launches the service.
- **Controller Class**: These handle incoming HTTP requests and delegate business logic to the service layer.

1. ProductController: Manages CRUD operations for products.

2. OrderController: Handles order placement operations.

- **Entity Class**: JPA entities that represent the data models stored in the relational database.

1. Product: Represents a product with attributes like name, description, price, and stock quantity.
2. Order: Represents an order with product reference and quantity.

- **Repository Interface**:Define how entities are accessed and manipulated at the database level.

1. ProductRepository: A standard JPA repository for `Product` entities.
2. OrderRepository: A JPA repository for managing `Order` entities.
3. CustomProductRepository: A custom repository that uses `EntityManager` to directly manipulate product stock.

- **Service Classes**:Encapsulate business logic and act as a bridge between the controller and repository layers.

1. ProductService: Manages product-related operations such as listing, creating, updating, and deleting products.
2. TransactionService: Handles order placement and ensures stock updates occur within a single transactional boundary.

Key Components to Implement

1. Controller Class

ProductController

In the `ProductController` class, you need to implement the following functionality:
- The class must have the appropriate annotation to define it as a REST controller.
- It must be mapped to handle requests with the base path `/api/products`.

Autowired Dependency:
The `ProductService` should be injected into the `ProductController` to interact with the business logic for product operations.

- **Method: Create Product (POST)**

  - Method Name: `createProduct`

  - Return Type: `Product`

  - Parameters: A `Product` object passed in the body of the HTTP POST request.

  - HTTP Method and Endpoint: Handles HTTP POST requests to `/api/products`.

  - Functionality: Saves the product using the service layer and returns the saved product.

- **Method: Update Product (PUT)**

  - Method Name: `updateProduct`

  - Return Type: `Product`

  - Parameters: Path variable `id` and request body `Product` object.

  - HTTP Method and Endpoint: Handles HTTP PUT requests to `/api/products/{id}`.

  - Functionality: Updates the product with the given ID by setting its ID and saving the updated product.

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

  - Method Name: `getAllProducts`

  - Return Type: `List<Product>`

  - Parameters: None

  - HTTP Method and Endpoint: Handles HTTP GET requests to `/api/products`.

  - Functionality: Retrieves and returns a list of all products in the database.

- **Method: Delete Product (DELETE)**

  - Method Name: `deleteProduct`

  - Return Type: `void`

  - Parameters: Path variable `id` (Long)

  - HTTP Method and Endpoint: Handles DELETE requests to `/api/products/{id}`.

  - Functionality: Deletes the product with the specified ID.

OrderController

In the `OrderController` class, you need to implement the following functionality:
- The class must be annotated properly as a REST controller.
- It should be mapped to handle requests with the base path `/api/orders`.

Autowired Dependency:
The `TransactionService` should be injected into the `OrderController` to manage order placement and transactional logic.

- **Method: Place Order (POST)**

- Method Name: `placeOrder`

- Return Type: `void`

- Parameters: A `Order` object passed in the body of the HTTP POST request.

- HTTP Method and Endpoint: Handles HTTP POST requests to `/api/orders`.

- Functionality: Saves the order and triggers stock deduction for the associated product using the transactional service.

2. Entity Class

- In the `Product` entity class, you need to implement the following:

1. **Entity Annotation**:
   Add the appropriate annotation to the class to mark it as a JPA entity, which will map to a database table.

2. **Primary Key**:
   Add an annotation to the `id` field to mark it as the primary key, and set it to auto-generate its value using Identity technique.

- In the `Order` entity class, you need to implement the following:

1. **Entity Annotation**:
   Add the appropriate annotation to the class to mark it as a JPA entity, and map it to the 'orders' table.

2. **Primary Key**:
   Add an annotation to the `id` field to mark it as the primary key, and set it to auto-generate its value using Identity technique.

3. Repository Interface

- In the `ProductRepository` interface, you need to implement the following:
1. **Repository Annotation**: Ensure the interface is marked as a Spring Data JPA repository.

- In the `OrderRepository` interface, you need to implement the following:
1. **Repository Annotation**: Ensure the interface is marked as a Spring Data JPA repository.

- In the `CustomProductRepository` interface, you need to implement the following:
1. **Repository Annotation**: Ensure the interface is marked as a Spring Data JPA repository.
2. **Persistence Context**: Inject `EntityManager` using `@PersistenceContext` to perform low-level operations on the `Product` entity.
3. **Custom Method - Update Stock**:
    - **Method Name**: `updateProductStock`
    - **Parameters**: `productId` (Long), `quantitySold` (Integer)
    - **Functionality**: Fetch the product using its ID and decrease the stock based on the quantity sold.
    - **Note**: This method is manually updating the state of the entity; make sure it's invoked within a transactional context to persist changes.

4. Service Classes

ProductService class

**Service Annotation:
Annotate the class with `@Service` to mark it as a Spring-managed service component.

**Autowired Dependency:
Inject the `ProductRepository` into the service to interact with the database layer.

- Method: Get All Products

  - Method Name: `getAllProducts`
  - Return Type: `List<Product>`
  - Parameters: None
  - Functionality: Retrieves and returns a list of all products from the database using `findAll()`.

- Method: Save Product

  - Method Name: `saveProduct`
  - Return Type: `Product`

- Parameters: A `Product` object
- Functionality: Creates or updates a product using the `save()` method of `ProductRepository`.

- Method: Delete Product

- Method Name: `deleteProduct`
- Return Type: `void`
- Parameters: `id` (Long)
- Functionality: Deletes a product by its ID using the `deleteById()` method of `ProductRepository`.

TransactionService class

**Service Annotation:
Annotate the class with `@Service` to make it a Spring service bean.

** Autowired Dependencies:
Inject the following:
- `OrderRepository` to save the order.
- `ProductRepository` to fetch and persist product data.
- `CustomProductRepository` to handle custom stock update logic using `EntityManager`.

** Transactional Management:
Annotate the `placeOrder` method with `@Transactional` to ensure atomic operations during the order and stock update process.

- Method: Place Order

- Method Name: `placeOrder`
- Return Type: `void`
- Parameters: An `Order` object
- Functionality:
  - Saves the order to the database.
  - Retrieves the associated product using `productId`.
  - Updates the product's stock using `CustomProductRepository`.
  - Persists the updated product back to the database.
  - All steps occur within a single transaction to maintain data consistency.

**TransactionService – Implementation Details:

- **Save Order**: Persist the order using the order repository to ensure it is recorded in the database.
- **Retrieve Product**: Fetch the product associated with the order using its product ID. If the product does not exist, throw a `RuntimeException` with the message: `"Product not found"`.

- **Update Product Stock**: Use the custom product repository to reduce the product's stock based on the quantity in the order.
- **Persist Updated Product**: Save the updated product to reflect the reduced stock level in the database.
- **Transactional Integrity**: The entire `placeOrder` method must be annotated with `@Transactional` to ensure atomicity. If any part of the operation fails, all changes should be rolled back to maintain data consistency.

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
8. **You need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.**