Querying and Modifying Entities

Overview

In this lab you'll enhance your "Online Retailer" app so that it persists product suggestions data to a relational database. You'll put all your persistence logic in a dedicated repository class.

IntelliJ starter project

If you're happy to continue where you left off in the previous lab, use the following project:

- **`student\student-online-retailer`**

If you'd prefer a fresh start, use the solution project from the previous lab instead:

- **`solutions\solution-integrating-data-sources`**

IntelliJ solution project

The solution project for this lab is located here:

- **`solutions\solution-querying-modifying-entities`**

Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Getting started defining a repository class
2. Implementing the repository class
3. Using the repository class
4. (If time permits) Defining a parameterized operation

Familiarization

The **`student`** folder has an interface named **`ProductSuggestionRepository`**. Add this interface into your IntelliJ project now. The interface specifies the persistence methods we want you to implement in this lab…

Exercise 1: Getting started defining a repository class

Define a class named **`ProductSuggestionRepositoryImpl`** as follows:

- The class should implement the **`ProductSuggestionRepository`** interface.
- Annotate the class with **`@Repository`**. This tells Spring Boot to create a singleton instance of the class during application start-up.

Exercise 2: Implementing the repository class

Implement the **`ProductSuggestionRepositoryImpl`** class as follows:

- Declare an **`EntityManager`** instance variable. Annotate the instance variable with **`@PersistenceContext`** (under the surface, there's an **`EntityManagerFactory`** bean that will create an **`EntityManager`** object and inject it into this variable).
- Implement all the methods for the class, as specified by the interface. For any methods that modify data in the database, annotate these methods with **`@Transactional`**.
- Note that the **`insertProductSuggestion()`** method needs to return the id of the product suggestion after it has been inserted into the database. To achieve this effect, call **`entityManager.flush()`** within your method. This causes the **`EntityManager`** to perform the INSERT statement immediately, which generates the id and copies it into the **`ProductSuggestion`** entity. You can then get the id out of the entity, and return it from your method.

Exercise 3: Using the repository class

Add some code in the application class to test that your repository class works as expected, e.g., test that you can insert product suggestions, query them, and modify their details. (Later in the course, you'll define a proper REST API that utilizes your repository class in a full application.)

Exercise 4 (If time permits): Additional suggestions

In the **`ProductSuggestionRepository`** interface, declare a new method as follows:

```
@Transactional
int increasePriceForPopularProducts(long sales);
```

Implement this method in your **`ProductSuggestionRepositoryImpl`** class, so that it increases the price by 10% for all products that exceed a certain number of sales per year. Follow these steps…

First, define a parameterized JPQL query string as follows:

```
String jpql = "update ProductSuggestion " +
              "set recommendedPrice=recommendedPrice*1.1 " +
              "where estimatedAnnualSales >= :s";
```

Then create a **`Query`** and set a parameter on it:

```
Query query = entityManager.createQuery(jpql);
query.setParameter("s", sales);
```

Execute the query as an *update* statement as follows:

```
int numRowsAffected = query.executeUpdate();
```

Return the result from the function. Then add some code in the application class, to test your new repository method.