16-JPQL & NativeQuery

Author: Vincent Lau

Learning Objectives

> Understand the basics of implementing JPQL & Native Query

JPQL (Java Persistence Query Language)

Introduction

JPQL is a query language defined by the JPA specification and is used to query data from the database using object-oriented syntax. It allows you to write database queries in a way that is independent of the specific database management system (DBMS) being used. JPQL queries operate on entity objects and their attributes.

Key points about JPQL

- JPQL queries are written as strings, and they resemble SQL queries, but they operate on Java entities.
- JPQL queries are type-safe and object-oriented, meaning they use entity classes and their attributes directly in the queries.
- JPQL queries are database-agnostic, so you can write queries that work with different databases as long as they are JPA-compliant.

Code Example in Spring Boot

In a Spring Boot application using JPA (Java Persistence API), you can use JPQL (Java Persistence Query Language) to perform database queries. Here are some examples of using JPQL in Spring Boot:

Basic JPQL Query

In this example, we'll create a simple JPQL query to retrieve all entities of a specific type from the database.

```
1 import org.springframework.data.jpa.repository.Query;
2 import org.springframework.data.repository.CrudRepository;
3
4 public interface MyEntityRepository extends CrudRepository<MyEntity, Long> {
5
6     @Query("SELECT e FROM MyEntity e")
7     List<MyEntity> findAllEntities();
8
9 }
```

In this code, `MyEntityRepository` is a Spring Data JPA repository. The `@Query` annotation is used to specify the JPQL query, and the `findAllEntities` method will return a list of `MyEntity` objects.

Parameterized JPQL Query

You can create parameterized JPQL queries to filter results based on certain criteria. Here's an example of a parameterized query:

```
1 @Query("SELECT e FROM MyEntity e WHERE e.name = :name")
2 List<MyEntity> findEntitiesByName(@Param("name") String name);
```

In this example, the `:name` placeholder is a named parameter, and the `@Param` annotation is used to specify the parameter name.

JPQL with Sorting

You can also use JPQL to perform sorting on query results. For instance, to retrieve entities sorted by a specific field:

```
1 @Query("SELECT e FROM MyEntity e ORDER BY e.creationDate DESC")
2 List<MyEntity> findAllEntitiesSortedByDateDesc();
```

In this example, the `ORDER BY` clause is used to specify the sorting criteria.

JPQL with Aggregation

JPQL allows you to perform aggregate functions. For instance, to calculate the average of a numeric field:

```
1 @Query("SELECT AVG(e.price) FROM Product e")
2 Double calculateAveragePrice();
```

This query calculates the average price of all products.

JPQL with Joins

You can use JPQL to perform joins between related entities. For example, to retrieve orders along with their associated customers:

```
1 @Query("SELECT o FROM Order o JOIN FETCH o.customer")
2 List<Order> findAllOrdersWithCustomers();
```

The `JOIN FETCH` clause ensures that the related customer is eagerly fetched in the result.

These are some common use cases of JPQL in Spring Boot JPA development. You can use JPQL to perform various types of queries, including simple selections, filtering, sorting, aggregations, and joins, all within a Java-oriented and type-safe syntax.

Native Query

Introduction

A native query is a SQL query written in the native query language of the database system being used (e.g., SQL for PostgreSQL, MySQL, etc.). Native queries are executed directly on the database, and they are not database-agnostic like JPQL. In JPA, you can execute native queries using the `createNativeQuery` method.

Key points about Native Queries

- Native queries are written in SQL, which is specific to the database system.
- They provide more control and flexibility, allowing you to use database-specific features.
- Native queries can be useful when you need to perform complex operations that are not easily expressed in JPQL.

Code Examples in Spring Boot

In Spring Boot with JPA, you can execute native SQL queries using the `@Query` annotation with the `nativeQuery` attribute set to `true`, or by using the `EntityManager`. Below are examples of using native queries in Spring Boot:

`@Query` Annotation with `nativeQuery` Attribute

Select Query

To execute a native select query, use the `@Query` annotation with the `nativeQuery` attribute set to `true`.

This will execute a native SQL select query to retrieve all entities from the database.

```
1 import org.springframework.data.jpa.repository.Query;
2 import org.springframework.data.repository.CrudRepository;
3
4 public interface MyEntityRepository extends CrudRepository<MyEntity, Long> {
5     @Query(value = "SELECT * FROM my_entity", nativeQuery = true)
6     List<MyEntity> findAllEntities();
7 }
```

Update Query

Native queries can also be used for update operations.

In this example, the `@Modifying` annotation is used to indicate that the query will perform a modification, and the `nativeQuery` attribute is set to `true`.

```
1 @Modifying
```

```
@Query(value = "UPDATE my_entity SET name = :newName WHERE id = :id",
nativeQuery = true)
int updateEntityName(@Param("newName") String newName, @Param("id") Long id);
```

Using `EntityManager` for Native Queries

Select Query

You can use the `EntityManager` to execute native queries directly.

This code uses the `EntityManager` to execute a native SQL select query.

```
@Autowired
private EntityManager entityManager;

public List<MyEntity> findAllEntities() {
    return entityManager.createNativeQuery("SELECT * FROM my_entity",
MyEntity.class)
        .getResultList();
}
```

Update Query

For update queries using the `EntityManager`, you can use the `createNativeQuery` method.

The `@Transactional` annotation ensures that the update is performed within a transaction.

```
@Autowired
private EntityManager entityManager;

@Transactional
public int updateEntityName(String newName, Long id) {
    Query query = entityManager.createNativeQuery("UPDATE my_entity SET name =
:newName WHERE id = :id");
    query.setParameter("newName", newName);
    query.setParameter("id", id);
    return query.executeUpdate();
}
```

Summary

In summary, you can use native queries in Spring Boot JPA by either using the `@Query` annotation with the `nativeQuery` attribute or by directly using the `EntityManager` to execute native SQL queries. Native queries are useful for more complex SQL operations that cannot be expressed easily in JPQL.