**Spring Data JPA with Spring Boot, Hibernate**

## **🔍 Demonstrating Hibernate Query Language and Native Query in Spring Data JPA**

### **1. Introduction to Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL)**

#### **What is HQL?**

HQL stands for **Hibernate Query Language**. It is an object-oriented query language, similar to SQL but instead of working with tables and columns, it works with Java objects (entities) and their properties.

#### **What is JPQL?**

JPQL stands for **Java Persistence Query Language**. It is defined as part of the JPA specification and is a platform-independent object-oriented query language. It operates on entity objects and their relationships.

### **2. Syntax Similarity and Key Differences between HQL and JPQL**

| **Feature** | **HQL** | **JPQL** |
| --- | --- | --- |
| Framework | Provided by Hibernate ORM | Standard in JPA specification |
| Syntax Base | SQL-like but works with entity names | SQL-like but portable and works with entities |
| Platform Dependency | Tightly coupled with Hibernate | Vendor-agnostic, works with any JPA provider |
| Native Query Support | Supports native queries | Native queries supported via @Query |
| Support in Spring Data | Fully supported | Fully supported |

Example of both (they look similar):

// JPQL

@Query("SELECT c FROM Country c WHERE c.name = ?1")

List<Country> findByNameJPQL(String name);

// HQL (same syntax, just conceptually from Hibernate)

@Query("FROM Country WHERE name = ?1")

List<Country> findByNameHQL(String name);

### **3. The @Query Annotation in Spring Data JPA**

Spring Data JPA provides the @Query annotation to define **custom JPQL or native SQL queries** directly on repository methods.

#### **Syntax**

@Query("SELECT c FROM Country c WHERE c.name = ?1")

List<Country> findByName(String name);

This is a JPQL query. You refer to the **entity name** (Country) and **entity properties** (name) rather than the table and column names.

#### **Named Parameters**

@Query("SELECT c FROM Country c WHERE c.name = :name")

List<Country> findByName(@Param("name") String name);

### **4. Using fetch Keyword in HQL / JPQL**

#### **Why use fetch?**

In JPA, relationships between entities can be loaded **lazily** or **eagerly**. The fetch keyword allows for fetching associated entities in a single query to avoid the N+1 select problem.

#### **Example:**

@Query("SELECT c FROM Country c JOIN FETCH c.states WHERE c.name = ?1")

Country findCountryWithStates(String name);

Here, states is a mapped collection in Country, e.g., @OneToMany.

Without fetch, accessing country.getStates() would trigger a separate query. With JOIN FETCH, it is loaded eagerly in one query.

### **5. Aggregate Functions in HQL / JPQL**

JPQL and HQL both support the following aggregate functions:

* COUNT()
* SUM()
* AVG()
* MAX()
* MIN()

#### **Example Queries**

// Count the number of countries

@Query("SELECT COUNT(c) FROM Country c")

Long countAllCountries();

// Find the maximum population

@Query("SELECT MAX(c.population) FROM Country c")

Long findMaxPopulation();

// Find the average area

@Query("SELECT AVG(c.area) FROM Country c")

Double findAverageArea();

### **6. Native Queries in Spring Data JPA**

Sometimes, you may want to execute **raw SQL queries** instead of JPQL/HQL. This can be done using the @Query annotation with nativeQuery = true.

#### **Syntax:**

@Query(value = "SELECT \* FROM country WHERE name = ?1", nativeQuery = true)

List<Country> findByNameNative(String name);

Here:

* value contains the SQL query string.
* nativeQuery = true indicates this is a raw SQL query.
* You must use **table and column names**, not entity or property names.

#### **Named Parameters in Native Query:**

@Query(value = "SELECT \* FROM country WHERE name = :name", nativeQuery = true)

List<Country> findByNameNative(@Param("name") String name);

### **7. Native Query with Custom Result Mapping**

When a native query does not return a full entity, you can map it to an interface or DTO.

#### **Using interface-based projection:**

public interface CountryNameOnly {

String getName();

}

@Query(value = "SELECT name FROM country", nativeQuery = true)

List<CountryNameOnly> findCountryNames();

### **8. Summary of @Query Usage Scenarios**

| **Use Case** | **Syntax Example** |
| --- | --- |
| JPQL query | @Query("SELECT c FROM Country c WHERE c.name = ?1") |
| Native query | @Query(value = "SELECT \* FROM country WHERE name = ?1", nativeQuery = true) |
| Named parameters | @Query("SELECT c FROM Country c WHERE c.name = :name") |
| Aggregate function | @Query("SELECT COUNT(c) FROM Country c") |
| Join Fetch | @Query("SELECT c FROM Country c JOIN FETCH c.states WHERE c.name = ?1") |
| Return DTO / Interface Projection | List<CountryNameOnly> |

### **9. Code Demonstration in a Spring Boot Application**

#### **Entity: Country**

@Entity

public class Country {

@Id

private String code;

private String name;

private long population;

private double area;

// getters and setters

}

#### **Repository**

public interface CountryRepository extends JpaRepository<Country, String> {

// JPQL Query

@Query("SELECT c FROM Country c WHERE c.name = ?1")

List<Country> findByNameJPQL(String name);

// HQL (same syntax, works in Hibernate)

@Query("FROM Country WHERE name = ?1")

List<Country> findByNameHQL(String name);

// Native Query

@Query(value = "SELECT \* FROM country WHERE name = ?1", nativeQuery = true)

List<Country> findByNameNative(String name);

// Count using Aggregate Function

@Query("SELECT COUNT(c) FROM Country c")

Long countCountries();

// Max population

@Query("SELECT MAX(c.population) FROM Country c")

Long getMaxPopulation();

}

### **10. Best Practices**

1. **Use JPQL for most cases** to maintain portability across JPA providers.
2. **Use JOIN FETCH** to optimize loading of lazy relationships.
3. **Use Native Queries** only when necessary (complex SQL, performance tuning).
4. **Avoid business logic in queries**. Keep them simple and focused.
5. **Use DTO projections** when not returning full entities.