# Spring Data JPA Queries

## Introduction

Spring Data JPA is a sub-project of Spring Data and provides an abstraction over the Data Access Layer using Java Persistence API, simplifying the interaction with databases in a Spring Boot application. It provides convenient ways to query and manipulate data using predefined query methods, derived queries, JPQL (Java Persistence Query Language), and native queries. Let's explore the different types of queries in Spring Data JPA:

## Derived Queries

Derived queries in Spring Boot provide a convenient way to define queries in your repositories by deriving the query from the method name. This feature is made possible by Spring Data JPA, a robust abstraction layer built on top of JPA (Java Persistence API), makes this feature possible. Here's how you can leverage derived queries in Spring Boot:

1. **Repository Definition**
    a. Start by defining an interface for your repository. Typically, you extend either the CrudRepository or JpaRepository interface, which provides basic CRUD (Create, Read, Update, Delete) operations.

       For Example,

```
public interface ItemDetailsRepository extends JpaRepository<ItemDetails, Integer>
{
     // Code to be written
}
```

In this example, ItemDetailsRepository extends JpaRepository and operates on the ItemDetails entity with a primary key of type integer.

## 2. Method Naming Convention

a. Derived queries rely on a naming convention to generate the SQL query. The method name should follow a specific pattern, where the query criteria are expressed using the method name keywords.

b. The general structure of a derived query method is *find**X**By**Y***, where **X** represents the type of object you want to retrieve, and **Y** denotes the property or field on which the query will be based.

c. You can also include additional keywords to express specific conditions or ordering requirements. Some commonly used keywords include And, Or, Between, LessThan, GreaterThan, OrderBy, and more.

d. Here are a few examples of derived queries:

```
public interface ItemDetailsRepository extends JpaRepository<ItemDetails, Integer>
{
    List<ItemDetails> findByPriceGreaterThan(double price);
    List<ItemDetails> findByDescriptionOrderByPriceDesc(String description);
    List<ItemDetails> findByCategoryAndBrand(String Category, String brand);

}
```

e. In the above examples, the queries will be generated based on the method names. For instance, findByDescriptionOrderByDesc will retrieve ItemDetails whose description matches the specified value and order them in descending order. Findbycategoryandbrand will retrieve ItemDetails whose Category and Brand match the specified values.

## 3. Execution and Results

a. When you invoke a derived query method, Spring Data JPA automatically generates the corresponding SQL query based on the method name and executes it on your behalf.

b. The return type of the query method should match the type you want to retrieve. In the examples above, we used List<ItemDetails>, so the result will be a list of ItemDetails objects.

c. You can include method parameters in the query method to filter the results further. Spring Data JPA will automatically map these parameters to the corresponding query conditions.
For example: The query below will fetch itemDetails with the given description and category.

```
List<ItemDetails> findByDescriptionAndCategory(String description, String category);
```

d. You can also use other keywords like Or to combine conditions, Between to specify a range, and more to create complex queries.

Derived queries offer a concise and expressive way to define queries in Spring Boot applications, reducing the need to write explicit SQL statements. Following the naming convention and leveraging method parameters, you can quickly build queries for various conditions and retrieve the desired data from your database.

## JPQL

JPQL (*Java Persistence Query Language*) is a query language specifically designed for working with entity objects in Java-based applications using the Java Persistence API (JPA). It is an object-oriented query language that allows you to perform database queries and manipulations in a database-agnostic way, i.e. it is independent of the type of database used. Here's a detailed description of JPQL and its key features:

### 1. Object-Orientation

a. JPQL operates on entity objects and their relationships rather than directly dealing with database tables and columns.
b. It allows you to write queries using your Java code's entity class names, properties, and relationships.

### 2. Syntax and Structure

a. JPQL queries resemble SQL queries, but they are written more object-oriented.

b. The basic structure of a JPQL query consists of a SELECT clause, an optional FROM clause, and optional WHERE, GROUP BY, HAVING, and ORDER BY clauses.

c. Here's an example of a simple JPQL query:
In the query below, Employee is the entity class name, e is an alias for the Employee entity, and the department is a named parameter.

```
SELECT e FROM Employee e WHERE e.department = :department
```

### 3. Entity and Property Names

a. JPQL uses entity class names and their properties to reference database tables and columns.

b. Entity class names are case-sensitive and should match the exact name of the entity class.

c. Property names in JPQL correspond to the fields or properties defined in the entity class.

4. **Named Parameters**
    a. JPQL supports named parameters that start with a colon (:) followed by a parameter name.

    b. Named parameters to pass dynamic values into queries, such as filtering criteria.

    c. Parameters can be used in the WHERE clause, ORDER BY clause, and other query parts.
    d. Here is an example of a JPQL query:

```
@Query("Select itd from ItemDetails itd where itd.category=?1   ORDEY BY itd.price DESC")
List<ItemDetails> findByCategoryOrderByPrice(String category);
```

5. **Querying Entities**
    a. JPQL allows you to retrieve entities based on various criteria using the SELECT and FROM clauses.

    b. JPQL allows you to retrieve entities based on various criteria using the SELECT and FROM clauses.

6. **Querying Relationships**
    a. JPQL supports navigating and querying relationships between entities.

    b. You can traverse relationships using dot notation, such as e.department.name, to access properties of related entities.

    c. Joins can be performed using the JOIN keyword to fetch related entities or perform complex queries involving multiple entities.

7. **Aggregation and Grouping**
    a. Using the GROUP BY and HAVING clauses, JPQL supports aggregations such as COUNT, SUM, AVG, MIN, and MAX.

    b. You can group entities based on specific properties and apply aggregate functions to those groups.

8. **Ordering Results**
    a. JPQL supports ordering query results using the ORDER BY clause.

    b. You can specify one or multiple properties by which the results should be sorted, along with ascending or descending order.

### 9. Integration with JPA

a. JPQL is tightly integrated with the Java Persistence API (JPA) and can be used with JPA providers such as Hibernate, EclipseLink, and OpenJPA.

b. JPA providers translate JPQL queries into the corresponding SQL statements specific to the underlying database.

JPQL provides a powerful and flexible way to query and manipulate entity objects in a database-agnostic manner. It allows you to express complex queries using familiar Java entity classes and their relationships. JPQL queries can be dynamically constructed and executed within JPA-based applications, making it a fundamental tool for database interactions in Java.

## Native Queries

Native queries in the context of Java Persistence API (JPA) allow you to execute SQL queries directly against the underlying database. Unlike JPQL (Java Persistence Query Language), which is a database-agnostic query language, native queries provide a way to work with database-specific SQL statements. This feature is useful when leveraging database-specific features or optimising performance for complex queries.

Here's a detailed description of native queries and their key features:

### 1. SQL-Based Queries

a. Native queries are written in SQL (Structured Query Language), the standard language for interacting with relational databases.

b. You can write SQL statements directly as strings to execute queries against the database.

c. SQL queries are specific to the database vendor and may not be portable across different systems.

d. Here is an example of SQL-based Native query,

```
@Query(value = “Select * from item where description like :desc LIMIT 4”, nativeQuery = true”)
List<Item> getItemByDesc(@Param(“desc”) String desc);
```

### 2. EntityManager Execution

a. In JPA, native queries are executed using the EntityManager interface, representing the persistence context and allowing interaction with the database.

b. The createNativeQuery method of EntityManager is used to create a native query object.

c. You can then set parameters, execute the query, and retrieve the results using methods provided by the native query object.

### 3. Parameter Binding

a. Native queries support parameter binding to safely pass dynamic values to the query.
b. Parameters can be positional or named. Positional parameters are denoted using question marks (?), while named parameters are prefixed with a colon (:).

c. Parameter values can be set using the setParameter or se*tParameter*erBy*Name* methods of the native query object.

### 4. Result Mapping

a. Native queries return results as database-specific result sets.

b. By default, the result set is returned as an array of objects, with each element representing a column value.

c. You can also map the result set to specific entity classes or use column aliases to map selected columns to specific properties.

### 5. Scalar Results

a. Native queries can retrieve scalar values, such as integers or strings, using aggregate functions or selecting specific columns.

b. Scalar results are returned as individual or arrays of values, depending on the query and mapping configuration.

### 6. Transaction Management

a. Native queries operate within the context of a JPA transaction.

b. If a transaction is inactive, JPA automatically starts a new transaction before executing the native query and commits or rolls back the transaction accordingly.

Native queries provide flexibility and direct access to the database, allowing you to leverage database-specific features and optimise performance when needed. However, it's important to note that native queries tie your code to a specific database, reducing portability across different database systems. Therefore, it's recommended to use native queries sparingly and favour JPQL for most database operations in JPA-based applications, as JPQL offers a more database-agnostic approach.

## Named and NamedNative Queries

Named and named native queries in JPA provide a way to define and reuse queries using a unique name instead of writing the query directly in code. Both named and native queries offer improved code readability, query management, and potential performance optimisations. Let's explore them in detail:

### Named Queries

1. Named queries are SQL-like queries written in JPQL (Java Persistence Query Language) and are database-agnostic.

2. They are defined and associated with an entity class using annotations or XML configuration files.

3. Named queries provide a level of abstraction over the underlying SQL statements and can be used across different database systems.

4. Benefits of named queries include improved maintainability, reusability, and the ability to externalise query definitions.

5. Here is an example of NamedQuery:

```
@Entity
@NamedQuery(name = “ItemDetails.findByCategoryOrderByPrice”
query = “Select itd from ItemDetails itd where itd.category=?1  ORDEY BY itd.price DESC”)
@Table(name = “item_details”)
public class ItemDetails {
    //...
}


public  class  ItemDetailsRepository  extends  JpaRepository<ItemDetails, Integer> {
    List<ItemDetails> findByCategoryOrderByPrice(String category);
}
```

## NamedNative Queries

1. Named native queries allow you to define and use database-specific SQL queries directly, leveraging the power of native SQL.

2. They are defined and associated with an entity class using annotations or XML configuration files.

3. Named native queries are useful when you need to utilise database-specific features or optimise performance with complex SQL statements.

4. However, they tie your code to a specific database system and are not portable across different systems.

5. Here is an example of NamedNative Query

```
@Entity
@NamedNativeQuery(name = “Item.getItemByDesc”
query = “Select * from item where description like CONCAT(?1, ‘%’) LIMIT
4”,
resultClass = Item.class)
@Table(name = “item_details”)
public class Item {
    //...
}

public class ItemRepository extends JpaRepository<ItemDetails, Integer>
{
   @Query(name = “Item.getItemByDesc”, nativeQuery=true)
   List<Item> getItemByDesc(String desc);
}
```

Named queries and named native queries provide a way to define queries separately from the code logic, offering better query management and code organisation. They can be easily reused, shared across multiple entities, and optimised by the JPA provider. However, it's important to note that named queries should be used judiciously, ensuring they align with your application's requirements and adhere to best practices.

## Conclusion

In conclusion, Spring Data JPA provides multiple options for querying and interacting with the database in a Spring Boot application. The choice of query approach depends on the complexity of the query, the need for database-specific features, and the level of control and flexibility required.

- **Derived queries** offer a convenient and concise way to define queries based on method names in the repository interface. They are automatically generated by Spring Data JPA based on the method names and parameter names, reducing the need for boilerplate code. Derived queries are suitable for simple queries and are easily maintained and understood.

- **JPQL (Java Persistence Query Language)** is a database-agnostic query language that allows you to write queries using entity classes, properties, and relationships. It provides a more expressive and powerful way to construct queries with support for filtering, sorting, and aggregations. JPQL is suitable for more complex queries and offers a good balance between readability and flexibility.

- **Native queries** enable you to write SQL queries directly and leverage database-specific features and optimisations. They provide the most control and flexibility over the query structure, allowing you to write complex queries and take advantage of advanced database capabilities. Native queries are helpful for scenarios requiring fine-tuned performance, database-specific functionalities, or complex data transformations.

When choosing the appropriate query approach, consider the trade-offs between simplicity, maintainability, portability, and performance. Derived queries are a good choice for common use cases, providing a simple and intuitive way to define queries. JPQL offers a more expressive and database-agnostic approach, balancing convenience and flexibility. Native queries should be used judiciously, primarily for advanced scenarios requiring fine-grained control over the SQL statements and database-specific features.

Spring Data JPA simplifies database operations, promotes code reuse, and improves productivity in developing data access layers in Spring Boot applications.

## Instructor Codes

- CNKart Application

## References


1. Official Documentation
2. Derived Queries
3. Derived Queries II
4. JPLQ
5. Joins using JPA
6. Native Queries
7. NamedNative Queries
8. Query Examples