

Creating Database Queries With the *@Query* Annotation

By Ramesh Fadatare (Java Guides)

Problem with Query Methods

- Keyword support - If the method name parser doesn't support the required keyword, we cannot use this strategy.
- The method names of complex query methods are long and ugly.

```
List<Product> findByDescriptionContainsOrNameContainsAllIgnoreCase(String description, String name);
```

- And this is for *just two* parameters. What happens when you want to create a query for 5 parameters?
- This is the point when you'll most likely want to prefer to write your own queries. This is doable via the @Query annotation.

Understanding @Query Annotation

- We can configure the invoked database query by annotating the query method with the @Query annotation.

```
// Define JPQL query with index parameters  
@Query("select p from Product p where p.name = ?1 or p.description = ?2")  
Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
```

- No need to follow query method naming conventions.
- We can use the @Query annotation in Spring Data JPA to execute both JPQL and native SQL queries

Spring Data JPA @Query annotation works

Define JPQL or Native SQL query using @Query annotation

```
// Define JPQL query with index parameters
@Query("select p from Product p where p.name = ?1 or p.description = ?2")
Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
```

Spring Data JPA provides required JPA code to execute the statement as a JPQL or native SQL query

Reduces boilerplate code

Hibernate will execute the query and map the result.

Your preferred JPA implementation, e.g., Hibernate or EclipseLink, will then execute the query and map the result.

Quick Overview of JPQL

JPQL stands for the Java Persistence Query Language. It is defined in the JPA specification and is an object-oriented query language used to perform database operations on persistent entities.

Hibernate, or any other JPA implementation, has to transform the JPQL query into SQL.

The syntax of a JPQL FROM clause is similar to SQL but uses the entity model instead of table or column names.

```
// Define JPQL query with index parameters
@Query("select p from Product p where p.name = ?1 or p.description = ?2")
Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
```

```
@Entity
@Table(name="products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "product_generator")
    @SequenceGenerator(name = "product_generator",
        sequenceName = "product_sequence_name",
        allocationSize = 1)

    @Column(name = "id")
    private Long id;

    @Column(name = "sku")
    private String sku;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private BigDecimal price;
```


JPQL Features

- It is a platform-independent query language.
- It is simple and robust.
- It can be used with any type of relation database.
- It can be declared statically into metadata or can also be dynamically built in code.
- It is case insensitive.

If your database can change or varies from development to production, as long as they're both relational - JPQL works wonders and you can write JPQL queries to create generic logic that can be used over and over again.

Creating JPQL Queries

Steps to create JPQL query with the @Query annotation:


Step 2: Annotate the query method with the @Query annotation, and specify the invoked query by setting it as the value of the @Query annotation.

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    // Define JPQL query with index parameters  
    @Query("select p from Product p where p.name = ?1 or p.description = ?2")  
    Product findByNameOrDescriptionJPQLIndexParam(String name, String description);  
}
```

Step 1: Add a query method to our repository interface.

JPQL Query with Index (Position) Parameters

When using position-based parameters, you have to keep track of the order in which you supply the parameters in:

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    // Define JPQL query with index parameters  
    @Query("select p from Product p where p.name = ?1 or p.description = ?2")  
     Product findByNameOrDescriptionJPQLIndexParam(String name, String description);  
}
```

The first parameter passed to the method is mapped to ?1, the second is mapped to ?2, etc. If you accidentally switch these up - your query will likely throw an exception, or silently produce wrong results.

JPQL Query with Named Parameters

Named parameters can be referenced by name, no matter their position:

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    // Define JPQL query with named parameters  
    @Query("select p from Product p where p.name =:name or p.description =:description")  
    Product findByNameOrDescriptionJPQLNamedParam(@Param("name") String name,  
                                                    @Param("description") String description);  
}
```

The name within the @Param annotation is matched to the named parameters in the @Query annotation, so you're free to call your variables however you'd like - but for consistency's sake - it's advised to use the same name.

Creating Native SQL Queries

Steps to create Native SQL query with the @Query annotation:

Step 2: Annotate the query method with the @Query annotation, and specify the invoked query by setting it as the value of the @Query annotation.

Step 3: Set the value of the @Query annotation's nativeQuery attribute to true

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    // Define Native SQL query with index parameters  
    @Query(value = "select * from products p where p.name =?1 " +  
        "or p.description =?2", nativeQuery = true)  
    Product findByNameOrDescriptionSQLIndexParam(String name, String description);  
}
```

Step 1: Add a query method to our repository interface.

Native SQL Query with Index (Position) Parameters

When using position-based parameters, you have to keep track of the order in which you supply the parameters in:

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    // Define Native SQL query with index parameters  
    @Query(value = "select * from products p where p.name =?1 " +  
        "or p.description =?2", nativeQuery = true)  
    Product findByNameOrDescriptionSQLIndexParam(String name, String description);  
}
```

The first parameter passed to the method is mapped to ?1, the second is mapped to ?2, etc. If you accidentally switch these up - your query will likely throw an exception, or silently produce wrong results.

Native SQL Query with Named Parameters

Named parameters can be referenced by name, no matter their position:

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
    // Define Native SQL query with named parameters  
    @Query(value = "select * from products p where p.name =:name " +  
        "or p.description =:description", nativeQuery = true)  
    Product findByNameOrDescriptionSQLNamedParam(@Param("name") String name,  
        @Param("description") String description);  
}
```

The name within the `@Param` annotation is matched to the named parameters in the `@Query` annotation, so you're free to call your variables however you'd like - but for consistency's sake - it's advised to use the same name.