Spring Data JPA @Query

Last modified: December 30, 2020

by baeldung (https://www.baeldung.com/author/baeldung/)

Spring Data (https://www.baeldung.com/category/persistence/spring-persistence/spring-data/)

JPA (https://www.baeldung.com/tag/jpa/)

1. Overview

Spring Data provides many ways to define a query that we can execute. One of these is the *@Query* annotation.

In this tutorial, we'll demonstrate **how to use the *@Query* annotation in Spring Data JPA to execute both JPQL and native SQL queries.**

We'll also show how to build a dynamic query when the *@Query* annotation is not enough.

Further reading:

Derived Query Methods in Spring Data JPA Repositories (/spring-data-derived-queries)

Explore the query derivation mechanism in Spring Data JPA.

Read more (/spring-data-derived-queries) →

Spring Data JPA @Modifying Annotation (/spring-data-jpa-modifying-annotation)

Create DML and DDL queries in Spring Data JPA by combining the @Query and @Modifying annotations

Read more (/spring-data-jpa-modifying-annotation) →

2. Select Query

In order to define SQL to execute for a Spring Data repository method, we can **annotate the method with the *@Query* annotation — its *value* attribute contains the JPQL or SQL to execute.**

The *@Query* annotation takes precedence over named queries, which are annotated with *@NamedQuery* or defined in an *orm.xml* file.

It's a good approach to place a query definition just above the method inside the repository rather than inside our domain model as named queries. The repository is responsible for persistence, so it's a better place to store these definitions.

2.1. JPQL

By default, the query definition uses JPQL.

Let's look at a simple repository method that returns active *User* entities from the database:

```
@Query("SELECT u FROM User u WHERE u.status = 1")
Collection<User> findAllActiveUsers();
```

2.2. Native

We can use also native SQL to define our query. All we have to do is **set the value of the *nativeQuery* attribute to *true*** and define the native SQL query in the *value* attribute of the annotation:

```
@Query(
  value = "SELECT * FROM USERS u WHERE u.status = 1",
  nativeQuery = true)
Collection<User> findAllActiveUsersNative();
```

3. Define Order in a Query

We can pass an additional parameter of type *Sort* to a Spring Data method declaration that has the *@Query* annotation. It'll be translated into the *ORDER BY* clause that gets passed to the database.

3.1. Sorting for JPA Provided and Derived Methods

For the methods we get out of the box such as *findAll(Sort)* or the ones that are generated by parsing method signatures, **we can only use object properties to define our sort**:

```
userRepository.findAll(Sort.by(Sort.Direction.ASC, "name"));
```

Now imagine that we want to sort by the length of a name property:

```
userRepository.findAll(Sort.by("LENGTH(name)"));
```

When we execute the above code, we'll receive an exception:

> org.springframework.data.mapping.PropertyReferenceException: No property LENGTH(name) found for type User!

3.2. JPQL

When we use JPQL for a query definition, then Spring Data can handle sorting without any problem — all we have to do is add a method parameter of type *Sort*:

```
@Query(value = "SELECT u FROM User u")
List<User> findAllUsers(Sort sort);
```

We can call this method and pass a *Sort* parameter, which will order the result by the *name* property of the *User* object:

```
userRepository.findAllUsers(Sort.by("name"));
```

And because we used the *@Query* annotation, we can use the same method to get the sorted list of *Users* by the length of their names:

```
userRepository.findAllUsers(JpaSort.unsafe("LENGTH(name)"));
```

It's crucial that we use *JpaSort.unsafe()* to create a *Sort* object instance.

When we use:

```
Sort.by("LENGTH(name)");
```

then we'll receive exactly the same exception as we saw above for the *findAll()* method.

When Spring Data discovers the unsafe *Sort* order for a method that uses the *@Query* annotation, then it just appends the sort clause to the query — it skips checking whether the property to sort by belongs to the domain model.

3.3. Native

When the *@Query* annotation uses native SQL, then it's not possible to define a *Sort*.

If we do, we'll receive an exception:

> org.springframework.data.jpa.repository.query.InvalidJpaQueryMethodException: Cannot use native queries with dynamic sorting and/or pagination

As the exception says, the sort isn't supported for native queries. The error message gives us a hint that pagination will cause an exception too.

However, there is a workaround that enables pagination, and we'll cover it in the next section.

4. Pagination

Pagination allows us to return just a subset of a whole result in a *Page*. This is useful, for example, when navigating through several pages of data on a web page.

Another advantage of pagination is that the amount of data sent from server to client is minimized. By sending smaller pieces of data, we can generally see an improvement in performance.

4.1. JPQL

Using pagination in the JPQL query definition is straightforward:

```
@Query(value = "SELECT u FROM User u ORDER BY id")
Page<User> findAllUsersWithPagination(Pageable pageable);
```

We can pass a *PageRequest* parameter to get a page of data.

Pagination is also supported for native queries but requires a little bit of additional work.

4.2. Native

We can **enable pagination for native queries by declaring an additional attribute *countQuery*.**

This defines the SQL to execute to count the number of rows in the whole result:

```
@Query(
  value = "SELECT * FROM Users ORDER BY id",
  countQuery = "SELECT count(*) FROM Users",
  nativeQuery = true)
Page<User> findAllUsersWithPagination(Pageable pageable);
```

4.3. Spring Data JPA Versions Prior to 2.0.4

The above solution for native queries works fine for Spring Data JPA versions 2.0.4 and later.

Prior to that version, when we try to execute such a query, we'll receive the same exception we described in the previous section on sorting.

We can overcome this by adding an additional parameter for pagination inside our query:

```
@Query(
  value = "SELECT * FROM Users ORDER BY id \n-- #pageable\n",
  countQuery = "SELECT count(*) FROM Users",
  nativeQuery = true)
Page<User> findAllUsersWithPagination(Pageable pageable);
```

In the above example, we add "\n– #pageable\n" as the placeholder for the pagination parameter. This tells Spring Data JPA how to parse the query and inject the pageable parameter. This solution works for the *H2* database.

We've covered how to create simple select queries via JPQL and native SQL. Next, we'll show how to define additional parameters.

5. Indexed Query Parameters

There are two possible ways that we can pass method parameters to our query: indexed and named parameters.

In this section, we'll cover indexed parameters.

5.1. JPQL

For indexed parameters in JPQL, Spring Data will **pass method parameters to the query in the same order they appear in the method declaration**:

```
@Query("SELECT u FROM User u WHERE u.status = ?1")
User findUserByStatus(Integer status);

@Query("SELECT u FROM User u WHERE u.status = ?1 and u.name = ?2")
User findUserByStatusAndName(Integer status, String name);
```

For the above queries, the *status* method parameter will be assigned to the query parameter with index *1*, and the *name* method parameter will be assigned to the query parameter with index *2*.

5.2. Native

Indexed parameters for the native queries work exactly in the same way as for JPQL:

```
@Query(
  value = "SELECT * FROM Users u WHERE u.status = ?1",
  nativeQuery = true)
User findUserByStatusNative(Integer status);
```

In the next section, we'll show a different approach: passing parameters via name.

6. Named Parameters

We can also **pass method parameters to the query using named parameters.** We define these using the *@Param* annotation inside our repository method declaration.

Each parameter annotated with *@Param* must have a value string matching the corresponding JPQL or SQL query parameter name. A query with named parameters is easier to read and is less error-prone in case the query needs to be refactored.

6.1. JPQL

As mentioned above, we use the *@Param* annotation in the method declaration to match parameters defined by name in JPQL with parameters from the method declaration:

```
@Query("SELECT u FROM User u WHERE u.status = :status and u.name =
:name")
User findUserByStatusAndNameNamedParams(
  @Param("status") Integer status,
  @Param("name") String name);
```

Note that in the above example, we defined our SQL query and method parameters to have the same names, but it's not required as long as the value strings are the same:

```
@Query("SELECT u FROM User u WHERE u.status = :status and u.name =
:name")
User findUserByUserStatusAndUserName(@Param("status") Integer
userStatus,
  @Param("name") String userName);
```

6.2. Native

For the native query definition, there is no difference in how we pass a parameter via the name to the query in comparison to JPQL — we use the *@Param* annotation:

```
@Query(value = "SELECT * FROM Users u WHERE u.status = :status and
u.name = :name",
  nativeQuery = true)
User findUserByStatusAndNameNamedParamsNative(
  @Param("status") Integer status, @Param("name") String name);
```

7. Collection Parameter

Let's consider the case when the *where* clause of our JPQL or SQL query contains the *IN* (or *NOT IN*) keyword:

```
SELECT u FROM User u WHERE u.name IN :names
```

In this case, we can define a query method that takes *Collection* as a parameter:

```
@Query(value = "SELECT u FROM User u WHERE u.name IN :names")
List<User> findUserByNameList(@Param("names") Collection<String> names);
```

As the parameter is a *Collection*, it can be used with *List, HashSet*, etc.

Next, we'll show how to modify data with the @*Modifying* annotation.

8. Update Queries With *@Modifying*

We can **use the @*Query* annotation to modify the state of the database by also adding the @*Modifying* annotation** to the repository method.

8.1. JPQL

The repository method that modifies the data has two differences in comparison to the *select* query — it has the *@Modifying* annotation and, of course, the JPQL query uses *update* instead of *select*:

```
@Modifying
@Query("update User u set u.status = :status where u.name = :name")
int updateUserSetStatusForName(@Param("status") Integer status,
  @Param("name") String name);
```

The return value defines how many rows the execution of the query updated. Both indexed and named parameters can be used inside update queries.

8.2. Native

We can modify the state of the database also with a native query. We just need to add the *@Modifying* annotation:

```
@Modifying
@Query(value = "update Users u set u.status = ? where u.name = ?",
  nativeQuery = true)
int updateUserSetStatusForNameNative(Integer status, String name);
```

8.3. Inserts

To perform an insert operation, we have to both apply *@Modifying* and use a native query since INSERT is not a part of the JPA interface (/jpa-insert):

```
@Modifying
@Query(
  value =
    "insert into Users (name, age, email, status) values (:name, :age,
:email, :status)",
  nativeQuery = true)
void insertUser(@Param("name") String name, @Param("age") Integer age,
  @Param("status") Integer status, @Param("email") String email);
```

9. Dynamic Query

Often, we'll encounter the need for building SQL statements based on conditions or data sets whose values are only known at runtime. And in those cases, we can't just use a static query.

9.1. Example of a Dynamic Query

For example, let's imagine a situation where we need to select all the users whose email is *LIKE* one from a set defined at runtime — *email1*, *email2*, …, *emailn*:

```
SELECT u FROM User u WHERE u.email LIKE '%email1%'
    or  u.email LIKE '%email2%'
    ...
    or  u.email LIKE '%emailn%'
```

Since the set is dynamically constructed, we can't know at compile-time how many *LIKE* clauses to add.

In this case, **we can't just use the *@Query* annotation since we can't provide a static SQL statement.**

Instead, by implementing a custom composite repository, we can extend the base *JpaRepository* functionality and provide our own logic for building a dynamic query. Let's take a look at how to do this.

9.2. Custom Repositories and the JPA Criteria API

Luckily for us, **Spring provides a way for extending the base repository through the use of custom fragment interfaces.** We can then link them together to create a composite repository (/spring-data-composable-repositories).

We'll start by creating a custom fragment interface:

```
public interface UserRepositoryCustom {
    List<User> findUserByEmails(Set<String> emails);
}
```

And then we'll implement it:

```
public class UserRepositoryCustomImpl implements UserRepositoryCustom {

    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<User> findUserByEmails(Set<String> emails) {
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<User> query = cb.createQuery(User.class);
        Root<User> user = query.from(User.class);

        Path<String> emailPath = user.get("email");

        List<Predicate> predicates = new ArrayList<>();
        for (String email : emails) {
            predicates.add(cb.like(emailPath, email));
        }
        query.select(user)
            .where(cb.or(predicates.toArray(new
Predicate[predicates.size()])));

        return entityManager.createQuery(query)
            .getResultList();
    }
}
```

As shown above, we leveraged the JPA Criteria API (/hibernate-criteria-queries) to build our dynamic query.

Also, we need to make sure to include the *Impl* postfix in the class name. Spring will search the *UserRepositoryCustom* implementation as *UserRepositoryCustomImpl*. Since fragments are not repositories by

themselves, Spring relies on this mechanism to find the fragment implementation.

9.3. Extending the Existing Repository

Notice that all the query methods from section 2 through section 7 are in the *UserRepository*.

So, now we'll integrate our fragment by extending the new interface in the *UserRepository*:

```
public interface UserRepository extends JpaRepository<User, Integer>,
UserRepositoryCustom {
    //  query methods from section 2 - section 7
}
```

9.4. Using the Repository

And finally, we can call our dynamic query method:

```
Set<String> emails = new HashSet<>();
// filling the set with any number of items

userRepository.findUserByEmails(emails);
```

We've successfully created a composite repository and called our custom method.

10. Conclusion

In this article, we covered several ways of defining queries in Spring Data JPA repository methods using the *@Query* annotation.

We also learned how to implement a custom repository and create a dynamic query.

As always, the complete code examples used in this article are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-jpa-query-2).