Unknown date | Written by: baeldung

Derived Query Methods in Spring Data JPA Repositories

1. Overview🔗

For simple queries, it's easy to derive what the query should be **just by looking at the corresponding method name in our code.**

In this tutorial, we'll explore how Spring Data JPA leverages this idea in the form of a method naming convention.

Further reading:

Introduction to Spring Data JPA

Introduction to Spring Data JPA with Spring 4 - the Spring config, the DAO, manual and generated queries and transaction management.
Read more →

CrudRepository, JpaRepository, and PagingAndSortingRepository in Spring Data

Learn about the different flavours of repositories offered by Spring Data.
Read more →

Sorting Query Results with Spring Data

Learn different ways to sort results in Spring Data queries.
Read more →

2. Structure of Derived Query Methods in Spring🔗

Derived method names have two main parts separated by the first *By* keyword:

```
List<User> findByName(String name)
```
Copy

The first part — such as *find* — is the *introducer*, and the rest — such as *ByName* — is the *criteria*.

Spring Data JPA supports *find*, *read*, *query*, *count* and *get*. So, we could have done *queryByName*, and Spring Data would behave the same.

We can also use *Distinct*, *First* or *Top* to remove duplicates or limit our result set:

```
List<User> findTop3ByAge()
```
Copy

The criteria part contains the entity-specific condition expressions of the query. We can use the condition keywords along with the entity's property names.

We can also concatenate the expressions with *And* and *Or*, as we'll see in just a moment.

3. Sample Application

First, we'll of course need an application using Spring Data JPA.

In that application, let's define an entity class:

```
@Table(name = "users")
@Entity
class User {
    @Id
    @GeneratedValue
    private Integer id;

    private String name;
    private Integer age;
    private ZonedDateTime birthDate;
    private Boolean active;
    // standard getters and setters
}
```
Copy

Let’s also define a repository.

It’ll extend *JpaRepository*, one of the Spring Data Repository types:

```
interface UserRepository extends JpaRepository<User, Integer> {
```

This is where we’ll place all our derived query methods.

4. Equality Condition Keywords

Exact equality is one of the most-used conditions in queries. We have several options to express = or IS operators in the query.

We can just append the property name without any keyword for an exact match condition:

```
List<User> findByName(String name);
```
Copy

And we can add *Is* or *Equals* for readability:

```
List<User> findByNameIs(String name);
List<User> findByNameEquals(String name);
```
Copy

This extra readability comes in handy when we need to express inequality instead:

```
List<User> findByNameIsNot(String name);
```
Copy

This is quite a bit more readable than *findByNameNot(String)*!

As *null* equality is a special case, we shouldn’t use the = operator. Spring Data JPA handles *null* parameters by default. So, when we pass a *null* value for an equality condition, Spring interprets the query as IS NULL in the generated SQL.

We can also use the *IsNull* keyword to add IS NULL criteria to the query:

```
List<User> findByNameIsNull();
List<User> findByNameIsNotNull();
```

Note that neither *IsNull* nor *IsNotNull* requires a method argument.

There are also two more keywords that don’t require any arguments.

We can use *True* and *False* keywords to add equality conditions for *boolean* types:

```
List<User> findByActiveTrue();
List<User> findByActiveFalse();
```

Of course, we sometimes want something more lenient than exact equality, so let’s see what else we can do.

5. Similarity Condition Keywords

When we need to query the results with a pattern of a property, we have a few options.

We can find names that start with a value using *StartingWith*:

```
List<User> findByNameStartingWith(String prefix);
```

Roughly, this translates to “WHERE *name* LIKE *‘value%’*“.

If we want names that end with a value, *EndingWith* is what we want:

```
List<User> findByNameEndingWith(String suffix);
```

Or we can find which names contain a value with *Containing*:

```
List<User> findByNameContaining(String infix);
```

Note that all conditions above are called predefined pattern expressions. So, **we don’t need to add % operator inside the argument** when these methods are called.

But let’s suppose we are doing something more complex. Say we need to fetch the users whose names start with an *a*, contain *b* and end with *c*.

```
List<User> findByNameLike(String likePattern);
```

And we can then hand in our LIKE pattern when we call the method:

```
String likePattern = "a%b%c";
userRepository.findByNameLike(likePattern);
```

That’s enough about names for now. Let’s try some other values in *User*.

6. Comparison Condition Keywords

Furthermore, we can use *LessThan* and *LessThanEqual* keywords to compare the records with the given value using the < and <= operators:

```
List<User> findByAgeLessThan(Integer age);
List<User> findByAgeLessThanEqual(Integer age);
```

In the opposite situation, we can use *GreaterThan* and *GreaterThanEqual* keywords:

```
List<User> findByAgeGreaterThan(Integer age);
List<User> findByAgeGreaterThanEqual(Integer age);
```

Or we can find users who are between two ages with *Between*:

```
List<User> findByAgeBetween(Integer startAge, Integer endAge);
```

We can also supply a collection of ages to match against using *In*:

```
List<User> findByAgeIn(Collection<Integer> ages);
```

Since we know the users’ birthdates, we might want to query for users who were born before or after a given date.

We’d use *Before* and *After* for that:

```
List<User> findByBirthDateAfter(ZonedDateTime birthDate);
List<User> findByBirthDateBefore(ZonedDateTime birthDate);
```

7. Multiple Condition Expressions

We can combine as many expressions as we need by using *And* and *Or* keywords:

```
List<User> findByNameOrAge(String name, Integer age);
List<User> findByNameOrAgeAndActive(String name, Integer age, B
```

The precedence order is *And* then *Or*, just like Java.

While Spring Data JPA imposes no limit to how many expressions we can add, we shouldn't go crazy here. Long names are unreadable and hard to maintain. For complex queries, take a look at **the *@Query* annotation instead.**

8. Sorting the Results

Next, let's look at sorting.

We could ask that the users be sorted alphabetically by their name using *OrderBy*:

```
List<User> findByNameOrderByName(String name);
List<User> findByNameOrderByNameAsc(String name);
```

Copy

Ascending order is the default sorting option, but we can use *Desc* instead to sort them in reverse:

```
List<User> findByNameOrderByNameDesc(String name);
```

Copy

9. *findOne* vs *findById* in a *CrudRepository*

The Spring team made some major changes in *CrudRepository* with Spring Boot 2.x. One of them is renaming *findOne* to *findById*.

Previously with Spring Boot 1.x, we'd call *findOne* when we wanted to retrieve an entity by its primary key:

```
User user = userRepository.findOne(1);
```

Copy

```
User user = userRepository.findById(1);
```
Copy

Note that the *findById()* method is already defined in *CrudRepository* for us. So, we don't have to define it explicitly in custom repositories that extend *CrudRepository*.

10. Conclusion

In this article, we explained the query derivation mechanism in Spring Data JPA. We used the property condition keywords to write derived query methods in Spring Data JPA repositories.

The source code of this article is available in the GitHub project.
