Spring Data often gets praised for its derived query feature. As long as your method name starts with find...By, read...By, query...By, count...By, or get...By and follows the right pattern, Spring Data generates the required JPQL query.

That might sound like you will never need to write your own queries again. But that's not the case. It's a great way to define simple queries. But as soon as you need to use more than 2 query parameters or your query gets at least a little bit complex, you should use a custom query. That's either because the query name gets really complicated to write and read or because you exceed the capabilities of the method name parser.

Creating a simple derived query with parameters

The definition of a derived query is pretty simple and self-explaining. I started the name of the method with findBy and then referenced the entity attributes for which I want to filter in the WHERE clause by its name. And then I define a method parameter with the same name as the referenced entity attribute.

```
public interface AuthorRepository extends JpaRepository<Author,
Long> {


    List<Author> findByFirstName(String firstName);


}
```

Derived queries with multiple parameters

You can extend this method to search for Author entities with a given firstName and lastName by combining them with And. Spring Data JPA, of course, also allows you to concatenate multiple checks using an Or clause.

```
public interface AuthorRepository extends JpaRepository<Author,
Long> {


    List<Author> findByFirstNameAndLastName(String firstName,
String lastName);


}
```

Traverse associations in derived queries

If you want to filter for an attribute of an associated entity, you can traverse managed relationships by referencing the attribute that maps the association followed by the attribute of the related entity.

```
public interface AuthorRepository extends JpaRepository<Author,
Long> {


    List<Author> findByBooksTitle(String title);


}
```

Other comparison operators

If you just reference an entity attribute in your method name, Spring Data JPA will generate a simple equals comparison. You can also specify different comparison operations by using one of the following keywords together with the name of your entity attribute:

- Like – to check if the value of an entity is like a provided String.
- Containing – to check if the value of an entity attribute contains the provided String.
- IgnoreCase – to ignore the case when comparing the value of an entity attribute with a provided String.

- Between – to check if the value of an entity attribute is between 2 provided values.
- LessThan / GreaterThan – to check if the value of an entity attribute is less or greater then a provided one.

Order the results of a derived query

You can, of course, also order your query results. In JPQL, this would require an ORDER BY clause in your query. With Spring Data JPA, you just need to add the words OrderBy to your query followed by the name of the entity attribute and the abbreviations ASC or DESC for your preferred order.

```
public interface BookRepository extends JpaRepository<Book, Long>
{

    List<Book> findByTitleContainsOrderByTitleAsc(String title);

}
```

If you require dynamic ordering, you can add a parameter of type Sort to your query method. This is one of the special parameters supported by Spring Data JPA, and it triggers the generation of an ORDER BY clause.

```
public interface BookRepository extends JpaRepository<Book, Long>
{

    List<Book> findByTitleContains(String title, Sort sort);

}
```

Limiting the number of results

Using Hibernate or any other JPA implementation, you can limit the number of returned records on the Query interface. With Spring Data JPA, you can do the same by adding the keywords Top or First followed by a number between the find and By keywords.

```
public interface BookRepository extends JpaRepository<Book, Long>
{

    List<Book> findFirst5ByTitleOrderByTitleAsc(String title);

}
```

Paginate the results of a derived query

And after we had a look at ordering and limiting the number of returned records, we also need to talk about pagination. Spring Data JPA provides another special parameter for it. You just need to add a parameter of type Pageable to your query method definition and change the return type to Page<YourEntity>.

```
public interface BookRepository extends JpaRepository<Book, Long>
{

    Page<Book> findAll(Pageable pageable);

}
```