Derived queries are very comfortable to use as long as the queries are not too complicated. But as soon as you use more than 2-3 query parameters or need to define multiple joins to other entities, you need a more flexible approach. In these situations, you better use Spring Data JPA's @Query annotation to specify a custom JPQL or native SQL query.

The @Query annotation gives you full flexibility over the executed statement, and your message name doesn't need to follow any conventions. The only thing you need to do is to define a method in your repository interface, annotate it with @Query, and provide the statement that you want to execute.

Spring Data JPA provides the required JPA code to execute the statement as a JPQL or native SQL query. Your preferred JPA implementation, e.g., Hibernate or EclipseLink, will then execute the query and map the result.

JPQL Queries

When you define a JPQL query in your repository definition, Spring Data JPA only provides the required JPA code to execute that query. The query itself gets processed and executed by your JPA implementation. So, you can still use everything you learned about JPQL queries with Hibernate or EclipseLink with Spring Data JPA. It just gets a little bit easier.

Defining a Custom JPQL Query

Creating a JPQL query with Spring Data JPA's @Query annotation is pretty straightforward. You need to annotate a method on your repository interface with the @Query annotation and provide a String with the JPQL query statement.

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

    List<Author> findByFirstName(String firstName);

}
```

As you can see in the code snippet, both methods return a List of Author entities. But their SELECT clauses look very different.

The findByFirstName method doesn't define any SELECT clause, and the findByFirstNameAndLastName method uses the Author entity to specify the projection.

In the end, Spring Data JPA uses the same projection for both queries. The SELECT clause of the findByFirstName query gets automatically generated so that it selects all columns mapped by the entities referenced in the FROM clause. In this example, these are all columns mapped by the Author entity.

Sorting Your Query Results

For some use cases, you might want to retrieve the query result in a specific order. Using Spring Data JPA, you can define the sorting of your query results in 2 ways:

1. You can add an ORDER BY clause to your JPQL query or
2. You can add a parameter of type Sort to your method signature.

Using the ORDER BY Clause in JPQL

You're probably familiar with the first option. The ORDER BY clause is defined in the JPA standard, and it is very similar to the ORDER BY clause you know from SQL. You simply reference one or more entity attributes and use ASC or DESC to specify if you want to sort them in ascending or descending order.

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

    @Query("FROM Author WHERE firstName = ?1 ORDER BY lastName ASC")
    List<Author> findByFirstNameOrderByLastname(String firstName);
}
```

Using a Sort Object

With Spring Data JPA, you can also add a parameter of type Sort to your method definition. Spring Data JPA will then generate the required ORDER BY clause. That is the same approach as you can use in a derived query.

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

    @Query("FROM Author WHERE firstName = ?1")
    List<Author> findByFirstName(String firstName, Sort sort);
}
```

When you want to call the findAuthors method, you need to provide a Sort object. Its constructor expects an enumeration that defines the sorting direction and one or more Strings that reference the entity attributes, which you want to use to order the result, by their name.

```
Sort sort = new Sort(Direction.ASC, "firstName");
List<Author> authors = authorRepository.findByFirstName("Thorben", sort);
```

Paginating Your Query Results

In addition to sorting, Spring Data JPA also provides very comfortable support for pagination. If you add a method parameter of type Pageable to your repository method, Spring Data JPA generates the required code to handle the pagination of the query result.

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

    @Query("FROM Author WHERE firstName = ?1")
    List<Author> findByFirstName(String firstName, Pageable pageable);
}
```

When you call the findByFirstName method, you need to provide an object that implements the Pageable interface. You can do that by calling the of method on the PageRequest method with the number of the page and the size of each page as parameters.

```
Pageable pageable = PageRequest.of(0, 10);
List<Author> authors = authorRepository.findByFirstName("Thorben", pageable);
```

SpEL Expressions for Entity Names and Advanced Like Expressions

In addition to the previously discussed query features, Spring Data JPA also supports SpEL expressions within your query. You can use it to avoid hard references to the name of an entity or to create advanced like expressions.

Avoid Entity Name References

Referencing entities by their name prevents you from defining queries for generic repositories. It also causes a lot of work if you decide to rename an entity because you then need to update all queries that reference the old name of the entity.

You can avoid that by using the entityName variable in a SpEL expression. I do that in the following query to avoid referencing the Author entity by its name.

```
public interface AuthorRepository extends JpaRepository<Author, Long> {
    @Query("FROM #{#entityName} WHERE firstName = ?1")
    List findByFirstName(String firstName);
}
```

Spring Data JPA replaces the #{#entityName} expression with the entityName of the domain type of the repository. So, in this example, Spring Data JPA replaces #{#entityName} with Author.

Define Advanced Like Expressions

Another great feature that you get with the SpEL expressions is the definition of advanced like expressions. You can, for example, append '%' to the beginning and end of a parameter and change the provided bind parameter value to upper case.

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


    @Query("FROM Author WHERE UPPER(firstName) LIKE
%?#{[0].toUpperCase()}%")
    List<Author> findByFirstNameContainingIgnoreCase(String firstName);
}
```

When you call this method, Spring Data JPA calls the toUpperCase() method on the provided method parameter and adds a '%' to the beginning and end of it. In the next step, it sets the modified String as a bind parameter value.