Derived Queries

A derived query in Spring Data JPA refers to a query method automatically generated by the framework based on the method name. It allows developers to define data retrieval operations concisely and readably without writing explicit JPQL (Java Persistence Query Language) or SQL queries. Derived queries are especially useful for common and straightforward operations.

Here are the critical aspects of derived queries in detail:

1. Method Naming Conventions: Derived queries are created by following a specific naming convention in the repository interface. The method name consists of a combination of keywords that convey the intended action and the properties of the entity class involved.

2. Keywords: Keywords such as ***findBy***, ***findAllBy***, ***readBy***, ***getBy***, and logical operators like ***And***, ***Or***, and ***Not*** are used to construct the method names. These keywords indicate the operation and the criteria to be applied to the query.

3. Property Expressions: After the keywords, the properties of the entity class are referred to using their camel-case names. For example, if you have an Employee entity with a property named firstName, the method findByFirstName would be generated.

4. Comparison Keywords: Comparison keywords like Equals, IsNull, Like, GreaterThan, LessThan, and more can be used to specify conditions for filtering data.

5. Logical Operators: Logical operators like And and Or allow you to combine conditions within the query method, providing more advanced filtering capabilities.

6. Ordering and Limiting: Derived queries can include ordering (using OrderBy) and limiting the number of results (using First, Top, Distinct, and others).

7. Negation: The Not keyword allows the negation of a condition. For instance, findByActiveIsNotTrue fetches records where the active property is not true.

8. Multiple Parameters: Derived queries can accept multiple parameters, allowing you to specify various criteria for filtering data.

9. Null Handling: Derived queries handle null values automatically. For example, findByEmailIsNull retrieves records where the email is null.

10. Derived Query Limitations: While derived queries are powerful for simple and common queries, they might not cover complex scenarios that require dynamic conditions, joins, subqueries, or custom calculations. You can use @Query annotations with custom JPQL or SQL queries in such cases.

Let's see derived query examples with the help of a scenario.

Scenario: You have a **"Company"** database with a table: **"Employee"**. The **"Employee"** table contains employee information, such as their ID, first name, last name, salary, age etc.

id	first_name	last_name	age	email	salary
1	Ramesh	Verma	25	rv@gmail.com	20000
2	Mahesh	Anand	26	ma@gmail.com	25000

Employee Table

1. To retrieve all employees with the first name **"John."**

```
List<Employee> findByFirstName("John");
```

2. To retrieve employees with a last name "Doe" and age greater than or equal to 30:

```
List<Employee> findByAgeGreaterThanEqualAndLastName(30,"Doe");
```

3. To retrieve employees whose email contains "example.com":

```
List<Employee> findByEmailLike("%yahoo.com%");
```

4. To retrieve employees with a salary between 40000 and 60000:

```
List<Employee> findBySalaryBetween(40000, 60000);
```

5. To retrieve all employees ordered by their joining date in descending order:

```
List<Employee> findAllByOrderBySalaryDateDesc();
```

6. To retrieve employees with a salary greater than a certain amount, ordered by salary in ascending order:

```
List<Employee> findBySalaryGreaterThanOrderBySalaryAsc(double salary);
```