Class-25: Query Methods & JPQL

by Pial Kanti Samadder

Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

- By deriving the query from the method name directly.
- By using a manually defined query.

Available options depend on the actual store. However, there must be a strategy that decides what actual query is created.

Query Lookup Strategies

Spring has *three strategies* to determine how queries are created.

Strategy	What It Does	Behavior	Use Case
CREATE	Builds query *from the method name*	`findByEmail` → `SELECT ... WHERE email = ?`	When you want everything auto-generated
USE_DECLARED_QUERY	Requires query to be **explicitly defined**	If no query is defined → **exception**	When you want full control using `@Query`
CREATE_IF_NOT_FOUND *(default)*	First checks if a query is declared, otherwise generates from method name		

You can use the `queryLookupStrategy` attribute of the `EnableJpaRepositories` annotation.

```
@EnableJpaRepositories(
    basePackages = "com.yourapp.repository",
    queryLookupStrategy = QueryLookupStrategy.Key.CREATE_IF_NOT_FOUND // default
)
public class JpaConfig {
}
```

Query Creation

- Spring Data can **automatically create queries** based on repository **method names**.
- A derived query method has **two parts**:
  - **Subject (Intro)** → `find`, `read`, `get`, `count`, `exists`
  - **Predicate (Criteria)** → starts after `By`
- Conditions can be combined using **And** / **Or**.
- Additional keywords like **Distinct**, **First**, **Top**, **IgnoreCase**, and **OrderBy** provide extra control.

Example:

```
find   By   Lastname   And   Firstname   OrderBy   AgeDesc
|      |      |        |      |          |
Subject →   Predicate (conditions)  →  Sorting
```

Derived Query Method Examples

```
// Simple property match
List<Person> findByEmail(String email);

// Multiple conditions (AND)
List<Person> findByEmailAddressAndLastname(String email, String lastname);

// Remove duplicates
List<Person> findDistinctByLastnameOrFirstname(String lastname, String firstname);

// Case-insensitive comparison
List<Person> findByLastnameIgnoreCase(String lastname);

// Case-insensitive comparison across multiple fields
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Static sorting
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

Keyword	Meaning
Distinct	Removes duplicate results
IgnoreCase	Case-insensitive matching
OrderBy	Adds fixed sorting to the query
And / Or	Combine multiple conditions

You can find the **full query method list here**.

Equality Condition Keywords

Exact Match (=)

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

Not Equal (!=)

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

Null Checks

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

Boolean Conditions

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

Key Points

- The **predicate starts after** `By` in method names.
- `Is` / `Equals` → same meaning, improves readability.
- Null values automatically translate to **IS NULL** in SQL.
- `True` and `False` require **no method arguments**.

Similarity Condition Keywords

Predefined Pattern Expressions

Keyword	Description	Example	SQL Equivalent
StartingWith	Property starts with a value	`findByNameStartingWith("Jo")`	`WHERE name LIKE 'Jo%'`
EndingWith	Property ends with a value	`findByNameEndingWith("son")`	`WHERE name LIKE '%son'`
Containing	Property contains a value	`findByNameContaining("oh")`	`WHERE name LIKE '%oh%'`

Custom LIKE Patterns

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

- Allows complex patterns using `%` manually.
- Example: names starting with `a`, containing `b`, and ending with `c`.

Key Points

- Predefined pattern methods **do not require** `%` in arguments.
- `Like` keyword lets you **manually define patterns** for complex queries.
- Works with **String properties** for flexible searching.

Comparison Condition Keywords

Comparison Operators

Keyword	Description	Example
LessThan (<)	Property value is less than the given value	`findByAgeLessThan(30)`
LessThanEqual (<=)	Property value is less than or equal to the given value	`findByAgeLessThanEqual(30)`
GreaterThan (>)	Property value is greater than the given value	`findByAgeGreaterThan(18)`
GreaterThanEqual (>=)	Property value is greater than or equal to the given value	`findByAgeGreaterThanEqual(18)`

Range & Collections

Keyword	Description	Example
Between	Property value is between two values	`findByAgeBetween(18, 30)`
In	Property value is in a given collection	`findByAgeIn(Arrays.asList(18, 25, 30))`

Date Comparisons

Keyword	Description	Example
Before	Property value is before the given date	`findByBirthDateBefore(birthDate)`
After	Property value is after the given date	`findByBirthDateAfter(birthDate)`

Key Points

- Supports **numeric, date, and comparable types**.
- Combine keywords with **And / Or** for more complex conditions.
- Works seamlessly with **derived query method names**.

Combining Multiple Conditions

Combining Expressions with `And` / `Or`

```
// Simple OR condition
List<User> findByNameOrAge(String name, Integer age);

// OR + AND combination
List<User> findByNameOrAgeAndActive(String name, Integer age, Boolean active);
```

Key Points:

- The precedence order is `And` then *Or*, just like Java.
- You can combine **multiple conditions** in a single method.
- No strict limit, but **keep method names readable**.
- For **complex queries**, consider using `@Query` instead.

Best Practice:

- Avoid very long derived query names.
- Use `@Query` for clarity and maintainability when combining many conditions.

`@Query` Annotation

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.

JPQL

By default, the query definition uses JPQL (Java Persistence Query Language).

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")
List<User> findAllActiveUsers();
```

While JPQL offers portability across different database systems, its abstraction level sometimes requires developers to turn to native SQL queries to leverage database-specific features.

Understanding JPQL and Its Limitations

What is JPQL?

- Java Persistence Query Language (JPQL) is part of JPA.
- Queries are **written against entities**, not tables.
- **Database-agnostic**: works across different database systems.

Example:

```
String jpql = "SELECT u FROM User u";
```

Strengths

- **Database Independence** → portable queries.
- **Object-Oriented** → intuitive for entity-based development.
- **Simplification** → abstracts complex joins and SQL operations.

Limitations

- **No Database-Specific Features**
  - Cannot use functions like `STRING_AGG()` or window functions directly.
  - Example SQL not possible in JPQL:

```
SELECT STRING_AGG(u.name, '; ') FROM users u WHERE u.department_id = 1;
```

- **Performance Overhead**
  - JPQL queries are translated to SQL by the JPA provider.
  - Can generate inefficient SQL for complex queries.
- **Complex Queries Less Intuitive**
  - Constructing multi-entity or nested queries may be harder than SQL.
- **Limited Dynamic Query Capabilities**
  - Runtime query construction is less flexible than native SQL or Criteria API.

Native Query

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)
List<User> findAllActiveUsersNative();
```

Why Use Native Queries?

- **Direct access to the database** → bypass JPQL limitations.
- Can use **database-specific functions** not available in JPQL.
- Useful for **complex joins, aggregations, and performance optimizations**.

Example: Leveraging Database-Specific Functions

- PostgreSQL `STRING_AGG()` to concatenate values:

```
SELECT STRING_AGG(name, ', ')
FROM users
WHERE department_id = 1;
```

- JPQL cannot handle this directly; native SQL is required.

Joining Tables in a Query

JPQL

```
@Query("SELECT new ResultDTO(c.id, o.id, p.id, c.name, c.email, o.orderDate, p.productName, p.price) "
    + "FROM Customer c, CustomerOrder o, Product p "
    + "WHERE c.id = o.customer.id AND o.id = p.customerOrder.id AND c.id = ?1")
List<ResultDTO> findResultDTOByCustomer(Long id);
```

- Requires a **DTO** to hold selected fields.
- **Type-safe** and selects only needed columns.

Native

```
@Query(value = "SELECT c.id AS customerId, o.id AS orderId, p.id AS productId, " +
        "c.name AS customerName, c.email AS customerEmail, " +
        "o.order_date AS orderDate, p.product_name AS productName, p.price AS productPrice " +
        "FROM Customer c " +
        "JOIN CustomerOrder o ON c.id = o.customer_id " +
        "JOIN Product p ON o.id = p.customerOrder_id " +
        "WHERE c.id = ?1",
    nativeQuery = true)
List<CustomerOrderProductProjection> findByCustomer(Long id);
```

- Can return **interface projections** or `Map<String,Object>`, but **interface projections** are preferred.
- Supports **DB-specific features** and complex queries.

Key Points

- **JPQL:** Type-safe, DTO needed, selects specific fields.
- **Native SQL:** Flexible, DB-specific, projections preferred over Map.
- **Best Practice:** JPQL + DTO for clarity; native SQL when JPQL cannot handle the query.

Passing method parameters to query

- Queries often need **dynamic values** based on method input.
- Spring Data JPA supports **two main ways** to pass parameters:
  a. **Indexed Parameters (**`?1`**,** `?2`**, ...)**
  b. **Named Parameters (** `@Param("paramName")`**)**

Benefits

- Makes queries **flexible** and **reusable**.
- Keeps repository methods **type-safe**.
- Improves **readability and maintainability**.

Indexed Query Parameters

- Parameters are **assigned to the query based on their order** in the method.
- First method parameter → `?1`, second → `?2`, etc.

JPQL Example

```
@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);
```

Native Query Example

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

Named Query Parameters

- Parameters are **matched by name** rather than position.
- Use `:paramName` in the query and `@Param("paramName")` in the method.
- **Safer and more readable**, especially if you reorder or add parameters.

JPQL Example

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

Native Query Example

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

Collection Parameters in Queries

- Sometimes you want to check if a column matches **multiple values**.
- Use `IN (:param)` in JPQL or SQL.
- Pass a **Collection** (`List`, `Set`, etc.) as the method parameter.

Example (JPQL / Named Parameter)

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

How It Works

- `:names` is replaced by the collection elements.
- Returns all users whose `name` matches any value in the collection.

Notes

- Can be used with `IN` or `NOT IN`.
- Supports `List`, `Set`, or any `Collection` type.

N+1 Query Problem in JPA

What is N+1 Problem?

- Occurs when fetching **one entity** triggers **additional queries** for its related entities.
- Leads to **performance issues** due to excessive database queries.

Example

```
// Fetch all users
List<User> users = userRepository.findAll();

// For each user, lazy-loaded orders trigger a separate query
for (User u : users) {
    System.out.println(u.getOrders().size());
}
```

- `1 query` for users + `N queries` for orders → **N+1 problem**

Solutions for N+1 Query Problem

- **Use** `JOIN FETCH` **in JPQL**

```
@Query("SELECT u FROM User u JOIN FETCH u.orders")
List<User> findAllUsersWithOrders();
```

Fetches users **and their orders in a single query**.

- **Use** `@EntityGraph`

```
@EntityGraph(attributePaths = {"orders"})
List<User> findAll();
```

JPA automatically fetches **related entities eagerly**.

- **Change Fetch Type (if appropriate)**

```
@OneToMany(fetch = FetchType.EAGER)
private List<Order> orders;
```

Forces eager loading, but can **impact memory usage**.

Key Takeaways

- N+1 queries **hurt performance**.
- Prefer **JOIN FETCH** or **EntityGraph** to avoid extra queries.
- Be cautious with **global eager fetch** — only use where necessary.

Running Sample Project

- **Clone the Repository**

If you haven't cloned the repository yet, run the following command (ensure `git` is installed):

```
git clone https://github.com/PialKanti/Ostad-SpringBoot-Course.git
```

Then switch to the correct branch for today's class (replace with the actual branch name, e.g., `class-24-jpa-relationship`):

```
git fetch
git switch class-25-query
```

Or,

If You Already Have the Repository Cloned, simply open your existing project folder and switch (or checkout) to the appropriate branch:

```
git fetch
git switch class-25-query
```

- **Set Up and Run PostgreSQL Database**

You can run PostgreSQL **either via Docker** or a **desktop installation**.

Option 1: Run via Docker

A `compose.yml` file is available in the root of the repository.
Run the following command from the project root:

```
docker compose up -d
```

This will start a PostgreSQL container automatically.

Or,

Option 2: Run via PostgreSQL Desktop (Manual Setup)

If you already have PostgreSQL installed locally:

1. Start your PostgreSQL server.
2. Create a new database named `crud_db` if not exists.

- **Open the Project in IntelliJ IDEA**

1. Open IntelliJ IDEA.
2. Click **File → Open** and select the `crud-sample` folder inside the repository.
3. Let IntelliJ import Maven/Gradle dependencies automatically.