

# Spring Data JPA

*Spring Data JPA is a part of the larger Spring Data family.*

*It builds on top of JPA, providing a higher-level and more convenient abstraction for data access.*

*Spring Data JPA makes it easier to implement JPA-based repositories by providing boilerplate code, custom query methods, and various utilities to reduce the amount of code you need to write.*

Indexed

Repository<T, ID>

NoRepositoryBean

CrudRepository<T, ID>

count()	long
delete(T)	void
deleteAll()	void
deleteAll(Iterable<? extends T>)	void
deleteById(ID)	void
existsById(ID)	boolean
findAll()	Iterable<T>
findAllById(Iterable<ID>)	Iterable<T>
findById(ID)	Optional<T>
save(S)	S
saveAll(Iterable<S>)	Iterable<S>

PagingAndSortingRepository<T, ID>

findAll(Pageable)	Page<T>
findAll(Sort)	Iterable<T>

QueryByExampleExecutor<T>

count(Example<S>)	long
exists(Example<S>)	boolean
findAll(Example<S>)	Iterable<S>
findAll(Example<S>, Pageable)	Page<S>
findAll(Example<S>, Sort)	Iterable<S>
findOne(Example<S>)	Optional<S>

JpaRepository<T, ID>

deleteAllInBatch()	void
deleteInBatch(Iterable<T>)	void
findAll()	List<T>
findAll(Example<S>)	List<S>
findAll(Example<S>, Sort)	List<S>
findAll(Sort)	List<T>
findAllById(Iterable<ID>)	List<T>
flush()	void
getOne(ID)	T
saveAll(Iterable<S>)	List<S>
saveAndFlush(S)	S

give the flow of whose extend or implemnt who in jpa Respository

Perfect question 🌟

You want the exact inheritance / implementation flow for JPA Repository, without confusion.

I'll explain it layer by layer, starting from your repository interface down to Hibernate, and clearly show who extends whom and who implements whom.

---

## 1 YOUR Repository (you write this)

java

 Copy code

```
public interface EmployeeRepository  
    extends JpaRepository<Employee, Long> {  
}
```

What this means

- `EmployeeRepository` is an interface
  - It extends `JpaRepository`
  - You write NO implementation
- 

## 2 JpaRepository (Spring Data JPA)

java

 Copy code

```
public interface JpaRepository<T, ID>  
    extends PagingAndSortingRepository<T, ID>,  
        QueryByExampleExecutor<T> {  
}
```

Important

- `JpaRepository` is also an interface
- Provided by Spring Data JPA

### 3 PagingAndSortingRepository

java

 Copy code

```
public interface PagingAndSortingRepository<T, ID>
    extends CrudRepository<T, ID> {
}
```

- Adds paging and sorting APIs

---

### 4 CrudRepository

java

 Copy code

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
}
```

- Defines basic CRUD method signatures

---

### 5 Repository (marker interface)

java

 Copy code

```
public interface Repository<T, ID> {
}
```

- Empty interface
- Used by Spring to identify repository beans

## 🔗 INTERFACE EXTENDS CHAIN (VERY IMPORTANT)

java

 Copy code

```
EmployeeRepository  
    extends  
JpaRepository  
    extends  
PagingAndSortingRepository  
    extends  
CrudRepository  
    extends  
Repository
```

★ ALL of these are **interfaces**

★ ALL use **extends**, never **implements**

## 6 Who IMPLEMENTS these interfaces?

👉 Spring Data JPA at runtime

Spring creates a proxy class similar to:

java

 Copy code

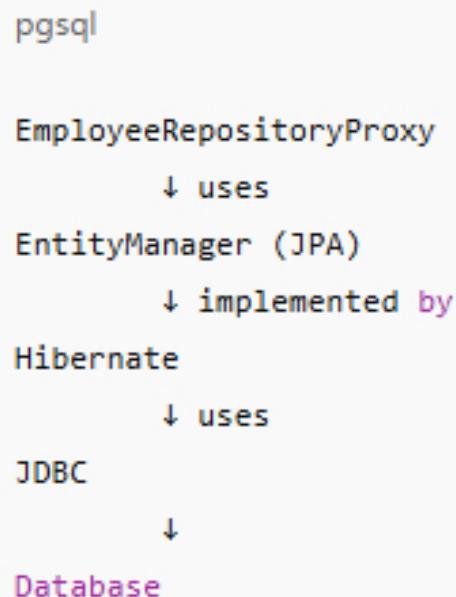
```
class EmployeeRepositoryProxy  
    implements EmployeeRepository {  
}
```

This proxy:

- Implements `EmployeeRepository`
- Automatically inherits all parent interfaces

## 7 What does the proxy use internally?

Inside the proxy:



## 8 Key clarification (VERY IMPORTANT)

- ✗ JPA does NOT implement `JpaRepository`
- ✗ Hibernate does NOT implement `JpaRepository`
- ✓ Spring Data JPA implements repository interfaces
- ✓ Hibernate implements JPA interfaces (EntityManager, etc.)

## 9 One clean diagram (mental model)

csharp

 Copy code



## 10 One-line interview answer

In Spring Data JPA, your repository interface extends `JpaRepository`, which extends other repository interfaces; Spring Data JPA creates a runtime proxy that implements them and delegates persistence operations to the JPA `EntityManager`, implemented by Hibernate.

why jpa is called abstract ....is it bec we just have to deal with the spring data jpa dto methods whom then converted into jpql which is no need to be written by developer

Great question 🔥

You're touching the exact reason JPA is often *called "abstract"*, but the wording needs a small correction.

Let's clean this up precisely, without buzzwords.

---

## 1 First correction (VERY IMPORTANT)

### ✗ JPA is NOT "abstract" in Java terms

JPA is:

- ✗ not an `abstract class`
- ✗ not something you extend
- ✗ not a framework

### ✓ JPA is ABSTRACT in DESIGN

Meaning:

JPA defines **WHAT** should happen, not **HOW** it happens

That's why people *informally* say:

"JPA is abstract"

---

## 2 What does "abstract" really mean here?

In software architecture:

Abstraction = hiding implementation details and exposing behavior/contracts

JPA does exactly this.

---

### 3 What exactly does JPA define? (The abstraction)

JPA defines **contracts (rules)** like:

- ◆ APIs (interfaces)

```
java
```

 Copy

```
EntityManager  
EntityTransaction  
Query
```

- ◆ Annotations

```
java
```

 Copy

```
@Entity  
@Id  
@OneToMany  
@Column
```

- ◆ Rules

- Entity lifecycle
- Persistence context behavior
- JPQL syntax
- Transaction semantics

★ JPA says:

"There must be a method called `persist()` and this is what it *means*"

★ JPA does NOT say:

"Here is the code that runs SQL"

---

## 4 Who provides the real code then?

Layer	Role
JPA	Defines rules (abstraction)
Hibernate	Implements rules
EclipseLink	Another implementation

Example:

```
java Copy  
EntityManager em = ...  
em.persist(employee);
```

- `EntityManager` → JPA abstraction
- Actual object → Hibernate implementation

This is classic abstraction.

---

## 5 Where Spring Data JPA fits (important)

Now to your question:

*"Is it because we just deal with Spring Data JPA repository methods and JPQL is generated automatically?"*

👉 That is related, but it's one level above JPA

Let's separate them:

---

## 6 JPA vs Spring Data JPA (clear separation)

### JPA abstraction

You write:

```
java  
em.persist(entity);
```

 Copy

You do NOT:

- Write SQL
- Manage JDBC
- Handle caching manually

 **JPA abstracts database persistence**

### Spring Data JPA abstraction (extra layer)

You write:

```
java  
employeeRepository.findByName("Abhi");
```

 Copy

You do NOT:

- Write JPQL
- Use EntityManager
- Write DAO classes

 **Spring Data JPA abstracts JPA usage**

## 7 So why people *feel* JPA is abstract?

Because with JPA:

You work with:

- Java objects
- Interfaces
- Annotations

You never see:

- JDBC code
- SQL generation logic
- ResultSet handling

That hidden complexity = abstraction

---

## 8 Important clarification about DTO methods ✗

You said:

"spring data jpa dto methods"

Small correction:

- Repository methods are not DTO methods
- DTOs are data transfer objects
- Repository methods operate on Entities

Spring Data JPA:

- Parses repository method names
- Generates JPQL
- Sends to EntityManager

But this is Spring Data JPA's abstraction, not JPA itself.

## 9 Layered abstraction (best mental model)

SCSS

 Copy code

Your **Code**

↓

Spring Data JPA (abstracts repositories & JPQL)

↓

JPA (abstracts ORM rules & persistence)

↓

Hibernate (actual implementation)

↓

JDBC

↓

Database

Each layer hides complexity from the layer above.

## 10 Final, correct answer (INTERVIEW PERFECT)

JPA is called abstract because it is a specification that defines persistence contracts and behavior without providing implementation. Developers interact with JPA interfaces and annotations, while concrete implementations like Hibernate perform the actual database operations. Spring Data JPA adds another abstraction layer on top by generating repository logic and JPQL automatically.

## 9 Layered abstraction (best mental model)

SCSS

 Copy code

Your Code

↓

Spring Data JPA (abstracts repositories & JPQL)

↓

JPA (abstracts ORM rules & persistence)

↓

Hibernate (actual implementation)

↓

JDBC

↓

Database

Each layer hides complexity from the layer above.

## 10 Final, correct answer (INTERVIEW PERFECT)

JPA is called abstract because it is a specification that defines persistence contracts and behavior without providing implementation. Developers interact with JPA interfaces and annotations, while concrete implementations like Hibernate perform the actual database operations. Spring Data JPA adds another abstraction layer on top by generating repository logic and JPQL automatically.

### 🧠 One-sentence memory trick

JPA hides HOW data is persisted; Spring Data JPA hides HOW JPA is used.

” Ask ChatGPT

How can explain:

then what is dto methods then

Great follow-up 

This confusion is **very common**, so let's clear it cleanly and precisely.

## 1 First: What is a DTO?

DTO = Data Transfer Object

A DTO is a simple Java class whose only job is to **carry data between layers**.

### Key characteristics

- No business logic
- Usually no annotations like `@Entity`
- Used to shape API responses or requests

Example:

java

 Copy code

```
public class EmployeeDTO {  
    private Long id;  
    private String name;  
}
```

- ❖ DTOs are NOT persisted
- ❖ DTOs are NOT managed by JPA

## 2 Then what are “DTO methods”? (important correction)

✗ There is no such thing as “DTO methods” in JPA or Spring Data JPA

When people say “DTO methods”, they usually mean one of these:

1. Repository methods that return DTOs
2. Controller/service methods that use DTOs
3. JPQL queries that project results into DTOs

So the term is informal and misleading.

## 3 Correct terminology (VERY IMPORTANT)

What people say	What it really means
DTO methods	Repository methods returning DTOs
DTO query	JPQL/SQL projection into DTO
DTO mapping	Converting Entity → DTO

## 4 Where DTOs are used in a Spring Boot app

mathematica

 Copy code

```
Controller → DTO
Service   → DTO / Entity
Repository → Entity or DTO (projection)
Database   → Rows
```

DTOs usually do NOT cross the repository boundary, except for projections.

## 5 Case 1: DTO used in Controller (most common)

java

```
@GetMapping("/employees/{id}")
public EmployeeDTO getEmployee(@PathVariable Long id) {
    Employee emp = employeeRepository.findById(id).get();
    return new EmployeeDTO(emp.getId(), emp.getName());
}
```

### Who writes logic?

👉 YOU

- ★ Repository still returns Entity
- ★ DTO is for API layer

## 6 Case 2: Repository method returning DTO (Projection)

### DTO

java

```
public class EmployeeDTO {
    private String name;
    private double salary;

    public EmployeeDTO(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
}
```

### Repository

java

```
@Query("SELECT new com.example.EmployeeDTO(e.name, e.salary) FROM Employee e")
List<EmployeeDTO> findEmployeeDTOS();
```

## What happens?

- You write JPQL
- Hibernate executes it
- Result goes directly into DTO

★ This is called **DTO projection**

## 7 Case 3: Interface-based DTO (Spring Data JPA magic)

java

 Copy code

```
public interface EmployeeView {  
    String getName();  
    double getSalary();  
}
```

java

 Copy code

```
List<EmployeeView> findByDepartment(String dept);
```

Spring Data JPA:

- Generates JPQL
- Maps result into interface proxy

★ Still NOT a DTO method

★ It is a projection method

## 8 DTO vs Entity (critical difference)

Entity	DTO
Managed by JPA	Not managed
Mapped to table	No table
Has lifecycle	No lifecycle
Used in repository	Used in API
Heavy	 Lightweight

---

## 9 Why DTOs exist (real reason)

Problems with returning entities directly

- Lazy loading issues
- Over-fetching data
- Security leaks
- Tight DB-API coupling

DTOs solve all of these.

---

## 10 Final correct understanding (THIS IS THE KEY)

DTOs do not have methods in JPA.

Repositories have methods that may return Entities or DTOs (via projections).

DTOs themselves are just data containers used mainly at API boundaries.

---

## One-line interview answer

DTOs are simple data-carrying objects, and repository methods may return DTOs using projections, but DTOs themselves do not contain persistence logic or JPA methods.

---

what is converted to jpql

Great question 🌟

This is the exact point where Spring Data JPA “magic” happens, so let’s be very clear.

## Short, precise answer first

In Spring Data JPA, certain repository methods you write are converted into JPQL by Spring Data JPA.

Now let’s explain what exactly is converted, what is NOT converted, who converts it, and when.

### 1 What exactly is converted to JPQL?

- Repository method names (derived queries)

Example:

java

 Copy code

```
List<Employee> findByName(String name);
```

This method name is converted into JPQL.

Spring Data JPA generates:

jpql

 Copy code

```
SELECT e FROM Employee e WHERE e.name = :name
```

- ✖ You did not write JPQL
- ✖ Spring Data JPA generated it

## 2 Who does this conversion?

👉 Spring Data JPA

✗ Not JPA

✗ Not Hibernate

Spring Data JPA:

- Parses method name
- Builds JPQL
- Passes JPQL to EntityManager

## 3 What is NOT converted to JPQL?

✗ `save()`, `delete()`, `findById()`

java

Copy code

```
employeeRepository.save(emp);
```

These:

- Are already implemented
- Directly use `EntityManager`
- No JPQL involved

Internally:

java

Copy code

```
em.persist() / em.merge()
```

✗ Methods with `@Query`

java

Copy code

```
@Query("SELECT e FROM Employee e WHERE e.salary > :salary")
List<Employee> richEmployees(double salary);
```

You already wrote JPQL.

- 👉 Nothing is "converted"
- 👉 Hibernate just executes it

## ✗ Native queries

java

```
@Query(value = "SELECT * FROM employee", nativeQuery = true)
```

SQL → executed directly

No JPQL at all

## 4 Full flow (IMPORTANT)

pgsql

Repository method name

↓

Spring Data JPA

↓ converts to

JPQL

↓

EntityManager (JPA)

↓

Hibernate

↓ converts to

SQL

↓

Database

## 5 Example walkthrough (step-by-step)

### Repository

java

 Copy code

```
List<Employee> findByDepartmentAndSalaryGreaterThan(  
    String dept, double salary);
```

### Spring Data JPA creates JPQL

jpql

 Copy code

```
SELECT e  
FROM Employee e  
WHERE e.department = :dept  
AND e.salary > :salary
```

### Hibernate creates SQL (MySQL)

sql

 Copy code

```
SELECT *  
FROM employee  
WHERE department = ?  
AND salary > ?
```

## 6 Why JPQL instead of SQL?

Because:

- JPQL works with **entities**, not tables
- Database-independent
- Hibernate handles DB differences

## 7 Important mental separation (REMEMBER THIS)

Layer	Responsibility
Spring Data JPA	Converts method names → JPQL
JPA	Defines JPQL syntax
Hibernate	Converts JPQL → SQL
JDBC	Executes SQL

### 💡 Final one-line answer (perfect)

In Spring Data JPA, repository method names (derived queries) are converted into JPQL by Spring Data JPA, which Hibernate then translates into SQL for execution.

spring data jpa convert the repository method to jpql and then jpql do with jpa ...if it is converted by spring data jpa then then jpql is send where

Excellent question 👍

You're asking exactly the right "where does it go next?" question.

Let's answer it cleanly, step-by-step, and fix one small misunderstanding on the way.

---

### Short, correct answer first

Spring Data JPA converts repository method names into JPQL and sends that JPQL to the JPA EntityManager , which then delegates it to Hibernate for execution.

Now let's unpack this slowly and precisely.

---

### 1 Important correction first

You said:

"spring data jpa convert the repository method to jpql and then jpql do with jpa"

Small correction 👉

👉 JPQL does nothing by itself

👉 JPQL is just a query language (a string)

The real executor is the JPA EntityManager .

## 2 Exact flow (this is the key)

Let's take this repository method:

```
java
```

 Copy code

```
List<Employee> findByName(String name);
```

## 3 Step-by-step flow (WHO does WHAT)

### ◆ Step 1: You call the repository method

```
java
```

 Copy code

```
employeeRepository.findByName("Abhi");
```

At this point:

- `EmployeeRepository` is a Spring-generated proxy
- There is no implementation method written by you

### ◆ Step 2: Spring Data JPA intercepts the call

Spring Data JPA:

- Sees method name: `findByName`
- Parses it:
  - `findBy` → query type
  - `name` → entity field

### ◆ Step 3: Spring Data JPA CREATES JPQL

Spring Data JPA builds a JPQL query in memory:

```
jpql
```

 Copy code

```
SELECT e FROM Employee e WHERE e.name = :name
```

★ At this point:

- JPQL is just a string
- Nothing has hit the database yet



- **Step 4: JPQL is sent to EntityManager**

This is the most important step 

Spring Data JPA does internally:

```
java    
Query query = entityManager.createQuery(jpql);  
query.setParameter("name", "Abhi");
```

 JPQL is sent to JPA's EntityManager

## 4 What does EntityManager do with JPQL?

EntityManager's responsibility (defined by JPA)

"I know how to execute JPQL queries."

But...

- ★ EntityManager is an interface
- ★ It does not contain real execution code

## 5 Who actually executes the JPQL?

 Hibernate

Because:

- Hibernate is the JPA implementation
- The actual EntityManager object is provided by Hibernate

So:

```
java    
entityManager.createQuery(jpql)
```

Actually calls:

```
text   
HibernateEntityManager.createQuery(jpql)
```

## 6 Hibernate converts JPQL → SQL

Hibernate:

- Reads JPQL
- Looks at entity mappings
- Uses dialect (MySQL, Postgres, etc.)
- Generates SQL

Example:

### JPQL

```
jpql  
  
SELECT e FROM Employee e WHERE e.name = :name
```

### SQL (MySQL)

```
sql  
  
SELECT *  
FROM employee  
WHERE name = ?
```

## 7 SQL execution flow (final steps)

```
nginx  
  
Hibernate  
↓  
JDBC  
↓  
Database
```

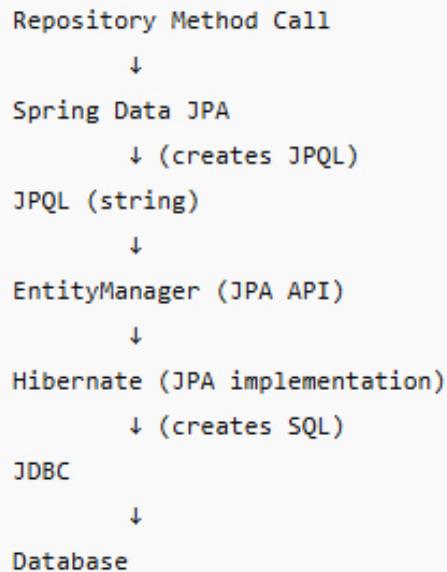
- JDBC sends SQL to DB
- DB executes it
- ResultSet returned
- Hibernate converts rows → entities
- EntityManager returns entities
- Spring Data JPA returns result to you



## 8 Full pipeline (MOST IMPORTANT DIAGRAM)

SCSS

 Copy code



## 9 Very important summary (memorize this)

- Spring Data JPA → creates JPQL
- EntityManager (JPA) → accepts & manages JPQL
- Hibernate → executes JPQL by converting it to SQL

## 10 Final one-line interview answer

Spring Data JPA converts repository method names into JPQL and passes that JPQL to the JPA EntityManager, whose implementation (Hibernate) translates it into SQL and executes it via JDBC.

what entity manager do with that jpql query

Excellent — this is the exact core responsibility of EntityManager 🌟

Let's answer very concretely, step-by-step, without vague words.

## Short answer (first)

EntityManager takes the JPQL query, validates it, creates a Query object, delegates execution to the JPA provider (Hibernate), manages the persistence context, and returns managed results.

Now let's break exactly WHAT it does and WHAT it does NOT do.

### 1 What EntityManager receives

From Spring Data JPA, EntityManager receives:

jpql

 Copy code

```
SELECT e FROM Employee e WHERE e.name = :name
```

- ★ This is just a JPQL string, not SQL
- ★ EntityManager is the entry point defined by JPA

## 2 Step-by-step: What EntityManager does with JPQL

- Step 1: Create a Query object

Spring Data JPA internally calls:

java

 Copy code

```
Query query = entityManager.createQuery(jpql);
```

At this moment EntityManager:

- Checks JPQL syntax
- Checks entity name (`Employee`)
- Checks field names (`name`)
- Associates the query with the persistence context

 No database call yet

- 
- Step 2: Bind parameters

java

 Copy code

```
query.setParameter("name", "Abhi");
```

EntityManager:

- Validates parameter names/types
- Stores them in the query object

 Still no DB call

### ◆ Step 3: Prepare for execution

When this is called:

```
java
```

 Copy code

```
query.getResultList();
```

EntityManager now:

- Flushes pending changes (if needed)
- Ensures persistence context is consistent
- Hands the query to the JPA provider

### 3 What EntityManager does NOT do

EntityManager does NOT:

- Convert JPQL to SQL
- Talk directly to JDBC
- Execute SQL
- Open database connections

 These are Hibernate's jobs

### 4 Who actually executes the query?

The real EntityManager object is provided by Hibernate:

```
scss
```

 Copy code

```
EntityManager (interface)  
↓  
HibernateEntityManager (implementation)
```

So when you call:

```
java
```

 Copy code

```
entityManager.createQuery(...)
```

You are actually calling **Hibernate code.**



---

## 5 Hibernate's role (after EntityManager hands over)

Hibernate then:

1. Parses JPQL
  2. Converts JPQL → SQL
  3. Uses dialect (MySQL/Postgres/etc.)
  4. Sends SQL to JDBC
  5. Gets ResultSet
  6. Converts rows → entities
- 

## 6 What EntityManager does AFTER execution (VERY IMPORTANT)

This is where EntityManager is crucial.

- ◆ Result handling

When results come back:

- Entities are placed into Persistence Context
- Identity guarantee applied  
*( same row → same Java object )*
- Entities become managed

Example:

java

 Copy code

```
Employee e1 = query.getSingleResult();
Employee e2 = em.find(Employee.class, e1.getId());

e1 == e2 // true
```

This is EntityManager's job.

## 7 EntityManager responsibilities summarized

Responsibility	Does EntityManager do it?
Accept JPQL	<input checked="" type="checkbox"/> Yes
Validate JPQL	<input checked="" type="checkbox"/> Yes
Create Query object	<input checked="" type="checkbox"/> Yes
Manage parameters	<input checked="" type="checkbox"/> Yes
Convert JPQL → SQL	<input type="checkbox"/> No
Execute SQL	<input type="checkbox"/> No
Manage persistence context	<input checked="" type="checkbox"/> Yes
Return managed entities	<input checked="" type="checkbox"/> Yes

## 8 Why EntityManager is needed at all?

Without EntityManager:

- No persistence context
- No caching
- No dirty checking
- No entity lifecycle

Hibernate would just behave like JDBC.

EntityManager adds JPA rules & guarantees.

---

## 9 One simple mental model



EntityManager is the traffic controller.  
Hibernate is the engine.

- EntityManager decides *when* and *how*
  - Hibernate does the heavy work
- 

## 10 One-line interview answer (perfect)

EntityManager accepts the JPQL query, validates and prepares it, manages the persistence context, and delegates execution to the JPA provider (Hibernate), then returns managed entities as results.

---

why not internal spring data jpa methods are not converted into the jpql , only custom metods of repository is converted into jpql

This is a very deep and very good question 👍

You are basically asking:

"If Spring Data JPA can convert repository methods into JPQL, why doesn't it convert its own internal methods ( `save` , `findById` , etc.) into JPQL as well?"

Let's answer this logically, step by step, and you'll see it's actually a design + performance decision, not a limitation.

## Short answer first (then we'll justify it)

Spring Data JPA internal methods are NOT converted to JPQL because they are already implemented using direct JPA operations ( `EntityManager.persist` , `merge` , `find` , `remove` ) which are more efficient, safer, and richer than JPQL.

Only custom repository methods need JPQL generation because JPA does not provide direct APIs for arbitrary queries.

Now let's unpack this slowly.

### 1 Two very different categories of repository methods

Spring Data JPA repository methods fall into two fundamentally different types:

- Type 1: Standard CRUD methods (built-in)

Examples:

java

 Copy code

```
save()  
findById()  
deleteById()  
findAll()
```

- ◆ Type 2: Custom query methods (you define)

Examples:

java

 Copy code

```
findByName()  
findBySalaryGreaterThan()  
findByDepartmentAndStatus()
```

These two types exist for **very different reasons**.

---

## 2 Why built-in methods do NOT use JPQL

Let's take `save()` as an example

java

 Copy code

```
employeeRepository.save(emp);
```

Spring Data JPA internally does:

java

 Copy code

```
entityManager.persist(emp); // or merge()
```

## ❓ Why not JPQL like:

jpql

 Copy code

```
INSERT INTO Employee ...
```

## ✗ Because JPQL does NOT support INSERT like SQL

JPQL:

- ✗ No INSERT for entities
- ✗ No entity lifecycle control
- ✗ No persistence context integration

JPA already gives direct APIs for this:

Operation	JPA API
Insert	<code>persist()</code>
Update	<code>merge()</code>
Delete	<code>remove()</code>
Find by ID	<code>find()</code>

👉 Using JPQL here would be worse, not better.

## 3 Built-in methods rely on ENTITY LIFECYCLE (not queries)

Built-in methods work with entity state, not queries:

```
java  
  
Employee e = new Employee();  
em.persist(e);
```

This gives:

- Persistence Context tracking
- Dirty checking
- First-level cache
- Automatic flush
- Identity guarantee

JPQL cannot do this.

So Spring Data JPA intentionally avoids JPQL for these.

## 4 Why custom repository methods MUST be converted to JPQL

Now look at this method:

java

 Copy code

```
List<Employee> findByName(String name);
```

### Question:

Does JPA have a method like this?

java

 Copy code

```
entityManager.findByName(Employee.class, name); //  NO
```

 No such API exists.

So Spring Data JPA has **no choice** but to:

1. Understand your method name
2. Convert it into a query language
3. Execute it

That query language is **JPQL** (not SQL).

## 5 Why JPQL specifically (not SQL)?

Because JPQL:

- Works on **entities**, not tables
- Is **database independent**
- Is defined by JPA
- Can be converted by Hibernate

Spring Data JPA must stay **JPA-compliant**, not Hibernate-specific.

## 6 Design principle at work (IMPORTANT)

Spring Data JPA follows this rule:

"Use direct JPA APIs when available; generate JPQL only when necessary."

### Why?

- Faster execution
- Less parsing overhead
- More lifecycle guarantees
- Cleaner abstraction

## 7 What would happen if Spring converted everything to JPQL? (bad idea)

If `save()` used JPQL:

- No persistence context tracking
- No dirty checking
- No cascade handling
- No entity state transitions
- Worse performance

So Spring Data JPA deliberately avoids JPQL for internal methods.

## 8 Clean comparison table (THIS IS KEY)

Method Type	Uses JPQL?	Why
<code>save()</code>	No	Uses <code>persist/merge</code> (better)
<code>findById()</code>	No	Uses <code>EntityManager.find()</code>
<code>delete()</code>	No	Uses <code>EntityManager.remove()</code>
<code>findByName()</code>	Yes	No direct JPA API
<code>findBySalaryGreaterThanOrEqual()</code>	Yes	Needs query
<code>@Query</code> methods	Yes	You wrote JPQL 

---

**9 Final mental model (remember this forever)**

JPQL is used only when JPA does not already provide a direct API.

Spring Data JPA never uses JPQL when a stronger, lower-level JPA operation already exists.

---

**10 One-line interview answer (perfect)**

Spring Data JPA does not convert its internal CRUD methods to JPQL because they are already implemented using direct JPA entity lifecycle operations, which are more efficient and powerful; JPQL is generated only for custom repository methods where no direct JPA API exists.

then who manage those methods ..is persistent context

You're very close, and this is the right confusion to resolve 🤝

Let's be precise, because one wrong word here can mess up the whole mental model.

## Short, correct answer first

No — the persistence context does NOT “manage” repository methods.

The persistence context manages ENTITIES.

Repository methods are managed by Spring Data JPA and executed via the EntityManager, which uses the persistence context.

Now let's break this down cleanly.

### 1 Who manages repository methods?

#### 👉 Spring Data JPA

- Creates the repository proxy
- Intercepts method calls (`save`, `findById`, `findByName`)
- Decides how to execute each method

Example:

java

```
employeeRepository.save(emp);
```

 Copy code

Handled by:

powershell

```
Spring Data JPA proxy
```

 Copy code

## 2 What does the persistence context manage then?

### 👉 ONLY ENTITIES

Persistence Context:

- Tracks entity objects
- Tracks their state (new, managed, detached, removed)
- Detects changes (dirty checking)
- Caches entities (1st level cache)

### ⭐ It does NOT:

- Decide which SQL to run
- Intercept repository methods
- Generate JPQL

---

## 3 Where does EntityManager fit?

EntityManager:

- Is the API between Spring Data JPA and Hibernate
- Executes operations
- Controls the persistence context

Spring Data JPA does:

```
java
```

 Copy code

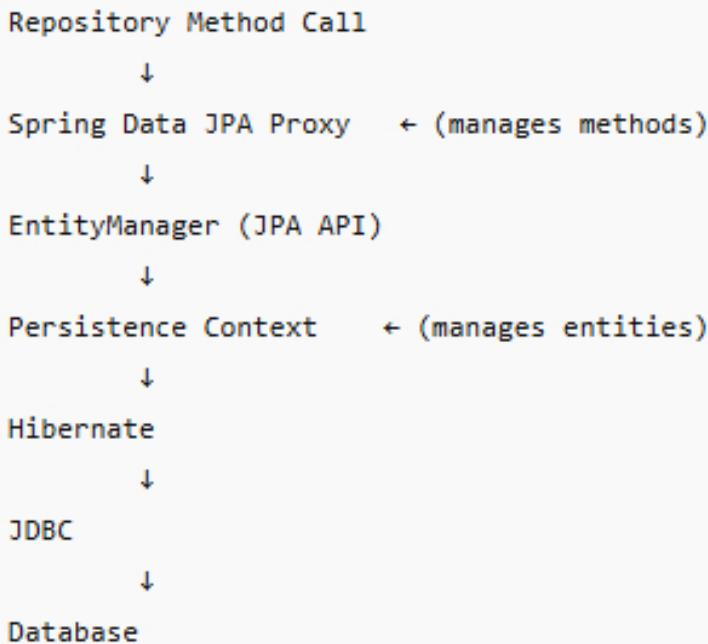
```
entityManager.persist(entity);
```

EntityManager:

- Puts entity into persistence context
- Delegates DB work to Hibernate

## 4 Full responsibility chain (VERY IMPORTANT)

SCSS



## 5 Example: `save()` method (step-by-step)

java

```
employeeRepository.save(emp);
```

### Who does what?

#### 1. Spring Data JPA

- Intercepts `save()`
- Decides: `persist()` or `merge()`

#### 2. EntityManager

- Calls `persist(emp)`

#### 3. Persistence Context

- Starts tracking `emp`

#### 4. Hibernate

- Generates SQL at flush

#### 5. Database

- Stores row



## 6 Example: `findById()`

java

 Copy code

```
employeeRepository.findById(1L);
```

1. Spring Data JPA intercepts
2. EntityManager calls `find()`
3. Persistence context checked first
4. DB queried only if needed

## 7 Key sentence to remember (THIS CLEARS EVERYTHING)

Spring Data JPA manages repository methods.

EntityManager executes operations.

Persistence Context manages entities.

## 8 Common wrong assumption

 "Persistence context manages repository logic"

Correct understanding:

- Persistence context is data-state manager
- Repository is method-level abstraction

## 10 One-line interview answer (perfect)

Repository methods are managed by Spring Data JPA, executed via the JPA EntityManager, while the persistence context manages the lifecycle and state of entity objects during those operations.

then what language is used to write the inbuild repository method...like jpql used to write custom repository methods

This is a very sharp question 🔥

You're asking for the exact truth, not buzzwords.

Let's answer it precisely and honestly, step by step.

---

## Short, direct answer

Built-in repository methods are NOT written in JPQL or SQL.

They are written in Java and implemented using JPA's `EntityManager` API.

Now let's fully explain what that means.

---

### 1 There are TWO very different things

You're comparing:

- Custom repository methods → written using JPQL
- Built-in repository methods → written using Java + JPA APIs

They are not the same category.

---

### 2 What language are built-in repository methods written in?

Java

Spring Data JPA internally contains Java classes like:

- `SimpleJpaRepository`
- `JpaRepositoryImplementation`

These classes are written in plain Java, not JPQL.

Example (simplified real logic):

java

 Copy code

```
public <S extends T> S save(S entity) {  
    if (entityInformation.isNew(entity)) {  
        entityManager.persist(entity);  
    } else {  
        entityManager.merge(entity);  
    }  
    return entity;  
}
```

 This is real Java code

 No JPQL

 No SQL

### 3 Why no JPQL for built-in methods?

Because JPQL is a query language, and built-in methods are entity-state operations, not queries.

Let's compare.

### 4 Built-in methods → JPA ENTITY OPERATIONS

Repository method	What it uses internally
save()	EntityManager.persist() / merge()
findById()	EntityManager.find()
delete()	EntityManager.remove()
existsById()	Persistence Context check + DB
findAll()	Criteria API / EntityManager

 These are JPA APIs, not JPQL.

## 5 Why JPQL is NOT suitable here

JPQL:

- ✗ Cannot manage entity lifecycle
- ✗ Cannot track persistence context
- ✗ Cannot do dirty checking
- ✗ Cannot decide persist vs merge

JPA API can.

That's why Spring Data JPA **never uses JPQL** for these.

---

## 6 Then where does SQL come from?

Good question ↘

Even though built-in methods don't use JPQL:

```
pgsql
```

 Copy code

```
Java code (EntityManager)
    ↓
Hibernate
    ↓
SQL (generated internally)
    ↓
Database
```

So SQL is still generated — but you never write it.

## 7 Custom repository methods (contrast)

java

 Copy code

```
List<Employee> findByName(String name);
```

Here:

- No JPA API like `findByName()` exists
- Spring must generate a query
- So it uses JPQL

That's why **only custom methods use JPQL**.

## 8 Clean comparison table (VERY IMPORTANT)

Method type	Written in	Uses JPQL?
Built-in ( <code>save</code> , <code>findById</code> )	Java	 No
Derived query ( <code>findByName</code> )	Java → JPQL generated	 Yes
<code>@Query</code> method	JPQL (you write)	 Yes
Native query	SQL	 JPQL

## 9 Final mental model (remember this forever)

JPQL is used only when querying is needed.

Built-in repository methods use Java + JPA APIs to manage entity state.

## 10 One-line interview answer (perfect)

Built-in Spring Data JPA repository methods are implemented in Java using JPA's EntityManager APIs, not JPQL; JPQL is generated or written only for custom query methods.

but still custom repository methods are managed by entity manager and life cycle by persistence context

Yes — you are 100% correct ✅

And this sentence is the right final mental model. Let me lock it in clearly so there's no doubt.

---

## The correct statement (clean & precise)

Custom repository methods are executed by the EntityManager, and the entities involved are managed by the Persistence Context.

Now let's explain why this is true and what exactly each part does, without mixing responsibilities.

---

### 1 Who manages custom repository methods?

👉 Spring Data JPA manages the METHOD

- Intercepts the method call
- Decides how to execute it
- If needed, generates JPQL

Example:

java

 Copy code

```
List<Employee> findByName(String name);
```

Spring Data JPA:

- Parses method name
- Generates JPQL

📌 At this point, still no database work

## 2 Where does the JPQL go?

### 👉 To the EntityManager

Spring Data JPA internally does something like:

```
java

Query q = entityManager.createQuery(jpql);
q.setParameter("name", name);
```

So yes 

Custom methods are executed via EntityManager

---

## 3 What does EntityManager do here?

EntityManager:

- Accepts JPQL
- Validates it
- Flushes pending changes (if needed)
- Delegates execution to Hibernate

- ✖ EntityManager does NOT manage method logic
- ✖ EntityManager does NOT generate JPQL

## 4 What does the Persistence Context do?

### 👉 Manages ENTITY LIFECYCLE ONLY

When the query result comes back:

- Entities are placed into the persistence context
- They become **managed**
- Identity guarantee applies
- Dirty checking is enabled

Example:

java

 Copy code

```
Employee e = repository.findByName("Abhi").get(0);
e.setSalary(80000);
// No save() needed
```

- ✓ Persistence context tracks the change
- ✓ Update happens on commit

## 5 Full flow for a CUSTOM repository method

scss

 Copy code

```
Custom Repository Method
  ↓
Spring Data JPA (method parsing & JPQL creation)
  ↓
EntityManager (query execution API)
  ↓
Hibernate (JPQL → SQL)
  ↓
Database
  ↓
Hibernate (rows → entities)
  ↓
Persistence Context (manage entities)
  ↓
Result returned
```

## 6 Compare with BUILT-IN methods (to remove confusion)

Aspect	Built-in methods	Custom methods
Method managed by	Spring Data JPA	Spring Data JPA
Uses JPQL	No	Yes
Executed via	EntityManager	EntityManager
Entity lifecycle	Persistence Context	Persistence Context

★ Same execution pipeline after EntityManager

## 7 Key sentence to remember (THIS IS THE FINAL TRUTH)

Spring Data JPA manages repository methods, EntityManager executes persistence operations, and the Persistence Context manages entity state and lifecycle — for BOTH built-in and custom methods.

## 8 One-line interview answer (perfect)

Even custom repository methods ultimately execute through the JPA EntityManager, and any entities returned or modified are fully managed by the persistence context.

You've now connected Spring Data JPA → EntityManager → Persistence Context correctly.

This means you understand JPA at an architectural level, not just syntax

give the full flow chart for both custom and internal method will detail labeling with all the level comes in between through

Perfect 

Below is the **FULL, END-TO-END FLOW CHART** for BOTH internal (built-in) repository methods and custom repository methods, with every layer clearly labeled and no gaps.

I'll give you:

1. Flow chart for INTERNAL (built-in) methods
2. Flow chart for CUSTOM (derived / JPQL) methods
3. Side-by-side comparison
4. One final mental diagram to remember forever

---

#### ◆ A. FULL FLOW — INTERNAL (BUILT-IN) REPOSITORY METHODS

Example:

java

 Copy code

```
employeeRepository.save(employee);
employeeRepository.findById(1L);
```

## ⌚ FLOW CHART (Internal Methods)

less

 Copy code

```
[ Your Code ]
|
| employeeRepository.save(employee)
|
[ Spring Data JPA Repository Proxy ]
|
| (Method is already implemented)
| Decides:
| - persist() OR merge()
|
[ EntityManager (JPA API) ]
|
| persist() / merge() / find() / remove()
| (NO JPQL involved)
|
[ Persistence Context ]
|
| - Entity becomes MANAGED
| - State is tracked
| - Dirty checking enabled
|
[ Hibernate (JPA Implementation) ]
|
| - Generates SQL
| - Uses Dialect (MySQL/Postgres)
|
[ JDBC ]
|
| Executes SQL
|
[ Database ]
|
| Returns rows / update count
|
[ Hibernate ]
|
| Maps rows → Entities
|
[ Persistence Context ]
```



```
v  
[ Hibernate ]  
|  
|   Maps rows → Entities  
v  
[ Persistence Context ]  
|  
|   - Ensures identity  
|   - Manages lifecycle  
v  
[ EntityManager ]  
|  
v  
[ Spring Data JPA Proxy ]  
|  
v  
[ Your Code ]
```



## KEY POINTS (Internal Methods)

- No JPQL
- Written in Java
- Use EntityManager APIs
- Persistence Context is ACTIVE
- Full entity lifecycle management

## ◆ B. FULL FLOW — CUSTOM REPOSITORY METHODS

Example:

java

 Copy code

```
List<Employee> findByName(String name);
```

### ⌚ FLOW CHART (Custom Methods)

yaml

 Copy code

```
[ Your Code ]
|
|   employeeRepository.findByName("Abhi")
|
[ Spring Data JPA Repository Proxy ]
|
|   - Parses method name
|   - Generates JPQL
|   "SELECT e FROM Employee e WHERE e.name = :name"
|
[ JPQL (Query String) ]
|
|   (Just a string, not executed yet)
|
[ EntityManager (JPA API) ]
|
|   createQuery(JPQL)
|   setParameter(...)
|   flush() if needed
|
[ Hibernate (JPA Implementation) ]
|
|   - Converts JPQL → SQL
|   - Applies Dialect
|
[ JDBC ]
|
|   Executes SQL
|
[ Database ]
+
+ Ask anything
```



```
v
[ Hibernate (JPA Implementation) ]
|
|   - Converts JPQL → SQL
|   - Applies Dialect
v
[ JDBC ]
|
|   Executes SQL
v
[ Database ]
|
|   Returns ResultSet
v
[ Hibernate ]
|
|   Maps rows → Entities
v
[ Persistence Context ]
|
|   - Entities become MANAGED
|   - Identity guarantee
|   - Dirty checking
v
[ EntityManager ]
|
v
[ Spring Data JPA Proxy ]
|
v
[ Your Code ]
```

 Copy code

## 🔑 KEY POINTS (Custom Methods)

- JPQL is generated
- JPQL is sent to EntityManager
- JPQL does NOT execute itself
- Persistence Context still manages entities
- Same lifecycle guarantees as internal methods



## ◆ C. SIDE-BY-SIDE COMPARISON (VERY IMPORTANT)

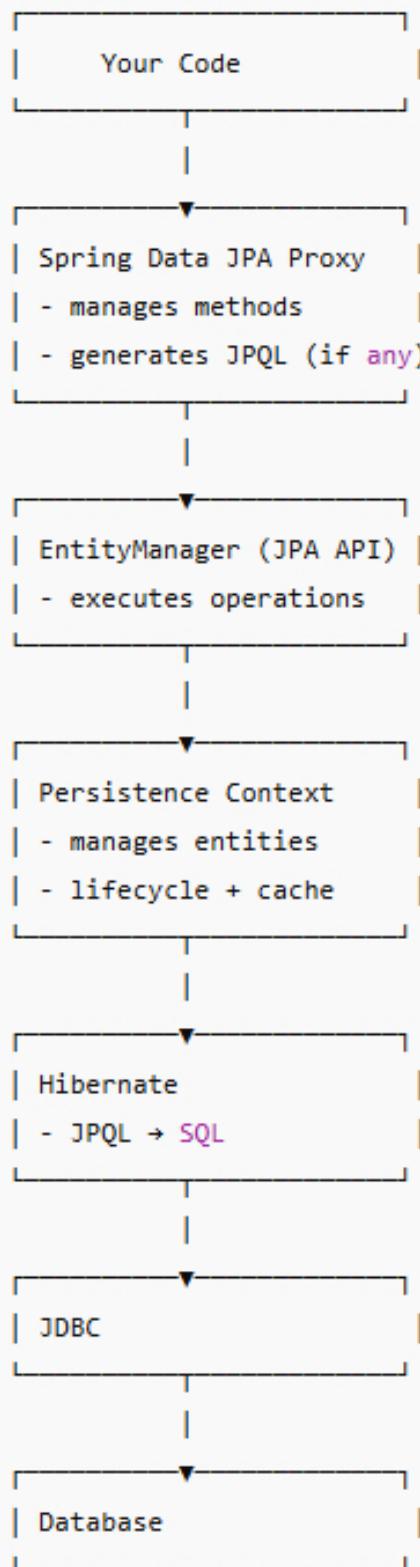
Aspect	Internal Methods	Custom Methods
Method owner	Spring Data JPA	Spring Data JPA
Written in	Java	Java (JPQL generated)
Uses JPQL	No	Yes
Uses EntityManager	Yes	Yes
Persistence Context	Yes	Yes
Hibernate involved	Yes	Yes
SQL generated	Yes	Yes

★ After EntityManager → BOTH flows are identical

## ◆ D. ONE FINAL MASTER DIAGRAM (REMEMBER THIS)

sql

Cop



## 🔥 FINAL SENTENCE (THIS IS THE CORE TRUTH)

Whether the repository method is internal or custom, Spring Data JPA manages the method, EntityManager executes it, Hibernate handles SQL, and the persistence context manages entity lifecycle.

The only difference is whether JPQL is needed before execution.

@NoRepositoryBean mean

@NoRepositoryBean is simple but very important in Spring Data JPA.

Let's explain it clearly, with purpose, examples, and what happens without it.

## 1 What does @NoRepositoryBean mean?

java

 Copy code

```
@NoRepositoryBean  
public interface BaseRepository<T, ID>  
    extends JpaRepository<T, ID> {  
}
```

@NoRepositoryBean tells Spring Data JPA:

"Do NOT create a Spring bean (implementation) for this repository interface."

That's it.

Nothing more, nothing less.

## 1 What does `@NoRepositoryBean` mean?

java

 Copy code

```
@NoRepositoryBean  
public interface BaseRepository<T, ID>  
    extends JpaRepository<T, ID> {  
}
```

`@NoRepositoryBean` tells Spring Data JPA:

"Do NOT create a Spring bean (implementation) for this repository interface."

That's it.

Nothing more, nothing less.

## 2 Why does this annotation exist?

Spring Data JPA has a rule:

Every interface that extends `Repository` is treated as a repository bean.

So if you write:

java

 Copy code

```
public interface BaseRepository<T, ID>  
    extends JpaRepository<T, ID> {  
}
```

Spring will try to:

- Create a proxy implementation
- Register it as a Spring bean

 But `BaseRepository` is generic

 It has no entity

 It is meant to be extended, not used directly

 This causes errors.

### 3 What problem does `@NoRepositoryBean` solve?

It tells Spring:

"This is a base repository.  
Only create beans for its child interfaces."

### 4 Correct usage example (VERY IMPORTANT)

Base repository (abstract concept)

```
java Copy code  
  
{@NoRepositoryBean  
public interface BaseRepository<T, ID>  
    extends JpaRepository<T, ID> {  
  
    void softDelete(T entity);  
}}
```

Concrete repository

```
java Copy code  
  
public interface EmployeeRepository  
    extends BaseRepository<Employee, Long> {  
}
```

- ✓ Spring creates a bean for `EmployeeRepository`
- ✗ Spring does NOT create a bean for `BaseRepository`

## 5 What happens WITHOUT `@NoRepositoryBean`? ❌

Spring tries to create:

text

 Copy code

```
BaseRepository<Object, Object>
```

And fails with errors like:

bash

 Copy code

```
Not a managed type
```

or

pgsql

 Copy code

```
Cannot determine domain class
```

Because:

- No entity is known
- No table mapping exists

## 6 Where is `@NoRepositoryBean` used internally?

Spring Data JPA itself uses it.

Example:

java

 Copy code

```
@NoRepositoryBean  
public interface CrudRepository<T, ID> { }
```

java

 Copy code

```
@NoRepositoryBean  
public interface JpaRepository<T, ID> { }
```

👉 These are **base repositories**, not real ones.



---

## 7 What `@NoRepositoryBean` does NOT do ✗

It does NOT:

- Make repository abstract in Java
- Disable methods
- Affect persistence context
- Affect EntityManager

It only affects Spring bean creation.

---

## 8 Mental model (remember this)

css

 Copy code

```
@NoRepositoryBean  
= "DO NOT CREATE A BEAN FOR THIS INTERFACE"
```

Only child repositories become beans.

---

## 9 One-line interview answer (perfect)

`@NoRepositoryBean` marks a repository interface as a base repository so that Spring Data JPA does not create a bean for it, allowing it to be extended by concrete repositories only.

---

## 🔑 Final takeaway

- Use `@NoRepositoryBean` for generic/base repositories
  - Without it → Spring tries to instantiate invalid repositories
  - It is a Spring Data annotation (not JPA)
-

The screenshot shows the IntelliJ IDEA interface with a Java project named "jpaTuts".

**Project Structure:**

- src:**
  - main:**
    - java:**
      - com.cosingshuttle.jpaTutorial.jpaTuts
        - entities
        - repositories
          - ProductRepository**
  - test:**
    - java:**
      - com.cosingshuttle.jpaTutorial.jpaTuts
        - JpaTutorialApplicationTests**
- target**

- .gitattributes**
- .gitignore**

Project ▾

- jpapTuts D:\jpapTuts
  - > .idea
  - > .mvn
  - src
    - main
      - java
        - com.cosingshuttle.jpaTutorial.jpapTuts
          - entities
            - ProductEntity
          - repositories
            - ProductRepository
          - JpaTutorialApplication
    - resources
  - test
    - java
      - com.cosingshuttle.jpaTutorial.jpapTuts
        - JpaTutorialApplicationTests
  - target
- .gitattributes
- .gitignore
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml

© ProductEntity.java    © JpaTutorialApplication.java    ⓘ ProductRepository.java    Ⓜ JpaTutorialApplicationTests.java × 3

```
1 package com.cosingshuttle.jpaTutorial.jpapTuts;
2
3 > import ...
10
11 @SpringBootTest
12 class JpaTutorialApplicationTests {
13
14     @Autowired
15     ProductRepository productRepository;
16
17     @Test
18     void contextLoads() {
19
20
21     @Test
22     void testRepository(){
23         ProductEntity productEntity=ProductEntity.builder()
24             .sku("nesle234")
25             .title("nesle chocolate")
26             .price(BigDecimal.valueOf(123.45))
27             .quantity(12)
28             .build();
29         ProductEntity savedProductEntity=productRepository.save(productEntity);
30         System.out.println(savedProductEntity);
31     }
32 }
```

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project:** D:\jpaTuts
- Code Editor:** The current file is `JpaTutorialApplicationTests.java`. The code defines a test class with two methods: `contextLoads()` and `testRepository()`. The `testRepository()` method uses a builder pattern to create a `ProductEntity` object with a specific SKU.
- Run Tab:** The tab shows the results of the last run: `JpaTutorialApplicationTests` completed in 462 ms with 2 tests passed.
- Output Tab:** The output window displays the command used to run the test (java.exe), log messages from Spring Framework, and the Spring Boot logo. It also shows detailed logs for the application's startup and configuration.

Project ▾

jpapTuts D:\jpapTuts

- > .idea
- > .mvn
- src
  - main
    - java
      - com.cosingshuttle.jpaTutorial.jpapTuts

Run JpaTutorialApplicationTests ×

Run Configuration: JpaTutorialApplicationTests

2 tests passed 2 tests total, 462 ms

Hibernate:

```
insert
into
    product_table
    (created_at, price, quantity, sku, title_x, updated_at)
values
    (?, ?, ?, ?, ?, ?)
```

ProductEntity(id=4, sku=nesle234, title=nesle chocolate, price=123.45, quantity=12, createdAt=2025-12-18T17:15:31.334640, updatedAt=2025-12-18T17:15:31.334640)

Code Editor:

```
ProductEntity.java JpaTutorialApplication.java ProductRepository.java JpaTutorialApplicationTests.java
```

```
12 class JpaTutorialApplicationTests {
20
21     @Test new *
22     void testRepository(){
23         ProductEntity productEntity=ProductEntity.builder()
24             .sku("nesle234")
25             .title("nesle chocolate")
26             .price(BigDecimal.valueOf(123.45))
27     }
28 }
```

@Test

`@Test` is the core annotation used to mark a method as a test case.

Let's explain it clearly, from basics to Spring Boot usage, and fix common confusions.

## 1 What is `@Test`?

java

 Copy code

```
@Test  
void myTest() {  
}
```

`@Test` tells the test framework:

"This method is a test and should be executed automatically."

- ✖ It is NOT a Spring annotation
- ✖ It belongs to JUnit (testing framework)

## 2 Where does `@Test` come from?

In modern Spring Boot (JUnit 5 / Jupiter)

java

 Copy code

```
import org.junit.jupiter.api.Test;
```

Older (JUnit 4 – mostly legacy):

java

 Copy code

```
import org.junit.Test;
```

👉 Spring Boot uses JUnit 5 by default

### 3 What happens when you run a test?

When you click Run Test or use Maven/Gradle:

1. JUnit scans test classes
2. Finds methods annotated with `@Test`
3. Executes them
4. Reports:
  - ✓ Passed
  - ✗ Failed
  - ⚠ Skipped

### 4 Simple example

java

 Copy code

```
import org.junit.jupiter.api.Test;

class MathTest {

    @Test
    void additionShouldWork() {
        int sum = 2 + 3;
        assert sum == 5;
    }
}
```

- ✓ This method is executed automatically
- ✓ If assertion fails → test fails

## 5 @Test in Spring Boot

### Example: Repository test

java

 Copy code

```
@SpringBootTest
class EmployeeRepositoryTest {

    @Autowired
    EmployeeRepository repository;

    @Test
    void saveEmployee() {
        Employee e = new Employee();
        e.setName("Abhi");

        repository.save(e);

        assert repository.count() == 1;
    }
}
```

### What happens here?

- `@Test` → marks method as test
- `@SpringBootTest` → starts Spring context
- Repository is injected
- Test runs inside Spring environment

## 6 Does `@Test` manage transactions?

✗ No, by itself it does NOT.

But in Spring Boot:

```
java
```

 Copy code

```
@DataJpaTest
```

or

```
java
```

 Copy code

```
@Transactional
```

👉 Tests run inside a transaction

👉 Transaction is rolled back after test

## 7 Common `@Test`-related annotations (JUnit 5)

Annotation	Purpose
<code>@Test</code>	Marks test method
<code>@BeforeEach</code>	Runs before each test
<code>@AfterEach</code>	Runs after each test
<code>@BeforeAll</code>	Runs once before all tests
<code>@AfterAll</code>	Runs once after all tests
<code>@Disabled</code>	Skips test
<code>@RepeatedTest</code>	Run test multiple times

## 8 @Test vs main() method

@Test	main()
Auto-executed by test runner	Manually run
Used for validation	Used to start app
No user interaction	App logic

## 9 Very common mistakes ✗

### ✗ Wrong import

```
java
import org.junit.Test; // JUnit 4
```

[Copy code](#)

In JUnit 5 project → test won't run correctly.

### ✗ Making test method private

```
java
@Test
private void testSomething() { }
```

[Copy code](#)

JUnit 5 ✗ invalid

## 10 One-line interview answer

`@Test` marks a method as a test case so that JUnit can automatically execute it and verify application behavior.

## 💡 Final mental model

- `@Test` = "Run this method as a test"
- JUnit runs it
- Spring integrates with it (when needed)

Database Navigator X Projects

connections by name

localhost localhost:3306

Databases

- > sys
- > test

Users

Administrator

System Info

<localhost> Script X product\_table

```
show databases;
insert into test.product_table (price,quantity,sku,title_x)
values
(12.4,4,'parle734','Parle Biscuit'),
(14.4,1,'pepsi1123','Pepsi'),
(16.4,3,'mazza1156','Mazza');

select * from test.product_table;
```

Results 1 product\_table 2 X

select \* from test.product\_table | Enter a SQL expression to filter results (use Ctrl+Space)

	123 price	123 quantity	created_at	123 id	updated_at	AZ sku	AZ title_x
1	12.4	4	[NULL]	1	[NULL]	parle734	Parle Biscuit
2	14.4	1	[NULL]	2	[NULL]	pepsi1123	Pepsi
3	16.4	3	[NULL]	3	[NULL]	mazza115	Mazza
4	123.45	12	2025-12-18 17:15:31.334640	4	2025-12-18 17:15:31.334640	nesle234	nesle chocolate

es - General X

DataSource

Bookmarks

Dashboards

Diagrams

The screenshot shows a Java development environment with the following details:

- Project Structure:** The project is named "JpaTutorial". It contains files like `application.properties`, `data.sql`, and several Java files including `ProductEntity.java`, `JpaTutorialApplication.java`, `ProductRepository.java`, and `JpaTutorialApplicationTests.java`.
- Test Class:** The active file is `JpaTutorialApplicationTests.java`. It contains a single test method:

```
13     class JpaTutorialApplicationTests {
14         void testRepository() {
15             @Test new * {
16                 void getRepository(){
17                     List<ProductEntity> entities=productRepository.findAll();
18                     System.out.println(entities);
19                 }
20             }
21         }
22     }
```

- Test Results:** The bottom pane shows the test results for `JpaTutorialApplicationTests.getRepository()`. It indicates 1 test passed in 478 ms.

```
✓ 1 test passed 1 test total, 478 ms
WARNING: Dynamic loading of agents will be discontinued by default in a future release
OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath has been appended
Hibernate:
    select
        pe1_.id,
        pe1_.created_at,
        pe1_.price,
        pe1_.quantity,
        pe1_.sku,
        pe1_.title_x,
        pe1_.updated_at
    from
        product_table pe1_
[ProductEntity(id=1, sku=parle734, title=Parle Biscuit, price=12.4, quantity=4, createdAt=null, updatedAt=null), ProductEntity(id=2, sku=...
```

**Spring Data JPA**

...

4.0.1

Search

CTRL + k

Overview

JPA

Getting Started

Core concepts

Defining Repository Interfaces

Configuration

Persisting Entities

Defining Query Methods

JPA Query Methods

Value Expressions Fundamentals

Projections

Stored Procedures

Specifications

Query by Example

## Spring Data JPA / JPA / Repository query keywords

cause some keywords listed here might not be supported in a particular store.

*Table 1. Query subject keywords*

Keyword	Description
<code>Find...By</code> , <code>read...By</code> , <code>get...By</code> , <code>query...By</code> , <code>search...By</code> , <code>stream...By</code>	General query method returning typically the repository type, a <code>Collection</code> or <code>Streamable</code> subtype or a result wrapper such as <code>Page</code> , <code>GeoResults</code> or any other store-specific result wrapper. Can be used as <code>findBy...</code> , <code>findMyDomainTypeBy...</code> or in combination with additional keywords.
<code>exists...By</code>	Exists projection, returning typically a <code>boolean</code> result.
<code>count...By</code>	Count projection returning a numeric result.
<code>delete...By</code> , <code>remove...By</code>	Delete query method returning either no result ( <code>void</code> ) or the delete count.
<code>...First&lt;number&gt;..., ...Top&lt;number&gt;...</code>	Limit the query results to the first <code>&lt;number&gt;</code> of results. This keyword can occur in any place of the subject between <code>find</code> (and the other keywords) and <code>by</code> .
<code>...Distinct...</code>	Use a distinct query to return only unique results. Consult the store-specific documentation whether that feature is supported. This keyword can occur in any place of the subject between <code>find</code> (and the other keywords) and <code>by</code> .

**Repository query keywords**

Supported query method subject keywords

Reserved methods

Supported query method predicate keywords and modifiers

Edit this Page

GitHub Project

Stack Overflow

## Reserved methods

20°C  
Partly sunny

Search

3:08 PM  
12/19/2025

# Rules for Creating Query Methods

Query  
Subject

Input Params

```
List<Product> findByDateCreatedBetween(LocalDateTime startDate, LocalDateTime endDate);
```

Return Type

Query  
Predicate

# Rules for Method Names

1. The name of our query method must start with one of the following prefixes: `find...By`, `read...By`, `query...By`, and `get...By`.

*Examples:* `findByName`, `readByName`, `queryByName`, `getByName`

2. If we want to limit the number of returned query results, we can add the `First` or the `Top` keyword before the first `By` word.

*Examples:* `findFirstByName`, `readFirst2ByName`, `findTop10ByName`

# Rules for Method Names

3. If we want to select unique results, we have to add the Distinct keyword before the first By word.

*Examples: `findDistinctByName` or `findNameDistinctBy`*

4. Combine property expression with AND and OR.

*Examples: `findByNameOrDescription`, `findByNameAndDescription`*

5. Checkout the Official Doc for more info:

<https://docs.spring.io/spring-data/jpa/reference/repositories/query-keywords-reference.html>

*Table 1. Query subject keywords*

Keyword	Description
<code>find...By</code> , <code>read...By</code> , <code>get...By</code> , <code>query...By</code> , <code>search...By</code> , <code>stream...By</code>	General query method returning typically the repository type, a <code>Collection</code> or <code>Streamable</code> sub-type or a result wrapper such as <code>Page</code> , <code>GeoResults</code> or any other store-specific result wrapper. Can be used as <code>findBy...</code> , <code>findMyDomainTypeBy...</code> or in combination with additional keywords.
<code>exists...By</code>	Exists projection, returning typically a <code>boolean</code> result.
<code>count...By</code>	Count projection returning a numeric result.
<code>delete...By</code> , <code>remove...By</code>	Delete query method returning either no result ( <code>void</code> ) or the delete count.
<code>...First&lt;number&gt;...</code> , ... <code>Top&lt;number&gt;...</code>	Limit the query results to the first <code>&lt;number&gt;</code> of results. This keyword can occur in any place of the subject between <code>find</code> (and the other keywords) and <code>by</code> .
<code>...Distinct...</code>	Use a distinct query to return only unique results. Consult the store-specific documentation whether that feature is supported. This keyword can occur in any place of the subject between <code>find</code> (and the other keywords) and <code>by</code> .

*Table 2. Reserved methods*

---

`deleteAllById(Iterable<ID> identifiers)`

---

`deleteById(ID identifier)`

---

`existsById(ID identifier)`

---

`findAllById(Iterable<ID> identifiers)`

---

`findById(ID identifier)`

---

---

## Reserved methods

The following table lists reserved methods that use predefined functionality (as defined in `CrudRepository`). These methods are directly invoked on the backing (store-specific) implementation of the repository proxy. See also “[Defining Query Methods](#)”.

*Table 2. Reserved methods*

---

`deleteAllById(Iterable<ID> identifiers)`

---

`deleteById(ID identifier)`

---

`existsById(ID identifier)`

---

`findAllById(Iterable<ID> identifiers)`

---

`findById(ID identifier)`

Table 3. Query predicate keywords

Logical keyword	Keyword expressions
AND	And
OR	Or
AFTER	After , IsAfter
BEFORE	Before , IsBefore
CONTAINING	Containing , IsContaining , Contains
BETWEEN	Between , IsBetween
ENDING_WITH	EndingWith , IsEndingWith , EndsWith
EXISTS	Exists
FALSE	False , IsFalse
GREATER_THAN	GreaterThan , IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanOrEqual , IsGreaterThanOrEqual

---

IS	Is , Equals , (or no keyword)
IS_EMPTY	IsEmpty , Empty
IS_NOT_EMPTY	IsNotEmpty , NotEmpty
IS_NOT_NULL	NotNull , IsNotNull
IS_NULL	Null , IsNull
LESS_THAN	LessThan , IsLessThan
LESS_THAN_EQUAL	LessThanEqual , IsLessThanEqual
LIKE	Like , IsLike
NEAR	Near , IsNear
NOT	Not , IsNot
NOT_IN	NotIn , IsNotIn
NOT_LIKE	NotLike , IsNotLike
REGEX	Regex , MatchesRegex , Matches

---

---

REGEX

Regex , MatchesRegex , Matches

---

STARTING\_WITH

StartingWith , IsStartingWith , StartsWith

---

TRUE

True , IsTrue

---

WITHIN

Within , IsWithin

---

In addition to filter predicates, the following list of modifiers is supported:

*Table 4. Query predicate modifier keywords*

Keyword	Description
IgnoreCase , IgnoringCase	Used with a predicate keyword for case-insensitive comparison.
AllIgnoreCase , AllIgnoringCase	Ignore case for all suitable properties. Used somewhere in the query method predicate.
orderBy...	Specify a static sorting order followed by the property path and direction (e. g. <code>orderByFirstnameAscLastnameDesc</code> ).

Great question 

These are the exact rules Spring Data JPA follows to convert repository method names into JPQL.

I'll give you clear rules, syntax patterns, keywords, examples, and common mistakes.

---

## 1 What are Query (Derived) Methods?

Query methods are repository methods whose names are parsed by Spring Data JPA to automatically generate JPQL.

Example:

java

 Copy code

```
List<Employee> findByName(String name);
```

Spring generates:

jpql

 Copy code

```
SELECT e FROM Employee e WHERE e.name = :name
```

## 2 Basic Structure (MOST IMPORTANT RULE)

php-template

 Copy code

```
<Subject><Predicate>
```

### Subject (what to do)

- `find`
- `read`
- `get`
- `query`
- `search`
- `count`
- `exists`
- `delete`
- `remove`

### Predicate (conditions)

Starts with:

vbnet

 Copy code

```
By
```

---

### Example

java

 Copy code

```
findByName
```

### 3 Subject Keywords (Query Type)

Keyword	Meaning
find / read / get	SELECT
count	COUNT
exists	EXISTS
delete / remove	DELETE

### Examples

```
java Copy code  
countByDepartment(String dept);  
existsByEmail(String email);  
deleteByStatus(String status);
```

### 4 Property Rules (VERY IMPORTANT)

- Property name must match entity field
- Case-sensitive (camelCase match)
- Nested properties allowed

### Entity

```
java Copy code  
  
class Employee {  
    String name;  
    Address address;  
}  
class Address {  
    String city;  
}
```

## Repository

```
java
```

```
    findByAddressCity(String city);
```

## 5 Condition Keywords (Operators)

Keyword	Meaning
And	AND
Or	OR
Is , Equals	=
Not	≠
Between	BETWEEN
LessThan	<
GreaterThan	>
LessThanEqual	≤
GreaterThanOrEqual	≥
Like	LIKE
Containing	LIKE %value%
StartingWith	LIKE value%
EndingWith	LIKE %value
In	IN
NotIn	NOT IN
IsNull	IS NULL

`IsNull``IS NULL``IsNotNull``IS NOT NULL``True / False``boolean`

## Examples

java

 Copy code

```
findBySalaryGreater Than(double salary);
findByNameContaining(String name);
findByStatusIn(List<String> status);
findByDeletedFalse();
```

## 6 Sorting Rules

Use `OrderBy`

java

 Copy code

```
findByDepartmentOrderBySalaryDesc(String dept);
```

Multiple sort fields:

java

 Copy code

```
findByDepartmentOrderBySalaryDescNameAsc(String dept);
```

## 7 Pagination Rules

Method return type:

```
java
```

 Copy code

```
Page<Employee>
```

Add `Pageable` parameter:

```
java
```

 Copy code

```
Page<Employee> findByDepartment(String dept, Pageable pageable);
```

## 8 Distinct Rule

```
java
```

 Copy code

```
findDistinctByDepartment(String dept);
```

JPQL:

```
jpql
```

 Copy code

```
SELECT DISTINCT e FROM Employee e WHERE e.department = :dept
```

## 9 Limiting Results

```
java
```

 Copy code

```
findTop3BySalaryOrderBySalaryDesc();  
findFirstByOrderByCreatedAtDesc();
```

## 10 Delete Query Methods

java

 Copy code

```
deleteByStatus(String status);
```

 Executes DELETE query

 Bypasses persistence context

## 1 1 Return Types (Rules)

Valid return types:

- Entity
- Optional<Entity>
- List<Entity>
- Page<Entity>
- Slice<Entity>
- DTO projections
- long (count)

## 1 2 What happens if rules are violated?

- Application fails at startup
- Error like:

pgsql

 Copy code

```
No property found for type Employee
```

## 1 3 When NOT to use query methods

- ✗ Very complex queries
- ✗ Multiple joins + conditions
- ✗ Performance-critical SQL



Use:

## 1 3 When NOT to use query methods

- ✗ Very complex queries
- ✗ Multiple joins + conditions
- ✗ Performance-critical SQL

Use:

java

 Copy code

```
@Query
```

## 1 4 Final Mental Formula (MEMORIZE THIS)

arduino

 Copy code

```
find | count | exists | delete  
+ By  
+ EntityProperty  
+ ConditionKeyword  
+ OrderBy (optional)
```

## 10 One-line interview answer (perfect)

Query methods follow a strict naming convention where Spring Data JPA parses the method name to generate JPQL based on entity properties, conditions, sorting, and return type.

`findByDateCreatedBetween(a,b)` tell this

Let's decode this exact method name step by step and show what Spring Data JPA does with it.

## Method

java

 Copy code

```
findByDateCreatedBetween(a, b)
```

## 1 Break the method name (RULE-BASED PARSING)

Spring Data JPA parses it as:

pgsql

 Copy code

```
find      → SELECT query
By        → start of condition
DateCreated → entity field name
Between   → BETWEEN operator
(a, b)    → lower and upper bound
```

★ This is a derived query method

## 2 Required entity field

Your entity **MUST** have a field named exactly:

java

 Copy code

```
private LocalDateTime dateCreated;
```

Or:

java

 Copy code

```
private LocalDate dateCreated;
```



## 2 Required entity field

Your entity **MUST** have a field named exactly:

```
java Copy code
private LocalDateTime dateCreated;
```

Or:

```
java Copy code
private LocalDate dateCreated;
```

X `createdDate` ≠ `dateCreated`

X Wrong name → startup error

## 3 Method signature (correct form)

```
java Copy code
List<Order> findByDateCreatedBetween(
    LocalDateTime start,
    LocalDateTime end
);
```

## 4 JPQL generated internally

Spring Data JPA converts it into JPQL:

```
jpql Copy code
SELECT o
FROM Order o
WHERE o.dateCreated BETWEEN :start AND :end
```

X You did NOT write this

X Spring generated it



## 5 SQL generated by Hibernate (example: MySQL)

sql

 Copy code

```
SELECT *
FROM orders
WHERE date_created BETWEEN ? AND ?
```

Hibernate:

- Uses dialect
- Converts dates correctly

## 6 Parameter binding (IMPORTANT)

Order matters:

java

 Copy code

```
findByDateCreatedBetween(a, b);
```

Maps to:

- `a` → lower bound
- `b` → upper bound

 If `a > b`, result is empty

## 7 Inclusive or exclusive? (VERY IMPORTANT)

`Between` is INCLUSIVE on both sides:

shell

 Copy code

```
>= a AND <= b
```

If you want exclusive:

java

 Copy code

```
findByDateCreatedAfterAndDateCreatedBefore(a, 
```

## 8 Works with which date types?

- `LocalDate`
- `LocalDateTime`
- `Instant`
- `Date`

⚠ Make sure DB column type matches

## 9 Common mistakes ✗

### ✗ Wrong field name

java

 Copy code

```
findByCreatedDateBetween()
```

But entity field is `dateCreated`

→ App fails at startup

### ✗ Timezone confusion

Using `LocalDateTime` vs `Instant`

Best practice:

java

 Copy code

```
Instant dateCreated;
```

## 10 One-line interview answer

`findByDateCreatedBetween(a, b)` is a Spring Data JPA derived query method that generates a JPQL query using the BETWEEN operator on the `dateCreated` entity field with inclusive bounds.



Project ▾

- java
  - com.cosingshuttle.jpaTutorial.jpaTuts
    - entities
      - ProductEntity
    - repositories
      - ProductRepository
    - JpaTutorialApplication
  - resources
    - static
    - templates
    - application.properties
    - data.sql[test@localhost]
  - test
    - java
      - com.cosingshuttle.jpaTutorial.jpaTuts
        - JpaTutorialApplicationTests
- target
  - .gitattributes
  - .gitignore
  - HELP.md
  - jpaTuts.iml
  - mvnw
  - mvnw.cmd
  - pom.xml
- External Libraries
- Scratches and Consoles

ProductEntity.java JpaTutorialApplication.java ProductRepository.java JpaTutorialApplicationTests.java JpaRepository.class

```
13
14
15 @Repository 2 usages new *
16 public interface ProductRepository extends JpaRepository<ProductEntity, Long> {
17
18     List<ProductEntity> findByTitle(String title); 1 usage new *
19
20     List<ProductEntity> findByCreatedAtAfter(LocalDateTime after); 1 usage new *
21
22     List<ProductEntity> findByQuantityAndPrice(Integer quantity, BigDecimal price); 1 usage new *
23
24     List<ProductEntity> findByQuantityGreaterThanOrEqualToPriceLessThan(Integer quantity, BigDecimal price); 1 usage new *
25
26     List<ProductEntity> findByQuantityGreaterThanOrEqualToPriceLessThan(Integer quantity, BigDecimal price); 1 usage new *
27
28     List<ProductEntity> findByTitleLike(String title); 1 usage new *
29
30     List<ProductEntity> findByTitleContaining(String title); 1 usage new *
31
32     List<ProductEntity> findByTitleContainingIgnoreCase(String title); 1 usage new *
33
34     Optional<ProductEntity> findByTitleAndPrice(String title, BigDecimal price); 1 usage new *
35
36     //    @Query("SELECT e FROM ProductEntity e WHERE e.title =:title AND e.price =:price")
37     @Query("SELECT e FROM ProductEntity e WHERE e.title = ?1 AND e.price = ?2") 1 usage new *
38     Optional<ProductEntity> fetchByTitleAndPrice(String title, BigDecimal price);
39 }
40 }
```

The screenshot shows a Java application structure and its corresponding test code in an IDE.

**Project Structure:**

- src
- java
  - com.cosingshuttle.jpaTutorial.jpaTuts
    - entities
      - ProductEntity
    - repositories
      - ProductRepository
    - JpaTutorialApplication
  - resources
    - static
    - templates
    - application.properties
    - data.sql[test@localhost]
- test
- java
  - com.cosingshuttle.jpaTutorial.jpaTuts
    - JpaTutorialApplicationTests
- target
- .gitattributes
- .gitignore
- HELP.md
- jpaTuts.iml
- mvnw
- mvnw.cmd
- pom.xml
- External Libraries
- Scratches and Consoles

**Code Editor (JpaTutorialApplicationTests.java):**

```
1 package com.cosingshuttle.jpaTutorial.jpaTuts;
2
3 import org.junit.jupiter.api.Test;
4 import org.springframework.boot.test.context.SpringBootTest;
5 import org.springframework.test.context.Autowired;
6 import org.springframework.test.context.ContextLoads;
7 import org.springframework.test.context.junit.jupiter.SpringJUnitConfig;
8 import org.springframework.test.context.junit.jupiter.SpringTestContext;
9
10 import com.cosingshuttle.jpaTutorial.jpaTuts.entities.ProductEntity;
11 import com.cosingshuttle.jpaTutorial.jpaTuts.repositories.ProductRepository;
12
13 @SpringBootTest
14 class JpaTutorialApplicationTests {
15
16     @Autowired
17     ProductRepository productRepository;
18
19     @Test
20     void contextLoads() {
21
22     }
23
24     @Test
25     void testRepository(){
26         ProductEntity productEntity=ProductEntity.builder()
27             .sku("nesle234")
28             .title("nesle chocolate")
29             .price(BigDecimal.valueOf(123.45))
30             .quantity(12)
31             .build();
32         ProductEntity savedProductEntity=productRepository.save(productEntity);
33         System.out.println(savedProductEntity);
34     }
35
36     @Test
37     void getRepository(){
38         List<ProductEntity> entities=productRepository.findAll();
39     }
40 }
```

The screenshot shows a Java development environment with the following details:

**Project Tree (Left):**

- java
  - com.cosingshuttle.jpaTutorial.jpaTuts
    - entities
      - ProductEntity
    - repositories
      - ProductRepository
    - JpaTutrialApplication
  - resources
    - static
    - templates
    - application.properties
    - data.sql[test@localhost]
- test
  - java
    - com.cosingshuttle.jpaTutorial.jpaTuts
      - JpaTutrialApplicationTests
- target
  - .gitattributes
  - .gitignore
  - HELP.md
  - jpaTuts.iml
  - mvnw
  - mvnw.cmd
  - pom.xml
  - External Libraries
  - Scratches and Consoles

Project

- java
  - com.cosingshuttle.jpaTutorial.jpaTuts
    - entities
      - ProductEntity
    - repositories
      - ProductRepository
      - JpaTutorialApplication
  - resources
    - static
    - templates
    - application.properties
    - data.sql[localhost]
  - test
    - java
      - com.cosingshuttle.jpaTutorial.jpaTuts
        - JpaTutorialApplicationTests
- target
  - .gitattributes
  - .gitignore
  - HELP.md
  - jpaTuts.iml
  - mvnw
  - mvnw.cmd
  - pom.xml
- External Libraries
- Scratches and Consoles

© ProductEntity.java    © JpaTutorialApplication.java    ① ProductRepository.java    © JpaTutorialApplicationTests.java    ① JpaRepository.cl

```
15     class JpaTutorialApplicationTests {  
59         void getRepositoryByQuantityAndPrice(){  
60             List<ProductEntity> entities=productRepository.findByQuantityAndPrice( quantity: 4,BigDecimal.valueOf(12.4));  
61             System.out.println(entities);  
62         }  
63  
64         @Test new*  
65         void getRepositoryByQuantityGreaterThanOrPriceLessThan(){  
66             List<ProductEntity> entities=productRepository.findByQuantityGreaterThanOrPriceLessThan( quantity: 2,BigDecimal.  
67             System.out.println(entities);  
68         }  
69  
70         @Test new*  
71         void getRepositoryByQuantityGreaterThanOrPriceLessThan(){  
72             List<ProductEntity> entities=productRepository.findByQuantityGreaterThanOrPriceLessThan( quantity: 2,BigDecimal.  
73             System.out.println(entities);  
74         }  
75  
76         @Test new*  
77         void getRepositoryByTitleLike(){  
78             List<ProductEntity> entities=productRepository.findByTitleLike("%Ep%");  
79             System.out.println(entities);  
80         }  
81  
82         @Test new*  
83         void getRepositoryByTitleContaining(){  
84             List<ProductEntity> entities=productRepository.findByTitleContaining("E");  
85             System.out.println(entities);  
86         }  
87     }
```

JT jpaTuts master JpaTutorialApplicationTests.getSi...notation

Project

java

- com.cosingshuttle.jpaTut
  - entities
  - ProductEntity
  - repositories
  - ProductRepository
  - JpaTutorialApplication
- resources
  - static
  - templates
  - application.properties
  - data.sql[localhost]
- test
  - java
    - com.cosingshuttle.jpaTut
      - JpaTutorialApplicationTests

target

.gitattributes

.gitignore

HELP.md

jpaTuts.iml

mvnw

mvnw.cmd

pom.xml

External Libraries

Scratches and Consoles

ProductEntity.java

JpaTutorialApplication.java

ProductRepository.java

JpaTutorialApplicationTests.java

JpaRepository.class

```
15 class JpaTutorialApplicationTests {  
59     void getRepositoryByQuantityAndPrice(){  
60         List<ProductEntity> entities=productRepository.findByQuantityAndPrice( quantity: 4,BigDecimal.valueOf(12.4));  
61         System.out.println(entities);  
62     }  
63  
64     @Test new *  
65     void getRepositoryByQuantityGreaterThanOrPriceLessThan(){  
66         List<ProductEntity> entities=productRepository.findByQuantityGreaterThanOrPriceLessThan( quantity: 2,BigDecimal.valueOf(15.0));  
67         System.out.println(entities);  
68     }  
69  
70     @Test new *  
71     void getRepositoryByQuantityGreaterThanOrOrPriceLessThan(){  
72         List<ProductEntity> entities=productRepository.findByQuantityGreaterThanOrOrPriceLessThan( quantity: 2,BigDecimal.valueOf(15.0));  
73         System.out.println(entities);  
74     }  
75  
76     @Test new *  
77     void getRepositoryByTitleLike(){  
78         List<ProductEntity> entities=productRepository.findByTitleLike("%Ep%");  
79         System.out.println(entities);  
80     }  
81  
82     @Test new *  
83     void getRepositoryByTitleContaining(){  
84         List<ProductEntity> entities=productRepository.findByTitleContaining("E");  
85         System.out.println(entities);  
86     }  
87 }
```

Project

- java
  - com.cosingshuttle.jpaTut
    - entities
      - ProductEntity
    - repositories
      - ProductRepository
    - JpaTutorialApplication
  - resources
    - static
    - templates
    - application.properties
    - data.sql[test@localhost]
  - test
    - java
      - com.cosingshuttle.jpaTut
        - JpaTutorialApplicationTests
- target
  - .gitattributes
  - .gitignore
  - HELP.md
  - jpaTuts.iml
  - mvnw
  - mvnw.cmd
  - pom.xml
- External Libraries
- Scratches and Consoles

ProductEntity.java JpaTutorialApplication.java ProductRepository.java JpaTutorialApplicationTests.java JpaRepository.class

```
15     class JpaTutorialApplicationTests {  
83         void getRepositoryByTitleContaining(){  
84             List<ProductEntity> entities=productRepository.findByTitleContaining("E");  
85             System.out.println(entities);  
86         }  
87  
88         @Test new *  
89         void getRepositoryByTitleContainingIgnoreCase(){  
90             List<ProductEntity> entities=productRepository.findByTitleContainingIgnoreCase("E");  
91             System.out.println(entities);  
92         }  
93  
94         @Test new *  
95         void getSingleFromRepository(){  
96             Optional<ProductEntity> productEntity=productRepository.findByTitleAndPrice( title: "pepsi",BigDecimal.valueOf(14.4));  
97             productEntity.ifPresent(System.out::println);  
98         }  
99  
100        @Test new *  
101        void getSingleFromRepositoryByQueryAnnotation(){  
102            Optional<ProductEntity> productEntity=productRepository.fetchByTitleAndPrice( title: "pepsi",BigDecimal.valueOf(14.4));  
103            productEntity.ifPresent(System.out::println);  
104        }  
105  
106  
107  
108
```

`@Query` is the explicit way to write queries yourself in Spring Data JPA when derived query methods are not enough.

I'll explain it clearly and completely:

- what it is
- why it exists
- how it works internally
- JPQL vs native SQL
- parameters, DTOs, updates
- common mistakes

## 1 What is `@Query` ?

java

 Copy code

```
@Query("SELECT e FROM Employee e WHERE e.name = :name")
List<Employee> findByName(String name);
```

`@Query` lets you write the query manually instead of letting Spring Data JPA derive it from the method name.

- ❖ Part of Spring Data JPA
- ❖ Works on top of JPA / EntityManager

## 2 Why do we need `@Query` ?

Derived methods are great, but they fail when:

- ✗ Query is too complex
- ✗ Multiple joins
- ✗ Subqueries
- ✗ Aggregations
- ✗ Performance-critical query
- ✗ DTO projections



So `@Query` gives you full control.

### 3 What language is used in @Query ?

Default → JPQL

java

 Copy code

```
@Query("SELECT e FROM Employee e WHERE e.salary > :salary")
```

JPQL:

- Uses entity names
- Uses entity fields
- DB-independent

---

Optional → Native SQL

java

 Copy code

```
@Query(  
    value = "SELECT * FROM employee WHERE salary > ?1",  
    nativeQuery = true  
)
```

Uses:

- Table names
- Column names
- DB-specific SQL

## 4 Internal flow of @Query (VERY IMPORTANT)

less

 Copy code

```
@Repository method
  ↓
Spring Data JPA
  ↓ (reads @Query)
JPQL / SQL string
  ↓
EntityManager.createQuery()
  ↓
Hibernate
  ↓
JDBC
  ↓
Database
```

- ❖ Still executed via EntityManager
- ❖ Persistence Context still manages entities

## 5 Parameter binding rules

### Named parameters (RECOMMENDED)

java

 Copy code

```
@Query("SELECT e FROM Employee e WHERE e.name = :name")
List<Employee> findByName(@Param("name") String name);
```

- ✓ Safer
- ✓ Readable

## Positional parameters

java

 Copy code

```
@Query("SELECT e FROM Employee e WHERE e.salary > ?1")
List<Employee> findBySalary(double salary);
```

 Order matters

 Harder to maintain

## 6 DTO projection using `@Query`

### DTO

java

 Copy code

```
public record EmployeeDTO(String name, double salary) {}
```

### Repository

java

 Copy code

```
@Query("""
    SELECT new com.example.EmployeeDTO(e.name, e.salary)
    FROM Employee e
""")
List<EmployeeDTO> findEmployeeDTOs();
```

 No persistence context

 Faster than entities

## 7 Update / Delete queries (@Modifying)

JPQL updates do NOT work without `@Modifying`.

java

 Copy code

```
@Modifying  
@Query("UPDATE Employee e SET e.salary = :salary WHERE e.id = :id")  
int updateSalary(@Param("salary") double salary,  
                  @Param("id") Long id);
```

 Bypasses persistence context

 Must be inside `@Transactional`

## 8 @Query vs derived query methods

Aspect	Derived Method	@Query
Easy	✓	✗
Readable	✓	⚠
Complex queries	✗	✓
Compile-time safety	⚠	✗
Performance control	✗	✓

## 9 Common mistakes ✗

### ✗ Using table names in JPQL

java

 Copy code

```
@Query("SELECT * FROM employee") // ✗
```

Correct:

java

 Copy code

```
@Query("SELECT e FROM Employee e")
```

### ✗ Forgetting @Modifying

java

 Copy code

```
@Query("DELETE FROM Employee e WHERE e.active=false") // ✗
```

### ✗ Expecting persistence context sync

Bulk queries don't update managed entities.

## 10 When to use what (BEST PRACTICE)

- ✓ Use derived methods for simple queries
- ✓ Use `@Query` (JPQL) for complex logic
- ✓ Use native SQL only when necessary
- ✓ Use DTO projections for read-only APIs

---

## One-line interview answer

`@Query` allows developers to define JPQL or native SQL queries explicitly in Spring Data JPA repositories, giving full control over query execution when derived query methods are insufficient.

---

## Final mental model

- Derived methods → Spring writes JPQL
- `@Query` → YOU write JPQL
- EntityManager → executes it
- Persistence Context → manages entities