**a. CrudRepository**

* This is the **basic repository**.
* It gives us all the **simple CRUD operations** (Create, Read, Update, Delete).
* Example: save(), findById(), findAll(), deleteById().
* You use this when your requirement is just **basic database operations**.

**b. PagingAndSortingRepository**

* This is like an **extended version** of CrudRepository.
* It gives **all the CRUD methods** (from CrudRepository) **+ extra features**.
* Extra features are **pagination and sorting**.
* Example: findAll(Pageable pageable), findAll(Sort sort).
* You use this when your project needs **large data handling with pages and sorting results**.

**c. JpaRepository**

* This is the **most powerful repository**.
* It includes **all methods from CrudRepository + PagingAndSortingRepository**.
* Plus, it adds **extra JPA-specific methods** like:
  + flush() (to save changes immediately),
  + saveAll() (to save multiple records at once),
  + getOne() (to get a reference without hitting DB immediately).
* In real-time projects, **JpaRepository is used the most** because it covers almost everything needed for JPA.

**d. Custom Methods**

* Sometimes, your project needs **queries that are not provided by default**.
* Example: You want to find students by name or by age range.
* In such cases, you write **custom methods** in your repository.
* Spring Data JPA can understand method names and generate queries automatically.
  + Example: findByName(String name), findByAgeGreaterThan(int age).
* If that’s not enough, you can also use **@Query** with JPQL or native SQL.

✅ **In real-time projects**:

* Start with CrudRepository (basic).
* If you need pagination/sorting → move to PagingAndSortingRepository.
* Most of the time, developers directly use JpaRepository because it already has everything.
* When built-in methods are not enough → add **custom methods**.

## Why Custom Persistence Operations?

Spring Data JPA gives us **predefined CRUD methods** (save, findAll, deleteById, etc.).  
But in real-time projects, we often need **customized queries** like:

* Find employees whose salary is greater than 50,000.
* Get customers by their city and status.
* Update a product’s price if stock is less than 10.
* Delete users who have been inactive for 2 years.
* Call a stored procedure for complex business logic.

That’s where **Custom Persistence Operations** come in.

# 📌 Custom Persistence Operations in Spring Data JPA (Simple Explanation)

Spring Data JPA helps us interact with the database without writing too much SQL code.  
Normally, we use methods like save(), findById(), delete(), etc. But sometimes, we need **custom queries**.  
That’s where **custom persistence operations** come in.

These operations allow us to:

1. Write queries based on **our own conditions**.
2. Run **HQL (Hibernate Query Language) / JPQL (Java Persistence Query Language)**.
3. Run **Native SQL (direct database queries)**.
4. Do operations like **update, delete, insert, select with conditions**, or even call **stored procedures**.

## 🔹 3 Ways to Write Custom Persistence Operations

### 1) Finder Methods

👉 **What are they?**

* These are custom methods that we write in the **Repository interface**.
* Spring Data JPA will look at the **method name** and **automatically generate a query** for it.
* We don’t need to write SQL, JPA will do it for us.

👉 **What can they do?**

* Fetch complete entity objects.
* Fetch only specific columns (called **Projection**).
* Add multiple conditions like And, Or, Between, In, etc.

👉 **Syntax:**

public <ReturnType> findBy<PropertyName><Condition>(parameters...);

👉 **Examples:**

// Find employee by exact name

Employee findByName(String name);

// Find all employees from a country

List<Employee> findByCountry(String country);

// Find employees from a specific country AND company

List<Employee> findByCountryAndCompany(String country, String company);

// Find employees with salary between two values

List<Employee> findBySalaryBetween(Double min, Double max);

⚡ **Why important?**  
Most real-time projects use **finder methods a lot**, because they are **fast, simple, and easy to read**.

### 2) @Query Methods

👉 **What are they?**

* These allow us to write **our own queries** instead of depending only on method names.
* We can write queries in **JPQL (Java Persistence Query Language)** or **Native SQL (direct DB query)**.

👉 **Why use this?**  
Sometimes finder methods **cannot handle complex queries**, so we write them ourselves.

👉 **JPQL Example:**

@Query("SELECT e FROM Employee e WHERE e.country = ?1")

List<Employee> getEmployeesByCountry(String country);

* Here, Employee e refers to the **Entity class**, not the table.
* ?1 means the first parameter in the method.

👉 **Native SQL Example:**

@Query(value = "SELECT \* FROM employee\_list WHERE country = ?1", nativeQuery = true)

List<Employee> fetchByCountryNative(String country);

* Here we write **real SQL** directly.
* nativeQuery = true tells Spring that this is not JPQL but actual SQL.

⚡ **Why important?**  
In industry, when queries become **too complex**, we use @Query.

### 3) @Query + @Modifying Methods

👉 **What are they?**

* These are used for **non-select operations** (Update, Delete, Insert).
* We must use @Modifying because we are **changing the database**.
* We also need @Transactional to make sure the update/delete happens in a **transaction**.

👉 **Examples:**

✅ **Update salary by employee name**

@Modifying

@Transactional

@Query("UPDATE Employee e SET e.salary = :salary WHERE e.name = :name")

int updateEmployeeSalary(String name, Double salary);

* This will update the salary of an employee who has the given name.
* :name and :salary are method parameters.

✅ **Delete employees by country**

@Modifying

@Transactional

@Query("DELETE FROM Employee e WHERE e.country = :country")

int deleteByCountry(String country);

* This will delete all employees from a given country.

⚡ **Why important?**  
In real projects, we often need to **bulk update or bulk delete** records.  
Instead of fetching and looping through them, we write a **direct query** with @Modifying.

## 🔹 Quick Comparison

| **Mechanism** | **What it does** | **When to use** | **Industry Usage** |
| --- | --- | --- | --- |
| **Finder Methods** | Auto-generate SELECT queries based on method names | Simple queries like findByName | ✅ Used most often |
| **@Query** | Write custom SELECT queries (JPQL or SQL) | Complex queries finder methods can’t handle | ✅ Very common |
| **@Query + @Modifying** | Run UPDATE, DELETE, INSERT queries | When we need to modify DB records directly | ✅ Common for bulk updates/deletes |

✅ **Summary for Freshers:**

* If your query is **simple**, always try a **finder method** first.
* If your query is **complex**, use @Query.
* If you need to **change data** (update, delete), use @Query + @Modifying.

## Mechanisms Available

### ****a. Finder Methods (Derived Query Methods)****

* Finder methods are **custom abstract methods** we declare inside the Repository interface.
* Spring Data JPA will **analyze the method name** and **automatically generate the SQL/HQL** query for us.
* We don’t write the query — Spring generates it by looking at our method name.

## 1️⃣ **Entity Select (return full entity object)**

* Here, the method returns the **complete entity object** (all columns of the table).
* Example:

List<Student> findByCity(String city);

👉 Query generated:

SELECT \* FROM student WHERE city = ?;

✔ Returns **all columns** of Student (id, name, marks, city etc.) as **Student objects**.

* Best when you **need the complete row**.
* Common in real-time apps when you show **complete details** (e.g., "show me all students from Hyderabad").

## 2️⃣ **Scalar Select (Projection → only some columns)**

* Sometimes we **don’t need all columns**.
* Example: A dashboard may only need **student names and marks**, not their address or phone.
* Returning fewer columns is **faster and lighter**.

### Two ways to do Projection in Spring Data JPA:

**a. Return a single column directly**

List<String> findByCity(String city);

👉 Query generated:

SELECT city FROM student WHERE city = ?;

**b. Return multiple selected columns using Projection (DTO or Interface)**

interface StudentView {

String getName();

Integer getMarks();

}

List<StudentView> findByCity(String city);

👉 Query generated:

SELECT name, marks FROM student WHERE city = ?;

✔ This is called **Projection** → you are projecting **only part of the entity**.  
✔ Very common in industry when **loading data for reports or dashboards** where not all columns are required.

## 3️⃣ **Multiple Conditions with Keywords (And, Or, In, Between, etc.)**

Spring Data JPA lets us create **complex queries just by adding keywords** in the method name.

### Examples:

**a. Using And**

List<Student> findByCityAndMarks(String city, Integer marks);

👉 Query:

SELECT \* FROM student WHERE city = ? AND marks = ?;

**b. Using Or**

List<Student> findByCityOrMarks(String city, Integer marks);

👉 Query:

SELECT \* FROM student WHERE city = ? OR marks = ?;

**c. Using In**

List<Student> findByCityIn(List<String> cities);

👉 Query:

SELECT \* FROM student WHERE city IN (?, ?, ...);

✔ Useful when filtering data across **multiple values** (e.g., "All students from Hyderabad, Chennai, Bangalore").

**d. Using Between**

List<Student> findByMarksBetween(Integer start, Integer end);

👉 Query:

SELECT \* FROM student WHERE marks BETWEEN ? AND ?;

✔ Handy for **ranges**, e.g., "Students with marks between 70 and 90".

**e. Using LessThan / GreaterThan**

List<Student> findByMarksGreaterThan(Integer marks);

List<Student> findByMarksLessThan(Integer marks);

👉 Queries:

SELECT \* FROM student WHERE marks > ?;

SELECT \* FROM student WHERE marks < ?;

## 🔑 Key Insights for Real-Time Projects

* **Entity Select** → Use when you need **full entity** (all details).
* **Scalar Select (Projection)** → Use when you only need **some columns** (better performance).
* **Multiple Conditions** → Easily express real-world filters using method names instead of writing SQL manually.

✔ **Implementation**

* Spring internally creates an **in-memory proxy class** (at runtime) that provides the logic for our finder method.
* We just declare the method, Spring handles the query.

✔ **Default Condition**

* If we just say findByProperty, Spring applies **equals (=)** by default.

## **Syntax**

public <ReturnType> findBy<PropertyName><Condition>(params...);

### Examples

1. **Simple equals condition (default):**

List<Student> findByCity(String city);

// SELECT \* FROM student WHERE city = ?

1. **Multiple conditions with AND:**

List<Student> findByCityAndMarks(String city, Integer marks);

// SELECT \* FROM student WHERE city = ? AND marks = ?

1. **Using OR:**

List<Student> findByCityOrMarks(String city, Integer marks);

// SELECT \* FROM student WHERE city = ? OR marks = ?

1. **Using greater than:**

List<Student> findByMarksGreaterThan(Integer marks);

// SELECT \* FROM student WHERE marks > ?

1. **IN clause:**

List<Student> findByCityIn(List<String> cities);

// SELECT \* FROM student WHERE city IN (?,?,?)

1. **Projection (only selected columns):**

List<String> findByCity(String city);

// SELECT city FROM student WHERE city = ?

## **Key Points to Remember**

* Finder methods are **only for SELECT operations**.
* Naming convention is very important — Spring builds queries based on method names.
* No need to write SQL/HQL, Spring does it automatically.
* Works well for **simple queries**, but for **complex queries**, we use @Query or @Query + @Modifying.

✅ In short:

* Finder methods let us **quickly write queries just by defining method names**.
* They are **auto-implemented by Spring** at runtime.
* Default comparison is **equals (=)**, but we can add conditions like GreaterThan, LessThan, Between, etc.

# 📖 Spring Data JPA Predefined Method Keywords

## 1️⃣ Starting Keywords (Prefixes)

Used to define the **intent** of the query.

* find…By
* read…By
* get…By
* query…By
* search…By
* stream…By
* exists…By
* count…By
* delete…By
* remove…By

✅ Example:

List<Employee> findByCountry(String country);

boolean existsByName(String name);

long countByCompany(String company);

## 2️⃣ Conditional Keywords (WHERE Conditions)

Used for **filtering** records.

* And, Or
* Is, Equals
* Between
* LessThan, LessThanEqual
* GreaterThan, GreaterThanEqual
* After, Before
* IsNull, IsNotNull
* Not
* In, NotIn
* True, False

✅ Example:

List<Employee> findBySalaryBetween(Double min, Double max);

List<Employee> findByCountryAndCompany(String country, String company);

## 3️⃣ String Matching Keywords (LIKE Clauses)

Used for **partial matching**.

* StartingWith
* EndingWith
* Containing (like %value%)
* Like, NotLike
* IgnoreCase

✅ Example:

List<Employee> findByNameStartingWith(String prefix);

List<Employee> findByCompanyContainingIgnoreCase(String keyword);

## 4️⃣ Ordering Keywords

Used for **sorting results**.

* OrderBy…Asc
* OrderBy…Desc

✅ Example:

List<Employee> findByCountryOrderBySalaryDesc(String country);

## 5️⃣ Limiting Results (Top/First)

Used to **fetch limited records**.

* Top<number>
* First<number>

✅ Example:

Employee findTopByOrderBySalaryDesc(); // Highest paid employee

List<Employee> findFirst3ByCountry(String country); // First 3 employees

## 6️⃣ Special Keywords

Extra useful options.

* Distinct → removes duplicates
* Exists → checks if record exists

✅ Example:

List<Employee> findDistinctByCompany(String company);

boolean existsByPosition(String position);

📌 **How to Build a Method Name**  
👉 [Prefix] + [PropertyName] + [Keyword]  
👉 Example:

List<Employee> findTop5ByCountryAndSalaryGreaterThanOrderBySalaryDesc(String country, Double minSalary);

Eg: SpringDataJpaCustomMethods

# 📌 Mostly Used Spring Data JPA Methods in Industry

## 1️⃣ Basic Finders

Used **everywhere** to fetch entities by a property.

* findBy…
* getBy…
* readBy…

✅ Example:

Employee findByName(String name);

List<Employee> findByCountry(String country);

## 2️⃣ Conditional Filters

Very common when filtering with multiple conditions.

* And, Or
* GreaterThan, LessThan, Between
* IsNull, IsNotNull
* In

✅ Example:

List<Employee> findByCountryAndCompany(String country, String company);

List<Employee> findBySalaryBetween(Double min, Double max);

List<Employee> findByIdIn(List<Long> ids);

## 3️⃣ String Search (LIKE queries)

Frequently used in **search features**.

* Containing (most popular, LIKE %value%)
* StartingWith (LIKE value%)
* EndingWith (LIKE %value)
* IgnoreCase (for case-insensitive search)

✅ Example:

List<Employee> findByNameContainingIgnoreCase(String keyword);

## 4️⃣ Sorting Results

Common for listing ordered data.

* OrderBy…Asc
* OrderBy…Desc

✅ Example:

List<Employee> findByCountryOrderBySalaryDesc(String country);

## 5️⃣ Limiting Results (Top/First)

Used in **dashboards, leaderboards, reports**.

* Top1, TopN
* FirstN

✅ Example:

Employee findTopByOrderBySalaryDesc(); // Highest paid employee

List<Employee> findTop5ByCountryOrderBySalaryDesc(String country);

## 6️⃣ Count & Exists

Very common for **validations, reports, analytics**.

* countBy…
* existsBy…

✅ Example:

long countByCountry(String country);

boolean existsByName(String name);

## 7️⃣ Distinct

Used when **removing duplicates** is necessary.

* Distinct

✅ Example:

List<Employee> findDistinctByCompany(String company);

# 🎯 Industry Usage Summary (Most Popular)

👉 **CRUD basics** → findBy, save, deleteById, findAll  
👉 **Filters** → And, Or, Between, In, GreaterThan  
👉 **Search** → Containing, IgnoreCase  
👉 **Sorting** → OrderBy...Desc/Asc  
👉 **Limiting** → Top1, Top5  
👉 **Validation/Reports** → existsBy, countBy

Eg: SpringDataJpaCustomMethods-1