**📒 Notes – deleteById() Example**

**1. Purpose of deleteById()**

* deleteById(ID id) is a method from **JpaRepository**.
* It deletes a record from the database using the **primary key (ID)** value.
* If no record exists with that ID, **Spring Data JPA will throw an exception** (EmptyResultDataAccessException) — so checking first is a good idea.

**2. The Code Flow (Simple English)**

1. The user (or code) provides an **ID** to delete.
2. We first check if the student with that ID exists:
3. if (studentRepo.existsById(id)) { ... }
4. If it exists:
   * Call studentRepo.deleteById(id) → Spring creates a SQL DELETE query and executes it.
5. If it does not exist:
   * Show a message that the record was not found.

**3. What does studentRepo.existsById(id) do?**

* existsById(ID id) is another JpaRepository method.
* It **checks the database** to see if a record with the given ID exists.
* Internally, it runs a query similar to:
* SELECT COUNT(\*) FROM student\_master WHERE id = ?
* If count > 0 → returns true.
* If count = 0 → returns false.
* **Purpose:** Prevents deleting a non-existing record and avoids exceptions.

**4. Key Points to Remember**

* ✅ **Always check** with existsById() before deleteById() in real projects — helps avoid errors and better user messages.
* ✅ deleteById() only needs the **ID value** — not the whole object.
* ✅ If your entity has foreign key relationships (like student has courses), **deletion might fail** unless cascading is properly set.
* ✅ Works in a **transaction** — Spring Data handles it for you.

**5. Example Flow Diagram**

Start

↓

Get ID to delete

↓

Check existsById(id)

↓ Yes? ───────────────→ No?

↓ ↓

deleteById(id) Print "No record found"

↓

Success message

↓

End

**6. Why Industry Uses This Approach**

* Reduces **unnecessary SQL DELETE queries** when the record is not present.
* Provides **clearer logs** and **user-friendly messages**.
* Helps avoid EmptyResultDataAccessException in production.

Eg: SpringBootDataJpaDeleteById

## ****Notes for**** deleteAll() ****in Spring Data JPA****

### 1. ****Purpose of**** deleteAll()

* This method removes multiple records from the database in a single call.
* It accepts a **collection of entity objects** (not IDs directly).
* Hibernate internally uses the **primary key** from each entity object to execute a DELETE query.

### 2. ****Two ways to delete rows using**** deleteAll()

#### **✅ Works — Delete by Primary Key only (Minimal data)**

* You can create entity objects with only the primary key (ID) set.
* Other columns don’t need to be set (can remain null or unset).
* Hibernate will only consider the primary key for deletion.

// Works — primary key only

Student s1 = new Student();

s1.setId(1L);

Student s2 = new Student();

s2.setId(2L);

studentRepo.deleteAll(Arrays.asList(s1, s2));

**What happens internally**:  
Hibernate generates SQL like:

delete from student\_master where id=?

delete from student\_master where id=?

#### **❌ Does NOT work — Passing only IDs to** deleteAll()

* deleteAll() expects **entity objects**, not just ID values.
* If you try to pass IDs directly, you get a compile-time error.

// ❌ Does not work — Compile Error

List<Long> ids = Arrays.asList(1L, 2L);

studentRepo.deleteAll(ids); // ERROR: expects Iterable<Student>, not Iterable<Long>

#### **✅ Alternative — Delete by IDs using** deleteAllById()

* If you have only IDs, use deleteAllById() instead of deleteAll().

// Works — deleting multiple rows by only IDs

studentRepo.deleteAllById(Arrays.asList(1L, 2L));

**What happens internally**:  
Hibernate generates SQL like:

delete from student\_master where id in (?, ?)

### 3. ****Key Takeaways****

* deleteAll() → Requires **entity objects** (primary key is enough; no need for full row data).
* deleteAllById() → Directly accepts a list of primary key values.
* In both methods, Hibernate deletes rows based on the **primary key** only.
* You **don’t** need to set other columns, and you **should not** set them incorrectly (it won’t be used in deletion, but setting wrong values may cause confusion).

Eg: SpringBootDataJpaDeleteAllInterable

Eg: SpringBootDataJpaDeleteAllById

**📝 Notes for count() Example**

1. **What is count()?**
   * count() is a predefined method in JpaRepository.
   * It **returns the total number of records (rows)** present in the table associated with the entity.
   * Return type is **long**.
2. **Usage in our Example**
3. long count = studentRepo.count();
   * studentRepo is an object of StudentRepository, which extends JpaRepository<Student, Long>.
   * When we call count(), Spring Data JPA will generate the SQL query:
   * SELECT COUNT(\*) FROM student\_master;
   * The result of this SQL query (total number of rows) is stored in the variable count.
4. **Why do we need it?**
   * To quickly find out how many records are present in the database without fetching all of them.
   * This is **efficient** because it doesn’t load full entity objects, it just executes a simple COUNT(\*).
5. **Output Example**  
   If your student\_master table contains 3 rows like below:
6. id | name | course | fee
7. --------------------------------
8. 1 | Raju | Java | 2000
9. 2 | Sita | Python | 3000
10. 3 | Ravi | Spring | 2500

Then studentRepo.count(); will return:

Total number of students: 3

1. **Advantages of count()**
   * Very fast since it runs a COUNT(\*) query instead of fetching entire rows.
   * Useful in pagination, statistics, or reports.
   * Helps in checking database size before performing bulk operations.

✅ **Summary**:  
count() → Gives total row count in the table.  
It runs SQL: SELECT COUNT(\*) FROM <table\_name> internally and returns the result as long.

Eg: SpringBootDataJpaCount

**📝 Notes on PagingAndSortingRepository**

1. **What is it?**
   * PagingAndSortingRepository is an interface in Spring Data JPA.
   * It extends CrudRepository.
   * It provides **extra methods** for:
     + **Sorting** records (ascending or descending order).
     + **Pagination** (fetching large records in smaller chunks – page by page).
2. **Why do we need it?**
   * When we have **huge amounts of data**, showing everything at once is inefficient.
   * Pagination helps to load data **page by page** (e.g., 10 rows at a time).
   * Sorting helps to display data in **order** (like names A → Z, or fees high → low).
3. **Sorting Order in Java**  
   If we sort in **ascending order**, the sequence will be:
   * First: **Special characters** (\*, ?, -, ...)
   * Second: **Numbers** (0–9)
   * Third: **Uppercase letters** (A–Z)
   * Fourth: **Lowercase letters** (a–z)

👉 Descending order will simply reverse this.

1. **Important Method**

Iterable<T> findAll(Sort sort);

* + findAll() here is an **overloaded version** (not the same as CrudRepository’s findAll()).
  + It accepts a Sort object as parameter.
  + The Sort object tells Spring JPA **which property** (column) to sort by and whether it should be **ASCENDING or DESCENDING**.

1. **How to use it (example idea)**

// Sort students by "name" in ascending order

List<Student> students = studentRepo.findAll(Sort.by("name").ascending());

// Sort students by "fee" in descending order

List<Student> students = studentRepo.findAll(Sort.by("fee").descending());

// Sort students by multiple properties: first "course", then "name"

List<Student> students = studentRepo.findAll(Sort.by("course").ascending()

.and(Sort.by("name").descending()));

* + Here studentRepo is extending PagingAndSortingRepository.
  + Depending on the Sort object, JPA internally generates SQL like:
  + SELECT \* FROM student\_master ORDER BY name ASC;
  + SELECT \* FROM student\_master ORDER BY fee DESC;
  + SELECT \* FROM student\_master ORDER BY course ASC, name DESC;

1. **When to use it in real projects**
   * Whenever you need to show a **list of records to end users** (example: employee list, product list, student list).
   * You almost always use **pagination + sorting** together to make UI faster and cleaner.

✅ **Summary**:

* PagingAndSortingRepository = CrudRepository + Sorting + Pagination features.
* Method: findAll(Sort sort) → sorts records based on property/properties in ASC or DESC.
* Uses Sort object to define sorting rules.

# 📝 Notes for findAll(Sort sort) Example

### 1. ****What is**** PagingAndSortingRepository****?****

* It is an extension of CrudRepository.
* Provides **extra features** like:
  + **Pagination** (splitting results into pages).
  + **Sorting** (order results ASC or DESC based on properties).

### 2. ****Sorting Concept****

* Sorting means arranging data in a specific order.
* By default, sorting can be done in:
  + **Ascending (ASC):** Small → Large (A → Z, 0 → 9).
  + **Descending (DESC):** Large → Small (Z → A, 9 → 0).

### 3. ****Method Used****

Iterable<T> findAll(Sort sort)

* Accepts a Sort object as parameter.
* Sort tells Spring Data JPA **how to sort the data**.
* Works with both single and multiple properties.

### 4. ****Examples from Code****

1. **Sort by name in ascending order**

studentRepo.findAll(Sort.by("name").ascending());

* Fetches all students sorted by name alphabetically (A → Z).

1. **Sort by fee in descending order**

studentRepo.findAll(Sort.by("fee").descending());

* Fetches all students with highest fee first → lowest fee last.

1. **Sort by multiple properties**

studentRepo.findAll(

Sort.by("course").ascending()

.and(Sort.by("name").descending()));

* First sorts by course (ASC).
* If two students have the same course → sorts them by name in DESC.

### 5. ****Printing Records****

students.forEach(System.out::println);

* Loops through the students list.
* Uses **method reference** to call System.out.println(student) for each object.
* Each student’s details (from toString()) will be printed line by line.

### 6. ****Where We Wrote the Code****

* All logic is written inside **Service class methods**.
* The **Main Application** only calls service methods → makes the code **cleaner and reusable**.

Eg: SPringBootDataJpaFindAllSort

# 📘 Notes for findAll(Pageable pageable) Example

### 🔹 1. What is Pagination?

* When we have **huge records** in a table, fetching them all at once is not efficient.
* **Pagination** helps to fetch records **page by page** (like browsing search results on Google).
* This improves **performance** and **memory usage**.

### 🔹 2. Repository Used

public interface StudentRepository extends PagingAndSortingRepository<Student, Long> { }

* Here, we extend PagingAndSortingRepository.
* It provides the method:
* Page<T> findAll(Pageable pageable);

which supports pagination.

### 🔹 3. Service Layer Logic

PageRequest pageable = PageRequest.of(pageNo, pageSize);

Page<Student> page = studentRepo.findAll(pageable);

List<Student> students = page.getContent();

students.forEach(System.out::println);

**Explanation:**

1. PageRequest.of(pageNo, pageSize) →
   * pageNo = which page you want (0-based index).
   * pageSize = how many records per page.

Example: PageRequest.of(0, 2) → first **2 records**.

1. studentRepo.findAll(pageable) →
   * Executes SQL query with **LIMIT** & **OFFSET** internally.
   * Returns a Page<Student> object.
2. page.getContent() →
   * Extracts the actual list of students from the page.
3. students.forEach(System.out::println) →
   * Prints each student.
   * **Method reference** used here → shortcut for writing a loop.

### 🔹 4. Pagination Extra Info

We can also print useful details:

System.out.println("Total Pages : " + page.getTotalPages());

System.out.println("Total Records : " + page.getTotalElements());

System.out.println("Current Page Number : " + page.getNumber());

System.out.println("Page Size : " + page.getSize());

* getTotalPages() → How many pages exist in total.
* getTotalElements() → Total number of records in the table.
* getNumber() → Current page number.
* getSize() → Page size (number of records per page).

### 🔹 5. Example Output (Assume 5 records in DB, pageSize = 2)

Page 0:

Student [id=1, name=Ram, course=Java, fee=5000.0]

Student [id=2, name=Sam, course=Python, fee=6000.0]

Total Pages : 3

Total Records : 5

Current Page Number : 0

Page Size : 2

Page 1:

Student [id=3, name=John, course=React, fee=7000.0]

Student [id=4, name=Kiran, course=Spring, fee=8000.0]

Total Pages : 3

Total Records : 5

Current Page Number : 1

Page Size : 2

### 🔹 6. Industry Use

✅ Very commonly used in **real-time projects** where data is large (e.g., e-commerce product listings, user search results).  
✅ Reduces **load on DB & memory**.  
✅ Provides a **user-friendly way** to view large data sets.

Eg:SpringBootDataJpaFindAllPageable

# 📘 Notes on Optional Methods

### 1. isPresent()

* Checks if a value is inside the Optional.
* Returns **true** if value is present, **false** if empty.
* Often used with get().

👉 Example meaning:  
“If a student exists in DB, return true.”

### 2. isEmpty()

* Opposite of isPresent().
* Returns **true** if value is missing, **false** if present.

👉 Example meaning:  
“If no student found, return true.”

### 3. get()

* Returns the actual value stored inside Optional.
* Should be used **only after** checking isPresent() (otherwise throws NoSuchElementException).

👉 Example meaning:  
“Get me the student object if it is available.”

### 4. orElse(defaultValue)

* If value is present → returns that value.
* If value is missing → returns the given default value.

👉 Example meaning:  
“If no student found, return a dummy/default student.”

### 5. orElseGet(Supplier)

* Similar to orElse(), but creates the default object **only when needed** (lazy execution).
* Useful when default value creation is expensive (e.g., database call, file read).

👉 Example meaning:  
“If no student found, then and only then create a default student.”

### 6. orElseThrow(ExceptionSupplier)

* If value is present → return it.
* If missing → throw a custom exception.

👉 Example meaning:  
“If student not found, throw StudentNotFoundException.”

### 7. ifPresent(Consumer)

* Runs a piece of code **only if value is present**.
* Avoids writing explicit if conditions.

👉 Example meaning:  
“If student exists, print the student’s name.”

### 8. map(Function)

* Transforms the value inside the Optional if it exists.
* Returns another Optional with the transformed value.

👉 Example meaning:  
“If student exists, take only the name field from it.”

### 9. filter(Predicate)

* Checks a condition on the value inside the Optional.
* If condition passes → returns same Optional.
* If condition fails → returns empty Optional.

👉 Example meaning:  
“If student exists and fee > 5000, return the student, else return empty.”

# 📝 Quick Summary Table

| **Method** | **Meaning in Simple Words** |
| --- | --- |
| isPresent() | Check if value is available |
| isEmpty() | Check if value is missing |
| get() | Get the actual value (only safe after checking) |
| orElse() | Give real value or a default one |
| orElseGet() | Give real value or create a default only when needed |
| orElseThrow() | Give real value or throw an error |
| ifPresent() | Do something only when value exists |
| map() | Transform the value if it exists |
| filter() | Keep the value only if condition is true |

Eg: SpringBootDataJpaOptional.