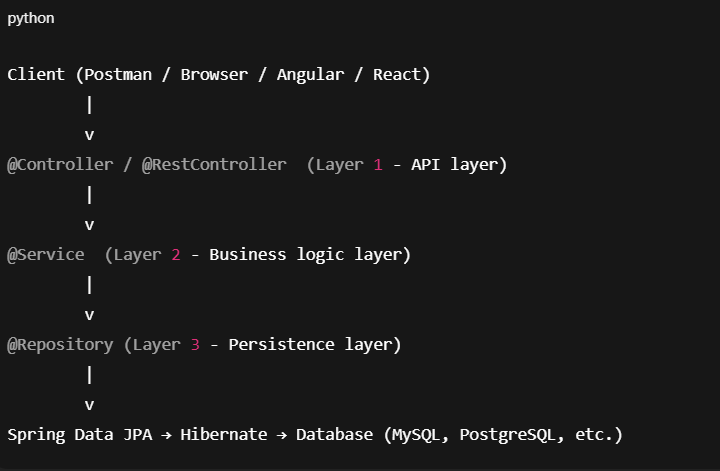
Spring JPA

Jpa-> Java Persistence API



**Step by Step:**

1. Client sends an HTTP request (e.g., POST /students with JSON body).
2. @RestController handles the request and calls the **Service layer**.
3. @Service contains business logic and calls the **Repository**.
4. @Repository interacts with the DB using **Spring Data JPA** (JpaRepository).
5. JPA delegates to Hibernate which generates SQL queries.
6. DB executes queries → results go back in reverse flow.

**JPA / Hibernate Annotations**

**Entity Definition**

* @Entity → Marks a class as a JPA entity.
* @Table(name = "table\_name") → Maps entity to DB table.
* @Id → Primary key.
* @GeneratedValue(strategy = …) → PK auto generation.

**Column Mapping**

* @Column(name = "col\_name", unique = true, nullable = false) → Customize column.
* @Enumerated(EnumType.STRING) → Store enums as text.

**Relationships**

* @OneToOne(mappedBy = …, cascade = …)
* @OneToMany(mappedBy = …, cascade = …)
* @ManyToOne
* @ManyToMany + @JoinTable → Many-to-Many with join table.

**Fetch Types**

* FetchType.LAZY → Loads data **on demand** (default in collections).
* FetchType.EAGER → Loads related data immediately.

**Cascade Types**

* CascadeType.PERSIST → Save child when parent is saved.
* CascadeType.MERGE → Update child when parent is updated.
* CascadeType.REMOVE → Delete child when parent is deleted.
* CascadeType.ALL → Apply all cascades.

**Validation Annotations (from jakarta.validation)**

* @NotNull → Field cannot be null.
* @NotEmpty → Field cannot be null/empty (for String/Collection).
* @Size(min = , max = ) → Length constraint.
* @Pattern(regexp = …) → Regex validation.
* @Email → Email format validation.
* @Min / @Max → Numeric range validation.

**📍 Transaction Management**

* @Transactional → Method/class runs inside a transaction.

**📍 Spring Data Repository**

* JpaRepository<T, ID> → CRUD + JPA methods.
* @Query("JPQL query") → Custom queries.
* **@Modifying + @Transactional → For update/delete queries.**

**------------------------------------------------------------------------------------**

* **Owning side = has the foreign key / join table** → needs @JoinColumn or @JoinTable.
* **Inverse side = just references back** → needs only mappedBy.

**One-to-One (1:1)**

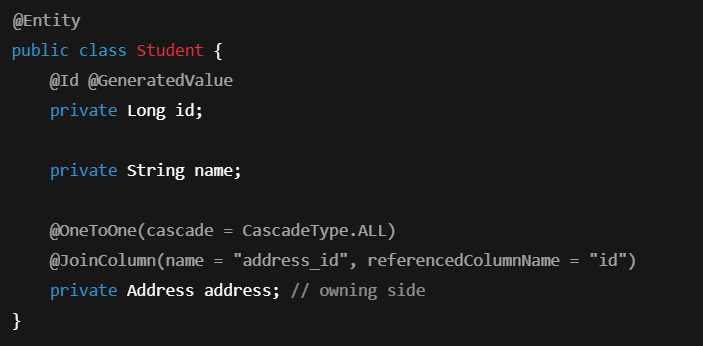
👉 **Definition**: One entity is associated with exactly one other entity.

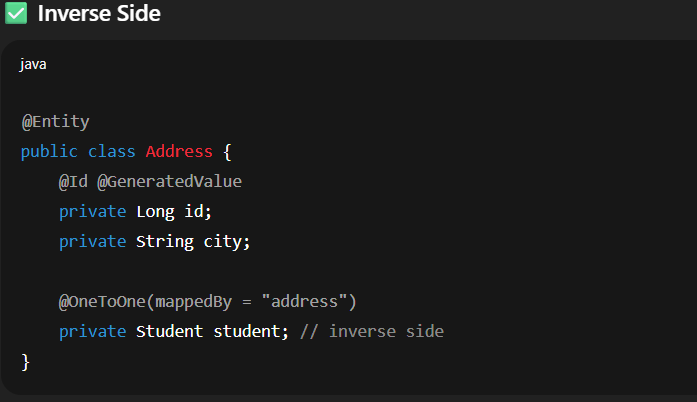
Example: **Student ↔ Address**

* Each student has **one address**
* Each address belongs to **one student**

**✅ Owning Side**

* **Owning side = the table that has the foreign key column**.
* Put @JoinColumn on **the owning side only**.





Use mappedBy only on the **inverse side** (no @JoinColumn here).

Use cascade = CascadeType.ALL if you want saving Student → saves Address too.

To avoid **recursion error** in JSON, use:

* @JsonManagedReference on owning side
* @JsonBackReference on inverse side  
  OR
* @JsonIgnoreProperties("student")

**Many-to-One (N:1)**

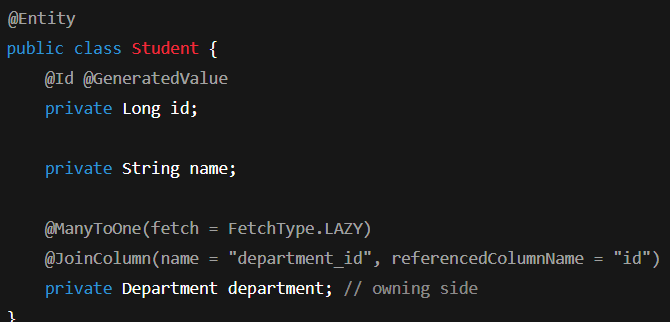
👉 **Definition**: Many entities reference one entity.

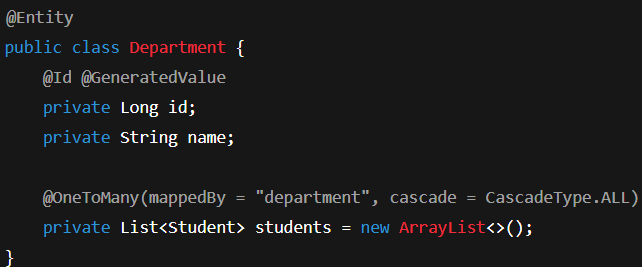
Example: **Student ↔ Department**

* Many students belong to one department.

**✅ Owning Side**

* **The "many" side owns the relationship** (since FK is stored here).
* Put @JoinColumn on **Student** (the many side).





* The **foreign key column** (department\_id) is always in the **many-side table**.
* Use mappedBy in the **one-side (Department)**.

1. To prevent recursion:
2. @JsonManagedReference on Department.students
3. @JsonBackReference on Student.department

**Many-to-Many (M:N)**

👉 **Definition**: Both sides can have multiple references to each other.

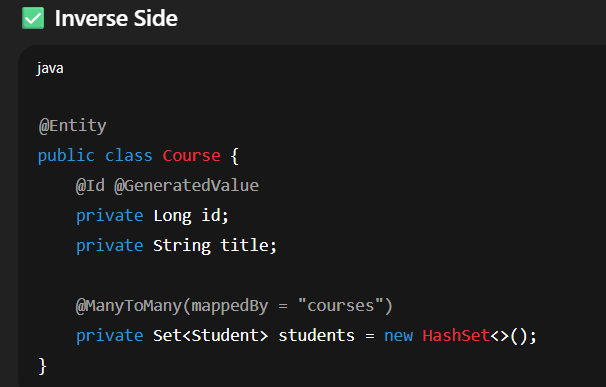
Example: **Student ↔ Course**

* A student can enroll in many courses
* A course can have many students

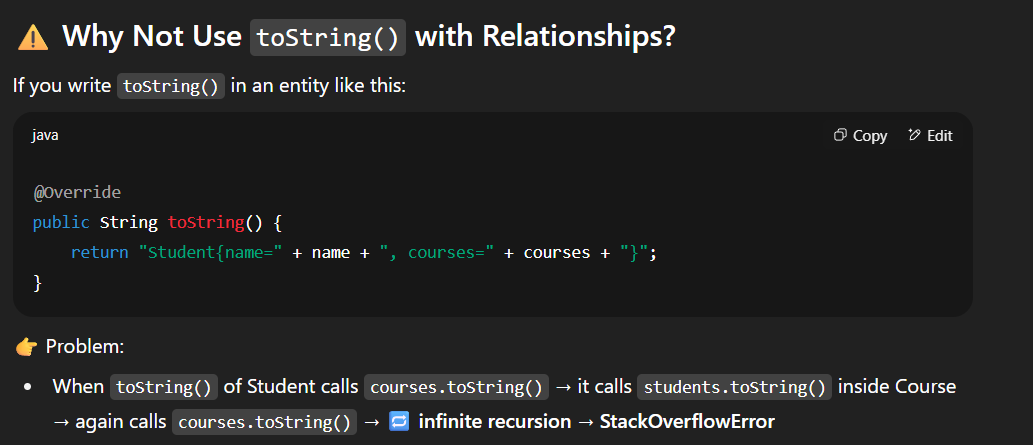
**✅ Owning Side**

* Pick one side (say Student) as owning side.
* Use @JoinTable to define the **join table** explicitly.





* **Use @JoinTable only on the owning side**.
* Inverse side uses mappedBy.
* To avoid infinite loop in JSON serialization:
* @JsonIgnore on one side, or
* @JsonManagedReference / @JsonBackReference



Never include relationships (@OneToMany, @ManyToMany) in toString()

**Attributes in Relationship Annotations**

* mappedBy = "fieldName" → defines inverse side (non-owning).
* cascade = CascadeType.ALL → operations cascade.
* fetch = FetchType.LAZY/EAGER → when data loads.
* @JoinColumn(name = "fk\_name", referencedColumnName = "pk\_name") → define FK.
* @JoinTable(name = "join\_table\_name", joinColumns = …, inverseJoinColumns = …) → define join table for many-to-many.

**CASCADE:**

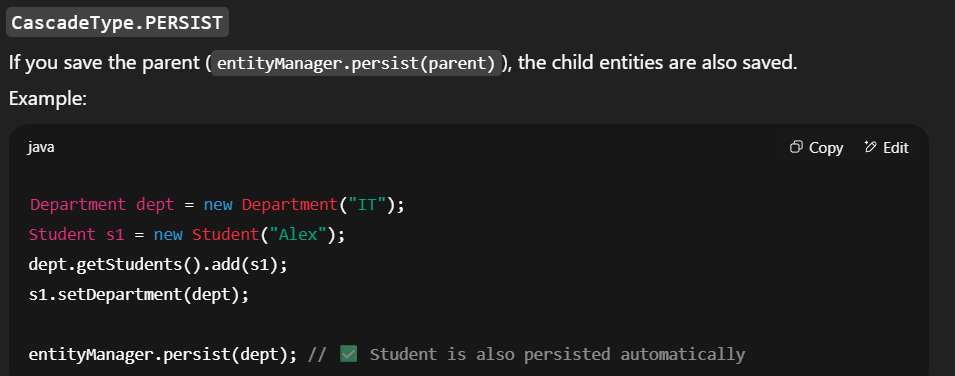
**Cascade Operations in JPA**

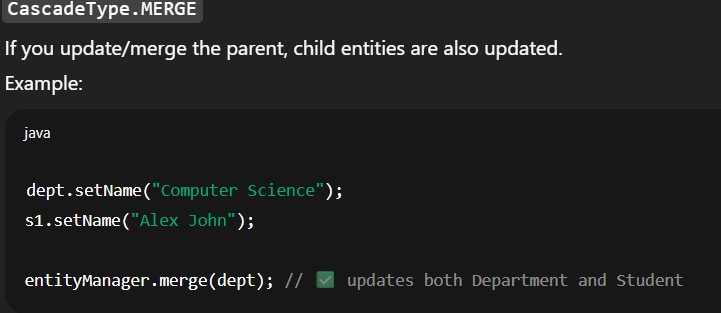
👉 **Definition**:  
cascade tells JPA what to do with related entities **when you perform an operation on the parent entity**.  
It’s like saying:

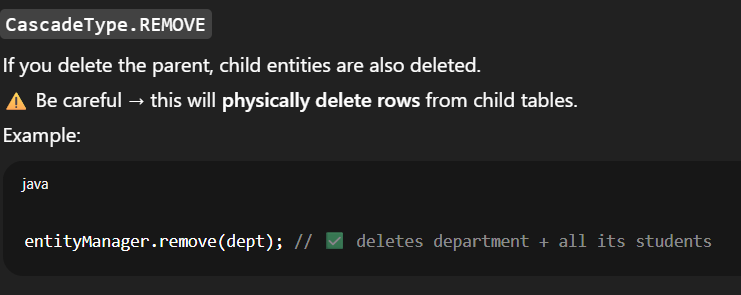
“If I save/delete/refresh the parent, should the same happen to the child too?”

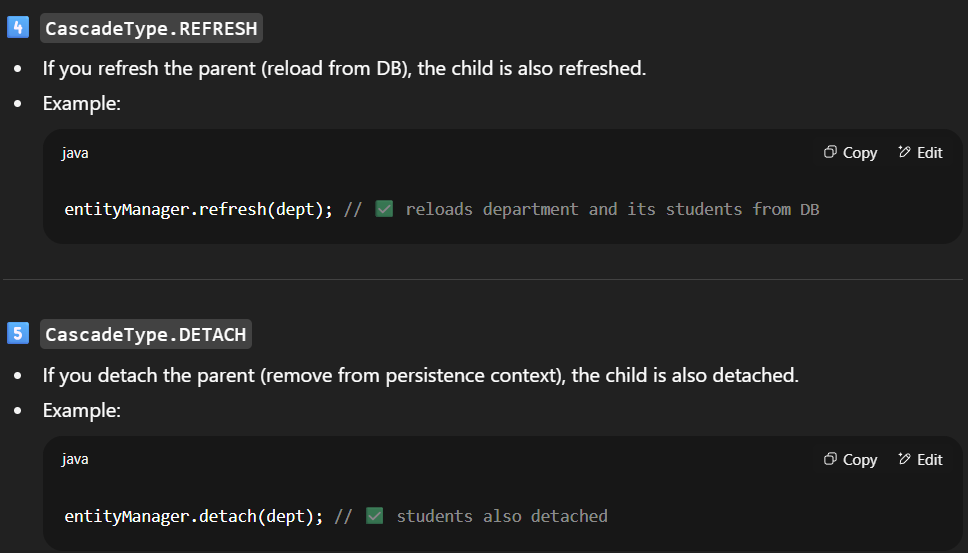
Types:

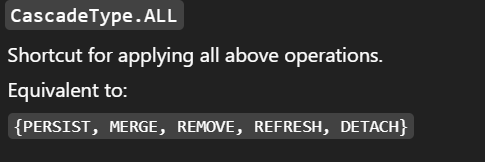
1. CascadeType.PERSIST
2. CascadeType.MERGE
3. CascadeType.REMOVE
4. Cascadetype.REFRESH
5. CascadeType.DETACH
6. CascadeType.ALL

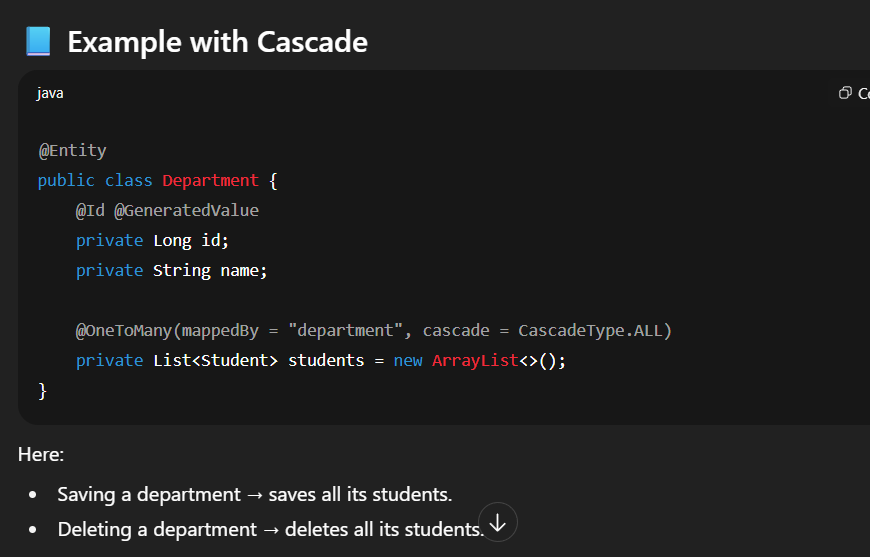












**Best Practices & Tips**

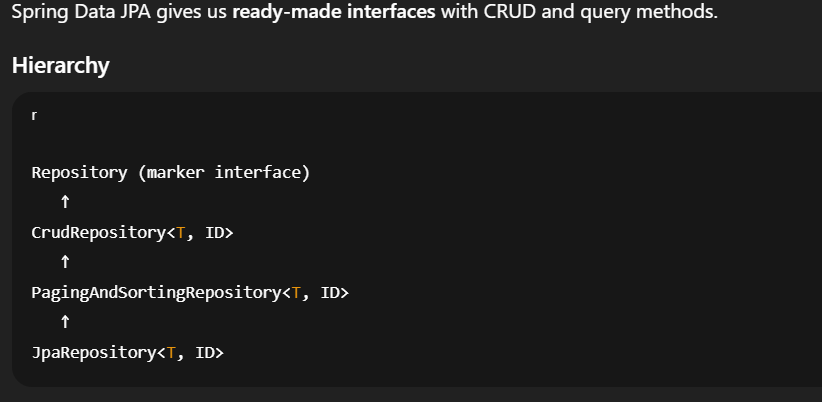
1. ✅ Use **CascadeType.ALL** carefully — mostly safe for **parent-child** relationships (e.g., Department → Students).
2. ❌ Avoid using REMOVE on relationships like **Student ↔ Course** (many-to-many). Deleting a Student shouldn’t delete a Course!
3. ✅ Use **PERSIST, MERGE** for shared entities like @ManyToOne (Student → Department).
4. ✅ For **Many-to-Many**, avoid cascades — handle explicitly.

**🎯 Analogy**

Think of **cascade** like a "family rule":

* If a parent goes shopping (persist), should kids go too? (persist child)
* If a parent moves house (merge), should kids move?
* If parent dies (remove), should kids also die? → ⚠️ Dangerous!
* If parent gets refreshed (refresh), kids also updated.

**JPA Repositories in Spring Data JPA**



**Repository**

* Just a marker interface. No methods.

✅ **CrudRepository<T, ID>**

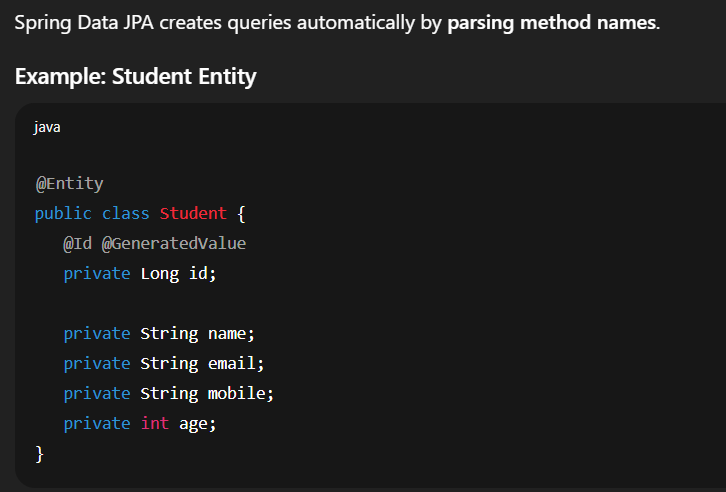
* Provides basic CRUD:
  + save(S entity)
  + findById(ID id)
  + findAll()
  + deleteById(ID id)

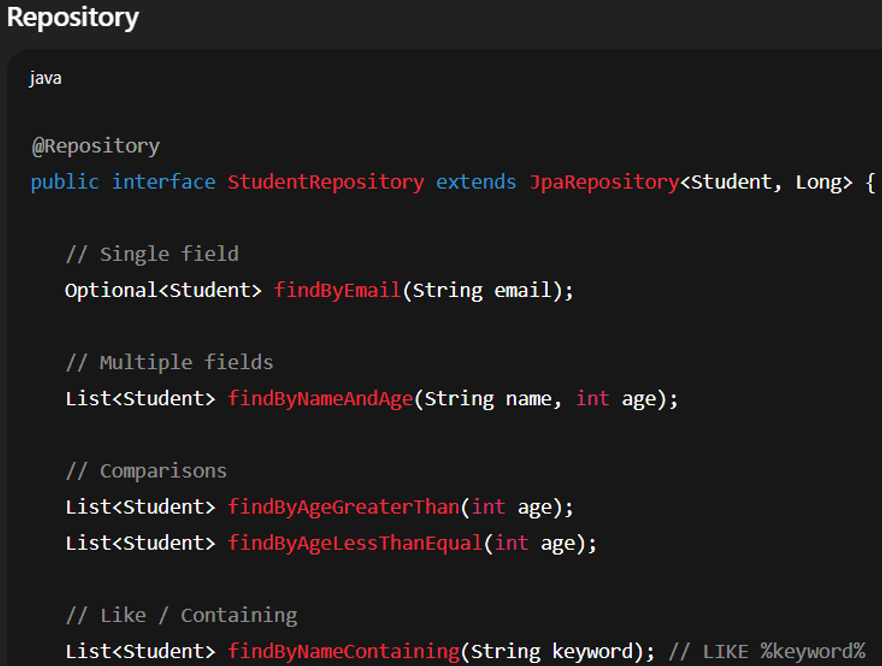
✅ **PagingAndSortingRepository<T, ID>**

* Adds pagination & sorting:
  + findAll(Pageable pageable)
  + findAll(Sort sort)

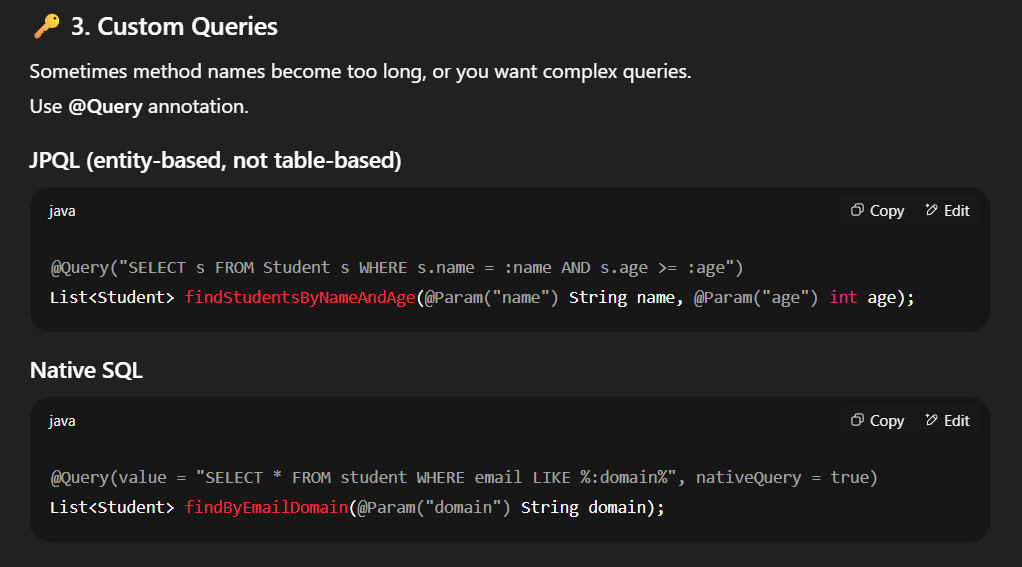
✅ **JpaRepository<T, ID>**

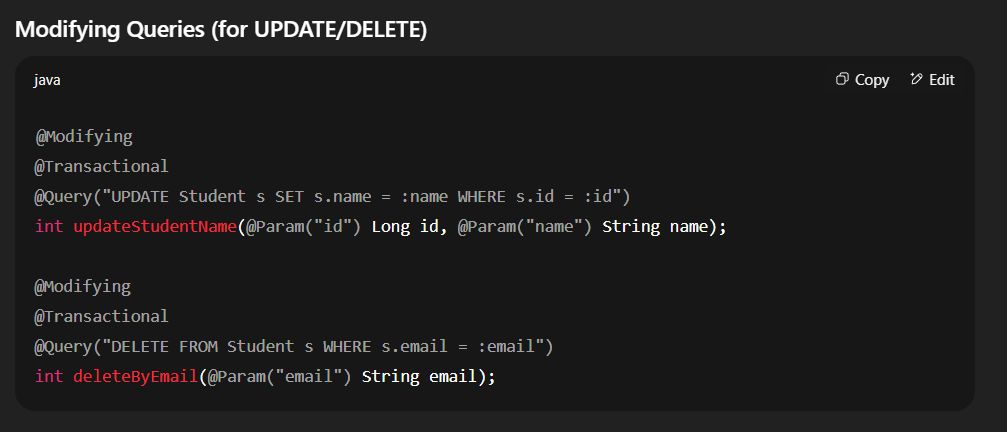
* Extends all above, adds **JPA-specific features**:
  + flush()
  + saveAndFlush(entity)
  + deleteInBatch(entities)
  + getOne(id) (lazy loading reference)
  + **Pagination + Sorting + Batch operations**

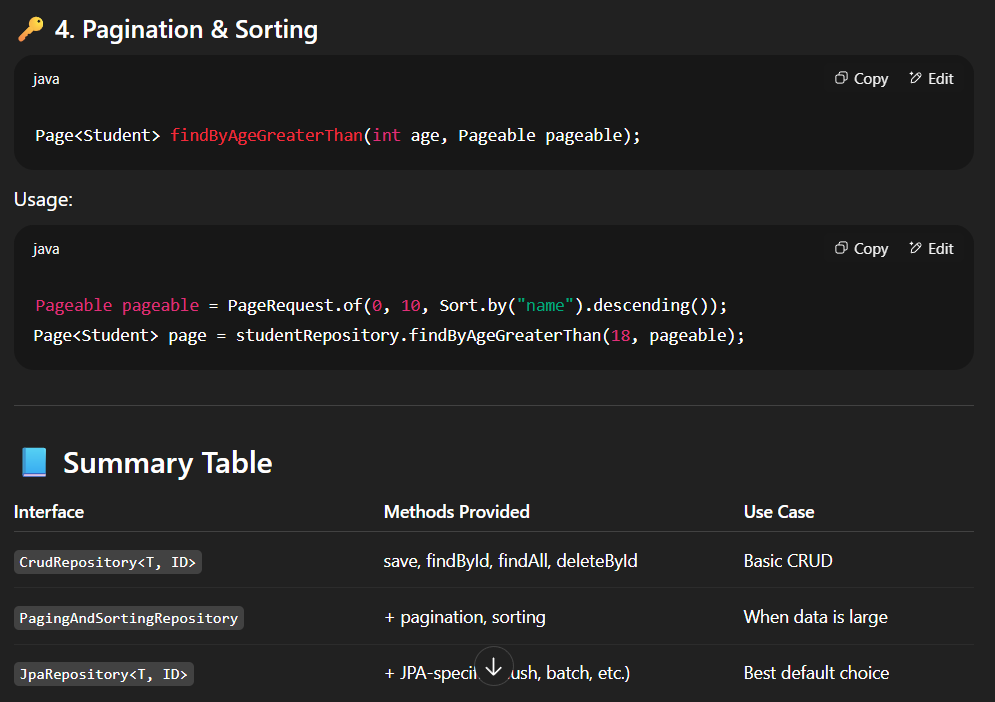




**@Query**







 **findBy** → tells Spring Data JPA this is a query method.  
Other valid prefixes: readBy, getBy, queryBy, searchBy.

 **NameAndAge** → corresponds to the **entity’s field names**.

* Name → matches the name field in Student.
* AndAge → matches the age field in Student.
* And is a connector (works like SQL AND).

