## Table Annotation:-

```
@Table(
    name = "employees",
    catalog = "employee_catalog",
    schema = "hr",
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"email"})
    },
    indexes = {
        @Index(name = "idx_name", columnList = "name"),
        @Index(name = "idx_department", columnList = "department")
    }
)
```

18/6/25

# Spring Data JPA & Dynamic Query Methods

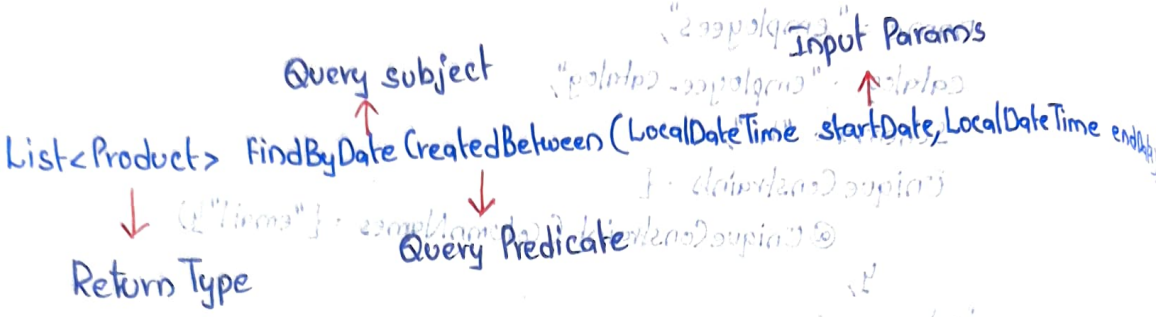
## Spring Data JPA:-

1. Spring Data JPA is a part of the larger Spring Data family.
2. It builds on top of JPA, Providing a higher-level and more convenient abstraction for data Access.
3. 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.

## Key features of Spring Data JPA:-

1. Repository Abstraction → Provides a Repository interface with methods for common data Access Operations.
2. Custom Query Methods → Allows defining custom query methods by simply declaring method names.
3. Pagination and Sorting → Offers built-in support for Pagination and Sorting.
4. Query Derivation → Automatically generates queries from method names.

# Rules for Creating Query Methods




## 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
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.

## SpringBoot JPA Testing Structure:-

```
Src
└─main
    └─java
        └─com.codingshuttle.jpaTutorial.jpaTuts
            ├─entities
            │   └─ProductEntity.java
            ├─repositories
            │   └─ProductRepository.java
            └─JpaTutorialApplication.java
test
└─java
    └─com.codingshuttle.jpaTutorial.jpaTuts
        └─JpaTutorialApplicationTests.java
```

```
@Test                          @Autowired
@Order(1)                      ProductRepository productRepository;
void inserProduct(){
    ProductEntity productEntity = ProductEntity.builder()
            .sku("nestle24")
            .title("Nestle Chocolate")
            .price(BigDecimal.valueOf(123.45))
            .quantity(12)
            .build();
    ProductRepository.save(ProductEntity);
```

Key points :-

* @SpringBootTest loads full Spring context for integration testing.
* Each @Test runs independently unless you control test execution order.
* @TestMethodOrder allows you to preserve DB state across tests.
* Avoid relying on data.sql during tests, instead insert Programmatically.
* Disable spring.sql.init.mode to Avoid SQL errors while running tests.
* Use Assertions.assertFalse() to add proper test validations.

→ In my SpringBoot JPA tests, I Prefer using full integration tests via @SpringBootTest.
→ To ensure stable DB state Across multiple tests, I use @TestMethodOrder so that insertions run before queries.
→ I insert test data Programmatically instead of Using data.sql to Avoid timing issues with Hibernate's table Creation.

## Spring Data JPA Repositories :-

→ Spring Data JPA allows me to focus on writing business logic. I simply extend JpaRepository<ProductEntity, Long> and Spring Boot Auto-generates full database CRUD functionally at runtime.

1. At the Top level is Repository interface.
2. CrudRepository gives basic Operations.
3. PagingAndSortingRepository adds Sorting/paging.
4. JpaRepository gives full JPA power - this is what we mostly use.
5. All of this works with Spring Boot auto-configuration - no SQL needed.

19/6/25

# Sorting & Pagination

## Sorting with method queries

### OrderBy

```
@Repository
Public interface EmployeeRepository extends JpaRepository < Employee Long>
{
   List < Employee > FindAllBy OrderByNameAsc();
   List < Employee > FindAllByOrder By Name Desc();
}
```

## Sorting with the Sort class

### Sort Parameter In Query Methods

```
@Repository
Public interface EmployeeRepository extends JpaRepository < Employee, Long > {
   List < Employee > FindByDepartment (String department , Sort sort);
}
```

### Using the Sort class

```
Sort sort = Sort.by (Sort.Direction.Asc, sortField);
Sort sort = Sort.by (Sort.Order.asc ("name"), Sort.Order.desc ("salary"));
```

## Key Concepts of Pagination:-

**Page :-** A single chunk of data that contains a subset of the Total dataset.
→ It is an interface representing a Page of data, including information about the Total number of Pages, total No. of elements, and the current Page's data.

**Pageable :-** An interface that Provides Pagination information such as Page number, Page size, and sorting options.

**PageRequest :-** A Concrete implementation of Pageable that Provides methods to create Pagination and sorting information.

# Using Pageable

```
Public interface UserRepository extends JpaRepository <User, Long>
    {
      Page<User> FindAll (Pageable pageable);
      Page<User> FindByLastName (String lastName, Pageable pageable);
    }
```

## Creating Pageable instance:-

```
Pageable pageable = PageRequest.of (PageNumber, size, Sort.by ("lastName").
                                                    ascending());
```

## Key points:-

→ Pagination is the Process of breaking down a large dataset into Smaller, manageable Pages, Instead of loading all the data in one go, we retrieve one Page at a time. This is Especially useful for handling large datasets in a memory-efficient way.

→ Sorting allows data to be ordered in Ascending (or) descending Order based on one (or) more fields.

→ Pageable is a interface used to create Pagination information Ex: Pagenumber, No. of records.

→ Sort is a interface helps to define the sorting logic, such as sorting by a field in Ascending (or) descending order.

→ we can Combine Pagination and sorting by passing both a Pageable object that contains sorting information.

20/6/25

# Spring Data Jpa Mappings Part-1

## Key Concepts of Mappings:-

### @Entity:-

1. it is a Java class that maps to a database table.
2. Every row in the database table represents an object of this class.
3. it helps in ORM - you work with objects, Hibernate handles DB.

### @Id:-

1. A Primary key uniquely identifies each record (row) in a table.

Foreign key:- → A field in One table that uniquely identifies a row in Another Table.

1. A Foreign key is a reference to a Primary key in Another Table.
2. it Connects two tables (relationships).
3. Maintains relationships blw tables.
4. In entity classes, it Comes via Annotations like @ManyToOne, @OneToMany, etc.

Ex:-

Employee table → ID=1

Address table → Employee_ID (Foreign key pointing to Employee ID).

### Cascade:-

1. it Means if you Perform Operation on parent entity, related child entities will be Automatically updated

Ex:-

→ if you delete an Employee, you may also want to Automatically delete all related Addresses → that's Cascading.

```
@OneToMany(Cascade = CascadeType.all)
Private List<Address> addresses;
```

Fetch Type:-

1. it defines when related entities are loaded from DB:
   * EAGER → load immediately.
   * LAZY → load only when Accessed.

Ex:- You load Employee, but may not always need full Address details immediately.

Default:-

@OneToMany → LAZY
@ManyToOne → EAGER

```
@OneToMany(fetch = FetchType.LAZY)
Private List<Address> addresses;
```

## Relationship Annotations:-

@OneToOne ⟶ One entity has exactly one related entity.

@OneToMany ⟶ One entity has many related entities.

@ManyToOne ⟶ Many entities relate to One entity.

@ManyToMany ⟶ Many-To-Many relationship

@JoinColumn

→ Specifies the foreign key Column name in the child table.

→ if Address table has column employee_id, you write

Ex:-
```
@ManyToOne
@JoinColumn(name = "employee_id")
Private Employee employee;
```

@JoinTable

→ it used in @ManyToMany relationships to define join (Mapping) table.

→ Student and Course both have many-to-many relation. You Create a join table Student_course with Student_id and Course_id.

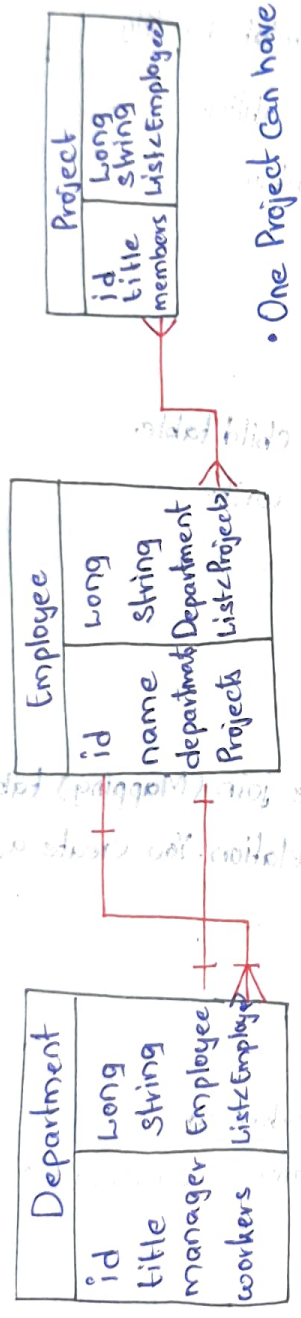
```
@ManyToMany
@JoinTable(
    name = "Student_Course",
    joinColumns = @JoinColumn(name = "student_id"),
    inverseJoinColumns = @JoinColumn(name = "course_id")
)
```

Explain ORM mappings in Spring Boot?

→ We Use @Entity to map Java class to database table.
→ @Id defines Primary key.
→ @OneToMany, @ManyToOne, @OneToOne, @ManyToMany defines relationships.
→ @JoinColumn & @JoinTable define foreign keys & join tables.
→ Cascade ⇒ Controls how Operations Propagate.
→ Fetch decides when related entities are loaded.

* The default Fetch type for @ManyToOne is EAGER.
* The default Fetch type for @OneToMany is LAZY.

25/6/25



@OneToOne

→ it Means: One entity object is related to exactly one other entity object.

→ This kind of relation is used when both sides have a Unique Connection To each other.

→ when we use @OneToOne, JPA/Hibernate will create a Foreign key in One of the tables to Connect both entities.

Ex:- ⇒ One Employee has one Department
⇒ One Department has one Employee.

Fields:-

1. target Entity
→ it tells Hibernate which class is being mapped. Mostly Optional because type is already clear from field declaration.
@OneToOne (targetEntity = Department.class)

2. Cascade
→ it allows automatic Propagation of Actions like save, update, delete from parent to related entity.
(cascade = CascadeType.ALL)

3. Fetch
→ it Controls the loading behavior. EAGER loads related entity immediately; LAZY loads only when accessed.
(fetch = FetchType.EAGER)

4. Optional
→ Defines whether this relationship is mandatory (or) Can be null.
True ⇒ related entity Can be null.
False ⇒ related entity must exist.
→ Optional false means - employee Cannot exist without department.
(optional = false)

5. mappedBy
→ Used on the inverse side To Say "I am not owning this relation, the other entity owns it".
→ Only one side should own the "relationship"; mappedBy tells which side is inverse
(mappedBy = "department")

6. OrphanRemoval
→ Automatically delete the child entity if it is removed from the relationship.
→ OrphanRemoval = true ensures if we remove reference to child entity, it gets Automatically deleted from DB.
(cascade = CascadeType.ALL, orphanRemoval = true)

26/6/25

# Spring Data Jpa Mapping Part-2 (OneToMany, ManyToMany)

* @OneToMany is used when One entity is related to Many others, like one Department having many Employees.
  → The mappedBy is used to point to the owning side (like Employee), and Cascade helps Propagate Operations.

1. Many objects relate to One object
   Ex:- Many Employees belong to One Department.
2. Employee class has a field pointing to Department.
3. A foreign key (dept_id) is added in Employee table.

* @ManyToOne is used when many entities are linked to One, like Many Employees belong to One Department.
  → The foreign key is maintained on this side Using @JoinColumn.

* @JoinColumn defines the foreign key Column. it's written on the owning side of the relationship, like in Employee for Department.

* @JoinTable is used to define a separate table for many-to-many relationship
  → it helps link both Tables using a thired Table with foreign keys of both entities.

* @Builder it helps Create Objects Step-by-step using a fluent API.
  → it Creates a flexible object builder. it Avoids long Constructors and improves Code readability by letting us set only required fields.

```
Employee emp = Employee.builder()
                .name("Nayeem")
                .age(25)
                .build();
```

→ Cascade is used to propagate actions from Parent to child.

→ if a Department has Employees, and we delete the Department with CascadeType.REMOVE, all associated Employees will also be deleted Automatically.

→ CascadeType.ALL is the Combination of all Operations like persist, merge, remove, etc.

1. No, it is Mandatory, it's Optional. Use it only when the child entity should be managed along with the parent.

* fetch controls when the related entity is loaded from the databases.
  → LAZY :- Data is fetched only when Accessed.
  → EAGER :- Data is fetched as Soon as Parent is loaded.

⇒ LAZY fetching to Optimize performance. if the related data like Department is not always required, I don't load it upfront.
  → But if I always need it, like in reports (or) dashboards, I might use EAGER.

Who is the Owner in @ManyToMany ?

1. In @ManyToMany, the side with @JoinTable is the Owner of the Relationship.

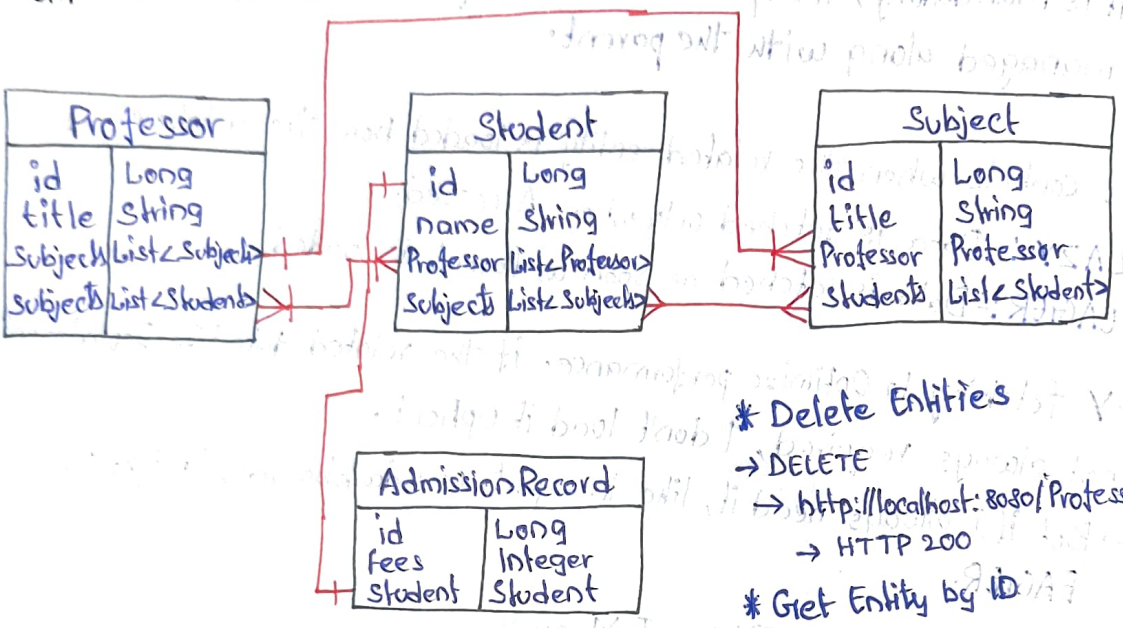
```
@ManyToMany
@JoinTable(name = "employee-project",
        joinColumns = @JoinColumn(name = "employee_id"),
        inverseJoinColumns = @JoinColumn(name = "Project_id"))
Private List<Project> projects;
```

→ Here, Employee is the Owning side because it has @JoinTable.

# Home work

1. Make a list of all the annotations we have seen so far in the Spring Data JPA.

2. Build this College management System in Spring Data JPA with all the mappings as well as APIs to assign the various mappings.




## Dependencies Used:-

1. Spring web.
2. Spring Data JPA
3. MySQL Driver
4. Lombok.

## APIS:-

### 1. Professors:-

→ POST
→ http://localhost:8080/Professors
→ {
"title": "Dr. Nayeem"
}

→ GET
→ http://localhost:8080/Professors

### 2. Student:-

→ POST
→ http://localhost:8080/students
→ {
"name": "Fayaz"
}

### 3. Subject

→ POST
→ http://localhost:8080/subjects
→ {
"title": "Java"
}

### 4. Create an Admission Record

→ POST
→ http://localhost:8080/admission-records
→ {
"Fees": 40000,
"student": {
"id": 1
}
}

### * Update Entities

→ PUT
→ http://localhost:8080/Professors/1
→ {
"title": "updated Professor Name"
}

### * Delete Entities

→ DELETE
→ http://localhost:8080/Professors/1
→ HTTP 200

### * Get Entity by ID

→ GET
→ http://localhost:8080/students/1

3. Create a Simple Spring Boot application to manage a library system. The System will have two main entities:-Book and Author. You will implement basic CRUD Operations, define relationships between entities, and Create Custom query methods. You need to define the Schema for this Assignment by Yourself

Implement endpoints To:-

* Create a new book and author.
* Retrieve a list of all books and authors.
* Retrieve a single book (or) Author by ID.
* Update book and author details.
* Delete a book (or) Author.
* Find books by title.
* Find books Published after a Certain data.
* Find authors by name.
* Find all books by a specific author.