**Spring framework**

# **3. ORM**

# **1. ORM là gì ? Sử dụng ORM mang lại lợi ích như thế nào cho ứng dụng. Cơ chế hoạt động của ORM như thế nào? So sánh performance của việc sử dụng ORM vs JDBC**

**ORM (Object-Relational Mapping)** là một kỹ thuật trong lập trình để ánh xạ (mapping) các đối tượng trong ngôn ngữ lập trình hướng đối tượng (OOP) với các bảng trong cơ sở dữ liệu quan hệ (RDBMS). ORM cho phép lập trình viên làm việc với cơ sở dữ liệu sử dụng các đối tượng thay vì viết trực tiếp các câu lệnh SQL, giúp giảm thiểu sự phức tạp khi thao tác với dữ liệu trong ứng dụng.

### **Lợi ích của việc sử dụng ORM:**

1. **Tăng tính trừu tượng và dễ bảo trì**:
   * ORM cung cấp một lớp trừu tượng giữa ứng dụng và cơ sở dữ liệu, giúp lập trình viên thao tác với dữ liệu như là các đối tượng trong OOP mà không cần quan tâm quá nhiều đến chi tiết của SQL hoặc cơ sở dữ liệu.
2. **Giảm thiểu lỗi SQL và bảo mật**:
   * ORM tự động tạo ra các câu lệnh SQL tương ứng từ các đối tượng, giúp giảm lỗi phát sinh từ việc viết SQL thủ công.
   * Nó cũng giảm thiểu nguy cơ **SQL injection** bằng cách sử dụng các cơ chế query an toàn.
3. **Tăng năng suất phát triển**:
   * Với ORM, lập trình viên có thể tiết kiệm thời gian viết và tối ưu các câu lệnh SQL bằng cách thao tác trực tiếp trên các đối tượng.
4. **Tính di động**:
   * Các framework ORM thường hỗ trợ nhiều loại cơ sở dữ liệu khác nhau, cho phép ứng dụng dễ dàng thay đổi cơ sở dữ liệu mà không phải thay đổi nhiều code liên quan đến việc truy vấn dữ liệu.

### **Cơ chế hoạt động của ORM:**

ORM hoạt động bằng cách ánh xạ các bảng trong cơ sở dữ liệu thành các lớp (class) trong ngôn ngữ lập trình, và các bản ghi (record) trong bảng thành các đối tượng (object) của lớp tương ứng. Các thuộc tính của lớp sẽ ánh xạ với các cột trong bảng. Khi lập trình viên thao tác với các đối tượng (ví dụ như thêm, sửa, xóa), ORM sẽ tự động dịch các thao tác này thành các câu lệnh SQL tương ứng và thực hiện chúng trên cơ sở dữ liệu.

Các framework ORM phổ biến như Hibernate (Java), Entity Framework (C#), và Sequelize (JavaScript) thường hỗ trợ:

* **CRUD operations (Create, Read, Update, Delete)**: Thao tác với dữ liệu một cách dễ dàng.
* **Lazy loading và Eager loading**: Điều khiển cách dữ liệu được tải từ cơ sở dữ liệu.
* **Caching**: Tối ưu hóa hiệu suất truy vấn bằng cách lưu tạm thời các đối tượng trong bộ nhớ.

### **So sánh performance giữa ORM và JDBC (Java Database Connectivity):**

* **ORM**:
  + **Ưu điểm**:
    - Giảm công sức phát triển và bảo trì code, dễ đọc và hiểu.
    - Có các tính năng như caching, lazy loading giúp tối ưu hóa việc truy vấn dữ liệu.
  + **Nhược điểm**:
    - Có thể chậm hơn JDBC vì nó thêm một lớp trừu tượng giữa ứng dụng và cơ sở dữ liệu.
    - Khi làm việc với các hệ thống lớn hoặc cần hiệu suất cao, ORM có thể gặp phải vấn đề "n+1 query", tức là thay vì chỉ một câu lệnh SQL, nhiều câu lệnh không cần thiết có thể được phát sinh.
* **JDBC**:
  + **Ưu điểm**:
    - Cho phép viết câu lệnh SQL thủ công, tối ưu hóa truy vấn cho từng trường hợp cụ thể, do đó có thể có hiệu suất tốt hơn.
    - Không có overhead của ORM, nên với các ứng dụng yêu cầu hiệu suất cao, JDBC có thể là sự lựa chọn tốt hơn.
  + **Nhược điểm**:
    - Việc viết SQL và quản lý kết nối cơ sở dữ liệu thủ công yêu cầu nhiều thời gian hơn và dễ gặp lỗi hơn.
    - Khó bảo trì, đặc biệt khi ứng dụng phát triển và trở nên phức tạp hơn.

# **2. Spring JPA có phải là 1 triển khai của ORM hay không**

**Spring Data JPA** là một phần của **Spring Framework** và nó không phải là một triển khai của ORM trực tiếp, mà là một **abstraction layer** (lớp trừu tượng) được xây dựng trên các framework ORM như **Hibernate**, giúp đơn giản hóa việc tương tác với cơ sở dữ liệu.

Spring Data JPA giúp lập trình viên dễ dàng sử dụng các thao tác truy vấn và quản lý thực thể (entity) mà không cần phải viết quá nhiều code. Nó cung cấp một cách tiếp cận dựa trên giao diện (interface-based programming) để làm việc với cơ sở dữ liệu thông qua JPA (Java Persistence API).

### **Các điểm chính về Spring Data JPA:**

1. **Spring Data JPA** hoạt động dựa trên JPA (Java Persistence API), mà JPA chính là một chuẩn cho ORM (Object-Relational Mapping). Tuy nhiên, JPA chỉ định nghĩa các đặc tả (specification), và cần có một **ORM implementation** cụ thể để hoạt động (ví dụ như Hibernate).
2. **Hibernate** là một trong những ORM phổ biến nhất và là **ORM implementation** của JPA. Khi bạn sử dụng Spring Data JPA, Hibernate thường được sử dụng ngầm như là **ORM provider** để xử lý việc ánh xạ giữa các đối tượng và bảng cơ sở dữ liệu.
3. **Spring Data JPA** mở rộng JPA bằng cách cung cấp thêm nhiều tính năng hữu ích, như:
   * **Repository abstraction**: Giúp tạo ra các phương thức truy vấn (query methods) mà không cần phải viết SQL.
   * **Query creation**: Bạn có thể định nghĩa các phương thức truy vấn bằng cách sử dụng cú pháp rõ ràng từ tên phương thức (method names), ví dụ như findByLastName(String lastName).
   * **Custom queries**: Nếu cần, bạn vẫn có thể viết các câu truy vấn SQL hoặc JPQL thủ công.
   * **Pagination và Sorting**: Spring Data JPA cung cấp cơ chế phân trang và sắp xếp một cách dễ dàng.

# **3. Tìm hiểu các loại quan hệ trong JPA : Many To One, One To Many, Many To Many**

# Trong JPA (Java Persistence API), các mối quan hệ giữa các thực thể (entity) phản ánh cách các bảng trong cơ sở dữ liệu quan hệ có thể tương tác với nhau. Các loại quan hệ phổ biến trong JPA bao gồm:

### **1. Many-to-One (Nhiều - Một)**

# Đây là mối quan hệ phổ biến nhất trong JPA, được sử dụng khi nhiều thực thể có thể liên kết với một thực thể khác. Ví dụ, nhiều sinh viên có thể thuộc về một lớp học.

# **Chi tiết:**

# Trong mối quan hệ này, @ManyToOne được sử dụng trên thuộc tính mà thể hiện liên kết với thực thể khác.

# @JoinColumn xác định khóa ngoại trong bảng Student, trỏ tới bảng ClassRoom.

### **2. One-to-Many (Một - Nhiều)**

# Mối quan hệ này ngược lại với Many-to-One: một thực thể có thể liên kết với nhiều thực thể khác. Ví dụ, một ClassRoom có nhiều Student.

# **Chi tiết:**

# @OneToMany được sử dụng trên thực thể có mối quan hệ "một" và thuộc tính chứa danh sách các đối tượng liên quan.

# mappedBy chỉ ra tên của trường ánh xạ trong thực thể "nhiều", tức là classRoom trong thực thể Student.

# JPA sử dụng mappedBy để biết mối quan hệ đã được định nghĩa bởi thực thể "nhiều" (trong ví dụ này là Student), vì vậy nó không cần phải tạo thêm bảng trung gian hoặc cột khóa ngoại ở bên "một".

# Trong JPA (Java Persistence API), các mối quan hệ giữa các thực thể (entity) phản ánh cách các bảng trong cơ sở dữ liệu quan hệ có thể tương tác với nhau. Các loại quan hệ phổ biến trong JPA bao gồm:

### **1. Many-to-One (Nhiều - Một)**

# Đây là mối quan hệ phổ biến nhất trong JPA, được sử dụng khi nhiều thực thể có thể liên kết với một thực thể khác. Ví dụ, nhiều sinh viên có thể thuộc về một lớp học.

# **Chi tiết:**

# Trong mối quan hệ này, @ManyToOne được sử dụng trên thuộc tính mà thể hiện liên kết với thực thể khác.

# @JoinColumn xác định khóa ngoại trong bảng Student, trỏ tới bảng ClassRoom.

### **2. One-to-Many (Một - Nhiều)**

# Mối quan hệ này ngược lại với Many-to-One: một thực thể có thể liên kết với nhiều thực thể khác. Ví dụ, một ClassRoom có nhiều Student.

# **Chi tiết:**

# @OneToMany được sử dụng trên thực thể có mối quan hệ "một" và thuộc tính chứa danh sách các đối tượng liên quan.

# mappedBy chỉ ra tên của trường ánh xạ trong thực thể "nhiều", tức là classRoom trong thực thể Student.

# JPA sử dụng mappedBy để biết mối quan hệ đã được định nghĩa bởi thực thể "nhiều" (trong ví dụ này là Student), vì vậy nó không cần phải tạo thêm bảng trung gian hoặc cột khóa ngoại ở bên "một".

### **3. Many-to-Many (Nhiều - Nhiều)**

# Mối quan hệ này thể hiện rằng một thực thể có thể liên kết với nhiều thực thể khác và ngược lại. Ví dụ, nhiều học sinh có thể tham gia nhiều khóa học, và mỗi khóa học có nhiều học sinh.

# **Chi tiết:**

# @ManyToMany được sử dụng ở cả hai thực thể để biểu diễn mối quan hệ nhiều-nhiều.

# @JoinTable chỉ định tên bảng trung gian và các khóa ngoại để ánh xạ giữa hai bảng.

# Trong bảng trung gian student\_course, mỗi dòng sẽ chứa một student\_id và một course\_id, tạo ra sự kết nối giữa hai thực thể.

### **Tóm tắt:**

# **Many-to-One**: Một khóa ngoại trên bảng "nhiều" trỏ tới bảng "một". Ví dụ, nhiều sinh viên thuộc về một lớp.

# **One-to-Many**: Một thực thể "một" chứa một danh sách các thực thể "nhiều". Ví dụ, một lớp có nhiều sinh viên.

# **Many-to-Many**: Cả hai thực thể đều có danh sách chứa các thực thể kia, sử dụng một bảng trung gian để lưu trữ quan hệ. Ví dụ, nhiều sinh viên học nhiều khóa học, và mỗi khóa học có nhiều sinh viên.

# **4. Tìm hiểu các loại quan hệ trong JPA : Many To One, One To Many, Many To Many**

### **1. Many-to-One (Nhiều - Một)**

Đây là mối quan hệ phổ biến nhất trong JPA, được sử dụng khi nhiều thực thể có thể liên kết với một thực thể khác. Ví dụ, nhiều sinh viên có thể thuộc về một lớp học.

**Chi tiết:**

* Trong mối quan hệ này, @ManyToOne được sử dụng trên thuộc tính mà thể hiện liên kết với thực thể khác.
* @JoinColumn xác định khóa ngoại trong bảng Student, trỏ tới bảng ClassRoom.

### **2. One-to-Many (Một - Nhiều)**

Mối quan hệ này ngược lại với Many-to-One: một thực thể có thể liên kết với nhiều thực thể khác. Ví dụ, một ClassRoom có nhiều Student.

* **Chi tiết:**
  + @OneToMany được sử dụng trên thực thể có mối quan hệ "một" và thuộc tính chứa danh sách các đối tượng liên quan.
  + mappedBy chỉ ra tên của trường ánh xạ trong thực thể "nhiều", tức là classRoom trong thực thể Student.
  + JPA sử dụng mappedBy để biết mối quan hệ đã được định nghĩa bởi thực thể "nhiều" (trong ví dụ này là Student), vì vậy nó không cần phải tạo thêm bảng trung gian hoặc cột khóa ngoại ở bên "một".

### **3. Many-to-Many (Nhiều - Nhiều)**

Mối quan hệ này thể hiện rằng một thực thể có thể liên kết với nhiều thực thể khác và ngược lại. Ví dụ, nhiều học sinh có thể tham gia nhiều khóa học, và mỗi khóa học có nhiều học sinh.

**Chi tiết:**

* @ManyToMany được sử dụng ở cả hai thực thể để biểu diễn mối quan hệ nhiều-nhiều.
* @JoinTable chỉ định tên bảng trung gian và các khóa ngoại để ánh xạ giữa hai bảng.
* Trong bảng trung gian student\_course, mỗi dòng sẽ chứa một student\_id và một course\_id, tạo ra sự kết nối giữa hai thực thể.

# **5. Các loại Cascade type, Fetch type trong JPA**

### **1. Cascade Type (Loại Cascade)**

Cascade type trong JPA xác định hành vi của các thao tác (thêm, sửa, xóa, v.v.) trên một thực thể sẽ ảnh hưởng như thế nào đến các thực thể liên quan của nó. Nói cách khác, khi một thực thể bị tác động (ví dụ như bị xóa hoặc lưu), JPA sẽ tự động áp dụng thao tác tương tự lên các thực thể có mối quan hệ với nó.

Có các loại **Cascade type** sau:

* **CascadeType.PERSIST**: Khi lưu thực thể chính (Entity A), nếu thực thể liên quan (Entity B) chưa được lưu, nó cũng sẽ tự động được lưu vào cơ sở dữ liệu.
* **CascadeType.MERGE**: Khi thực thể chính được cập nhật, JPA sẽ tự động cập nhật các thực thể liên quan.
* **CascadeType.REMOVE**: Khi thực thể chính bị xóa, các thực thể liên quan cũng sẽ bị xóa theo.
* **CascadeType.REFRESH**: Khi thực thể chính được làm mới từ cơ sở dữ liệu, các thực thể liên quan cũng sẽ được làm mới.
* **CascadeType.DETACH**: Khi thực thể chính bị tách (detach) ra khỏi context của JPA, các thực thể liên quan cũng sẽ bị tách theo.
* **CascadeType.ALL**: Áp dụng tất cả các hành vi cascade đã liệt kê (PERSIST, MERGE, REMOVE, REFRESH, DETACH).

### **2. Fetch Type (Loại Fetch)**

**Fetch type** quy định cách dữ liệu liên quan sẽ được tải từ cơ sở dữ liệu, cụ thể là khi nào các thực thể liên quan được nạp vào bộ nhớ khi bạn truy vấn một thực thể chính.

Có hai loại **Fetch type** chính:

* **FetchType.EAGER**: **Tải ngay**.
  + Khi truy vấn thực thể chính, các thực thể liên quan cũng sẽ được nạp ngay lập tức vào bộ nhớ.
* **Ưu điểm**: Dễ sử dụng khi bạn biết rằng bạn luôn cần dữ liệu liên quan ngay lập tức.
* **Nhược điểm**: Có thể gây ra truy vấn nặng và tốn nhiều tài nguyên khi không cần thiết, đặc biệt nếu có nhiều quan hệ.

**FetchType.LAZY**: **Tải chậm**.

* Thực thể liên quan chỉ được tải khi bạn thực sự truy cập đến nó. Khi thực thể chính được truy vấn, các thực thể liên quan không được nạp vào ngay lập tức.
* **Ưu điểm**: Tiết kiệm tài nguyên vì chỉ tải dữ liệu khi thực sự cần thiết.
* **Nhược điểm**: Có thể dẫn đến vấn đề **LazyInitializationException** nếu bạn cố truy cập dữ liệu liên quan sau khi session của JPA đã đóng.

**CascadeType.ALL**: Nếu Author được lưu, cập nhật hoặc xóa, các thực thể Book liên quan cũng sẽ trải qua cùng hành động đó.

**FetchType.LAZY**: Danh sách các Book sẽ không được tải ngay lập tức khi bạn lấy Author. Chúng sẽ chỉ được tải khi bạn gọi getBooks().

# **5. Tìm hiểu cách convert DTO -> Entity và ngược lại sử dụng 2 cách : dùng thư viện và sử dụng java reflection**

Chúng ta có thể thực hiện việc chuyển đổi này bằng nhiều cách, bao gồm:

1. **Sử dụng thư viện** (thường là thư viện mapping như MapStruct hoặc ModelMapper).
2. **Sử dụng Java Reflection** để tự động ánh xạ các thuộc tính giữa DTO và Entity.

##### **Ưu điểm:**

* Dễ cấu hình và sử dụng.
* Hỗ trợ mapping tự động các thuộc tính có tên giống nhau.
* Hỗ trợ cấu hình mapping tùy chỉnh với các thuộc tính khác tên.

##### **Nhược điểm:**

* Có thể làm ứng dụng nặng nề với những dự án lớn hoặc những mapping phức tạp.
* Hiệu suất không nhanh bằng MapStruct.

#### **Sử dụng MapStruct:**

**MapStruct** là một thư viện compile-time, tức là việc ánh xạ giữa các đối tượng sẽ được thực hiện tại thời gian biên dịch (compile-time), nên nó nhanh hơn so với các thư viện runtime như ModelMapper.

### **1. Sử dụng thư viện (ModelMapper/MapStruct)**

**ModelMapper** và **MapStruct** là hai thư viện phổ biến hỗ trợ việc chuyển đổi giữa DTO và Entity một cách tự động và nhanh chóng.

##### **Ưu điểm:**

* Hiệu suất tốt hơn ModelMapper vì mapping được tạo ra tại thời gian biên dịch (compile-time).
* An toàn hơn vì các lỗi sẽ được phát hiện ngay tại thời gian biên dịch.

##### **Nhược điểm:**

* Phải tạo thêm các interface mapper, có thể phức tạp hơn khi cần mapping tùy chỉnh.

### **2. Sử dụng Java Reflection**

Java Reflection cho phép bạn truy cập và thao tác với các thuộc tính, phương thức, và constructor của một class tại runtime. Chúng ta có thể sử dụng Reflection để tự động ánh xạ các thuộc tính giữa DTO và Entity.

##### **Ưu điểm:**

* Không cần phụ thuộc vào thư viện bên ngoài.
* Linh hoạt, có thể ánh xạ các thuộc tính động.

##### **Nhược điểm:**

* Hiệu suất thấp hơn so với việc sử dụng các thư viện tối ưu hóa sẵn có như MapStruct hoặc ModelMapper, vì reflection tiêu tốn nhiều tài nguyên tại runtime.
* Code phức tạp hơn và dễ gặp lỗi hơn.
* Không an toàn bằng compile-time mapping (như với MapStruct).

# **5. Native query**

Trong Java, khi sử dụng JPA (Java Persistence API), ngoài việc sử dụng HQL (Hibernate Query Language) hoặc JPQL (Java Persistence Query Language) để thao tác với cơ sở dữ liệu, bạn có thể sử dụng **native query**. Native query là các truy vấn SQL thuần túy được thực thi trực tiếp trên cơ sở dữ liệu mà không cần thông qua các lớp thực thể.

### **Native Query là gì?**

**Native query** là các truy vấn SQL thông thường mà bạn viết trực tiếp trong mã Java. Sử dụng native query cho phép bạn tận dụng các tính năng đặc biệt của hệ quản trị cơ sở dữ liệu mà không được hỗ trợ đầy đủ bởi HQL hoặc JPQL.

### **Khi nào nên dùng Native Query?**

* Khi bạn cần tận dụng các tính năng đặc biệt của hệ quản trị cơ sở dữ liệu mà JPQL không hỗ trợ.
* Khi bạn cần viết các truy vấn SQL phức tạp, bao gồm các phép nối (JOIN), truy vấn con (sub-query), hay các phép toán SQL nâng cao.
* Khi cần tối ưu hóa hiệu suất hoặc thao tác với bảng không ánh xạ trực tiếp với các entity.

### **Cách sử dụng Native Query trong JPA**

#### **1. Sử dụng EntityManager**

Trong JPA, bạn có thể sử dụng EntityManager để thực thi native query. Cách này thích hợp khi bạn không sử dụng Spring Data JPA.

#### **2. Sử dụng @Query với Native Query trong Spring Data JPA**

Nếu bạn đang sử dụng **Spring Data JPA**, bạn có thể dễ dàng viết native query bằng cách sử dụng annotation @Query.

#### **3. Truyền tham số vào Native Query**

Bạn có thể truyền tham số vào native query bằng cách sử dụng placeholder (?) hoặc tên tham số (:paramName).

#### **4. Sử dụng Native Query để cập nhật dữ liệu (Update/Delete)**

Native query cũng có thể được sử dụng để thực hiện các thao tác cập nhật hoặc xóa dữ liệu.

@Modifying được sử dụng để chỉ định rằng đây là một thao tác thay đổi dữ liệu (update hoặc delete).

Phương thức trả về một số nguyên cho biết có bao nhiêu hàng bị ảnh hưởng bởi câu lệnh update.

#### **5. Mapping kết quả phức tạp với Native Query**

Nếu kết quả của truy vấn SQL không thể ánh xạ trực tiếp đến một entity, bạn có thể ánh xạ nó bằng một DTO hoặc một loại không liên quan đến entity.

Ở đây, kết quả của native query được trả về dưới dạng một danh sách Object[], trong đó mỗi phần tử là một hàng dữ liệu.

Sau đó, kết quả được ánh xạ lại vào UserDTO.

### **Lưu ý khi sử dụng Native Query**

1. **Tính phụ thuộc vào cơ sở dữ liệu**: Native query phụ thuộc trực tiếp vào cú pháp SQL của hệ quản trị cơ sở dữ liệu mà bạn đang sử dụng. Điều này làm giảm tính di động của ứng dụng nếu bạn thay đổi cơ sở dữ liệu.
2. **Tính bảo trì**: Native query có thể khó bảo trì hơn vì bạn phải tự xử lý các truy vấn SQL và phải đảm bảo chúng tương thích với cấu trúc cơ sở dữ liệu.
3. **Hiệu suất**: Trong một số trường hợp, native query có thể có hiệu suất tốt hơn JPQL hoặc HQL, vì bạn có toàn quyền kiểm soát cách truy vấn được viết và tối ưu hóa.

# **6. Specification, Paging**

### **1. Specification trong Spring Data JPA**

**Specification** là một API của Spring Data JPA, dựa trên mô hình **Criteria API** của JPA. Nó cho phép xây dựng các truy vấn phức tạp một cách động mà không cần viết SQL thuần. Đây là cách hữu ích khi bạn cần tạo các truy vấn có điều kiện linh hoạt dựa trên các tham số đầu vào.

##### **Bước 2: Xây dựng Specification**

Bạn sẽ tạo một lớp Specification với các điều kiện truy vấn mong muốn.

#### **Lợi ích của Specification:**

* Giúp xây dựng các truy vấn phức tạp một cách linh hoạt và động.
* Có thể kết hợp nhiều điều kiện và phân nhánh truy vấn (dùng and, or).
* Tăng tính bảo trì và khả năng mở rộng của code.

### **2. Paging (Phân trang) trong Spring Data JPA**

**Paging** cho phép bạn phân trang kết quả truy vấn từ cơ sở dữ liệu, giúp giảm tải khi làm việc với dữ liệu lớn. Spring Data JPA cung cấp một API mạnh mẽ để phân trang và sắp xếp các kết quả.

#### **Cách sử dụng Paging**

##### **Bước 1: Sử dụng Pageable**

Spring Data JPA cung cấp đối tượng Pageable để xác định kích thước trang và số thứ tự trang bạn muốn truy vấn.

* PageRequest.of(page, size) sẽ tạo ra một Pageable với số trang và kích thước trang được chỉ định.
* Phương thức findAll(pageable) sẽ trả về một đối tượng Page chứa danh sách kết quả phân trang.

##### **Bước 2: Kết hợp Paging với Specification**

Bạn có thể kết hợp **Specification** và **Paging** để thực hiện phân trang với các điều kiện truy vấn động.

##### **Sắp xếp (Sorting):**

Spring Data JPA cũng hỗ trợ sắp xếp dữ liệu khi phân trang. Bạn có thể truyền thêm Sort vào PageRequest để sắp xếp dữ liệu.

* Ở đây, dữ liệu sẽ được sắp xếp theo trường name theo thứ tự tăng dần (ascending). Bạn cũng có thể sử dụng descending() để sắp xếp theo thứ tự giảm dần.

#### **Lợi ích của Paging:**

* Giảm tải cho hệ thống khi xử lý dữ liệu lớn bằng cách chỉ truy xuất một phần dữ liệu.
* Cung cấp các thông tin liên quan đến phân trang như tổng số trang, tổng số bản ghi, kích thước trang, v.v.
* Kết hợp tốt với Specification để tạo các truy vấn phức tạp và phân trang linh hoạt.

# **7. Lấy dữ liệu khi JOIN nhiều bảng sử dụng JPA (mỗi bảng lấy 1 tới 2 trường thông tin)**