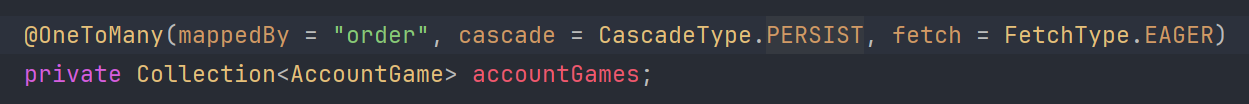
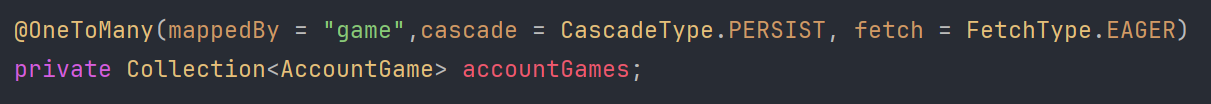
Một lỗi của CascadeType ảnh hưởng đến 3 bảng dưới

Ảnh có chứa văn bản, Phông chữ, số, biểu đồ

Mô tả được tạo tự động



Nếu orders quan hệ account\_game 1 nhìu và sử dụng CascadeType.All thay vì CascadeType.PERSIST



Tương tự nếu game quan hệ account\_game 1 nhìu và sử dụng CascadeType.All thay vì CascadeType.PERSIST

Thì hypernate sẽ hiểu lầm rằng account\_game là bảng trung gian và nối bảng orders với game thành 1 nhìu luôn gây dư thừa dữ liệu.

**NÊN SỬ DỤNG PERSIST THAY VÌ ALL**

**Ngoài ra**

**# neu muon thay schema nay bằng 1 schema moi thi phai doi update thanh create**

**spring.jpa.hibernate.ddl-auto = update**

| **Cascade Type** | **Hành vi** | **Khi nào nên sử dụng** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **PERSIST** | **Lưu tự động các thực thể con chưa tồn tại** | **Khi thực thể con phụ thuộc vào thực thể cha** |

|  |  |  |
| --- | --- | --- |
| **MERGE** | **Cập nhật tự động các thực thể con** | **Khi muốn đồng bộ các thay đổi của cha và con** |

|  |  |  |
| --- | --- | --- |
| **REMOVE** | **Xóa tự động các thực thể con** | **Khi thực thể con không có ý nghĩa độc lập** |

|  |  |  |
| --- | --- | --- |
| **REFRESH** | **Làm mới tự động các thực thể con** | **Khi cần đảm bảo trạng thái của cha và con khớp với cơ sở dữ liệu** |

|  |  |  |
| --- | --- | --- |
| **DETACH** | **Tách tự động các thực thể con khỏi ngữ cảnh** | **Khi không muốn cập nhật các thay đổi của thực thể cha và con** |

|  |  |  |
| --- | --- | --- |
| **ALL** | **Áp dụng tất cả các hành vi trên** | **Khi muốn tất cả các thao tác lan truyền** |

**Add 1 danh sách**

public class Main {

public static void main(String[] args) {

// Tạo danh sách ban đầu

List<Float> number = new ArrayList<>();

// Tạo danh sách các phần tử cần thêm

List<Float> additionalNumbers = new ArrayList<>();

additionalNumbers.add(3.3f);

additionalNumbers.add(4.4f);

additionalNumbers.add(5.5f);

// Thêm toàn bộ phần tử của danh sách mới vào danh sách ban đầu

number.addAll(additionalNumbers);

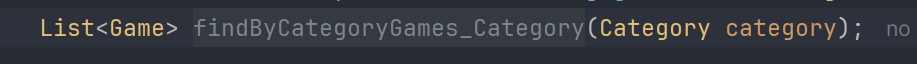
// In danh sách sau khi thêm

System.out.println("Danh sách sau khi thêm: " + number);

}

}

Giải thích dòng code này trong file gameRepository



Dòng code này là một **phương thức tùy chỉnh trong repository** Spring Data JPA. Nó được viết theo cú pháp **method naming convention** của Spring Data để tự động tạo truy vấn SQL mà không cần viết thủ công.

**Phân Tích Cú Pháp**

List<Game> findByCategoryGames\_Category(Category category);

**1. List<Game>**

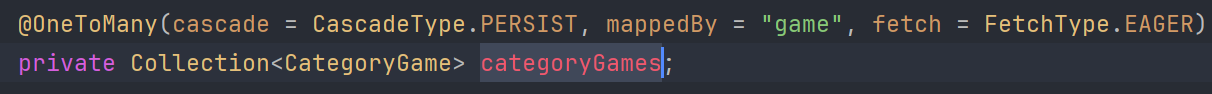
* **Loại trả về**: Một danh sách các đối tượng Game.
* Dữ liệu được trả về là danh sách các game khớp với điều kiện tìm kiếm.

**2. findBy**

* **Bắt đầu tên phương thức**: Quy ước của Spring Data JPA để tạo truy vấn.
* **findBy** biểu thị rằng phương thức này sẽ tìm kiếm các đối tượng dựa trên điều kiện.

**3. CategoryGames\_Category**

* **Đường dẫn thuộc tính**: Thể hiện quan hệ giữa các bảng.
  + CategoryGames: Là trường trong Game (quan hệ OneToMany) kết nối với bảng CategoryGame.



* + \_Category: Chỉ ra rằng điều kiện tìm kiếm nằm trên thuộc tính Category trong CategoryGame.

Ảnh có chứa văn bản, Phông chữ, ảnh chụp màn hình

Mô tả được tạo tự động

**4. Category category**

* **Tham số**: Một đối tượng Category được truyền vào để làm điều kiện tìm kiếm.
* Spring Data JPA sẽ tự động ánh xạ đối tượng Category sang cột liên quan trong bảng.

**Cách Hoạt Động**

Phương thức này yêu cầu JPA thực hiện truy vấn để tìm tất cả các game liên kết với một thể loại (Category) thông qua bảng trung gian CategoryGame.

**Tương Đương SQL**

Giả sử bạn có các bảng:

* game: Lưu thông tin game.
* category: Lưu thông tin thể loại.
* category\_game: Bảng trung gian liên kết game và category.

SQL tương đương:

SELECT g.\*

FROM game g

JOIN category\_game cg ON g.id = cg.game\_id

WHERE category\_game.category\_id = :categoryId;

**Tóm Lược**

* Phương thức findByCategoryGames\_Category giúp tự động tìm tất cả các Game liên quan đến một Category thông qua bảng trung gian CategoryGame.
* Spring Data JPA sẽ tự động ánh xạ quan hệ và tạo truy vấn SQL tương ứng dựa trên tên phương thức và quan hệ giữa các entity.

Có nghĩa là List<Game> findByCategoryGames\_Category(Category category);

**Nó sẽ tìm all cả game có categoryId ở bảng categoryGame bằng với categoryId ở bảng category**

Câu query truy vấn account game đã bán theo gameId sắp xếm giảm dần theo số lượng game đã bán.

*Chọn cột game\_id từ bảng account\_game, đếm game\_id trong account\_game và đặt tên là total\_game từ bảng account\_game, điều kiện là account\_game.status là 1 (1 là trạng thái những tài khoản đã bán) nhóm lại thoe game\_id trong account\_game, sắp xếp giảm dần theo số lượng account đã bán theo account\_game.game\_id*

**SELECT account\_game.game\_id,**

**COUNT(account\_game.game\_id) as total\_game**

**FROM account\_game**

**WHERE account\_game.status = 1**

**GROUP BY account\_game.game\_id**

**ORDER BY total\_game DESC;**

**Câu query trên nhưng làm việc với entiry**

** AccountGame là Entity trong Java:**

** ag.game.gameId:**

* **Tham chiếu đến trường gameId thông qua quan hệ với Entity Game.**

** COUNT(AccountGame.game.gameId):**

* **Đếm số lần gameId xuất hiện trong các bản ghi có status = 1.**

** GROUP BY AccountGame.game.gameId:**

* **Gom nhóm kết quả theo gameId.**

** ORDER BY totalGame DESC:**

* **Sắp xếp theo số lượng đã bán (totalGame) giảm dần.**

** List<Object[]>:**

* **Kết quả của truy vấn sẽ là danh sách các mảng đối tượng (Object[]), trong đó:**
  + **Object[0]: Là gameId.**
  + **Object[1]: Là totalGame.**

**Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động**

Status mình xài kiểu boolean trong database nó là 1 tương ứng với true

Ở trên query bị lỗi vì spring jpa yêu cầu phải có alias từ những thực thể được truy xuất

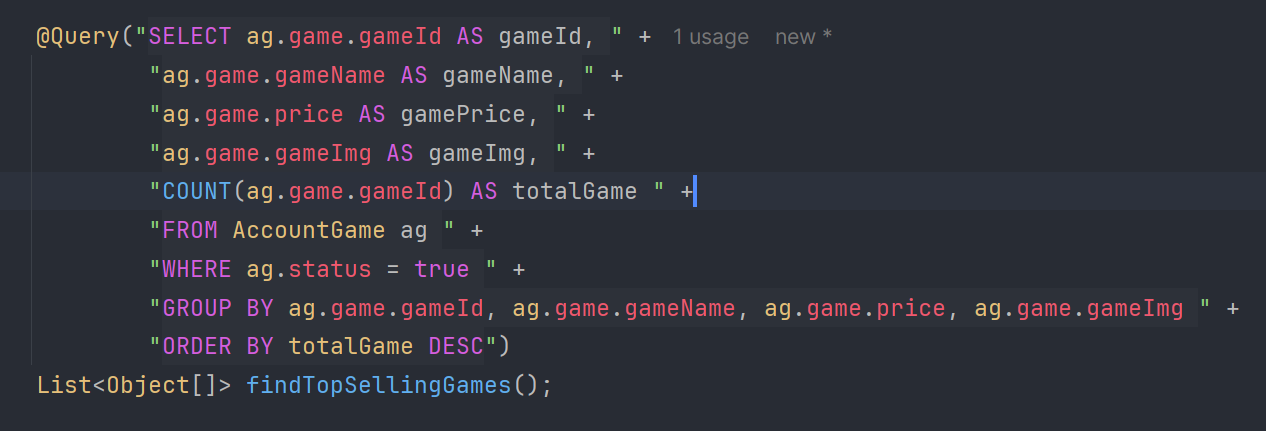
Còn lại những thứ khác vì vẫn đúng

**Bạn đang gọi đầy đủ AccountGame thay vì sử dụng alias. JPQL yêu cầu phải sử dụng alias nếu muốn thao tác với các trường. Ở dưới câu query đúng**

**Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động**

Đây là câu query phiên bản cuối ở trên chỉ select gameId nhưng ở dưới mình muốn select thêm 3 thuộc tính nữa vẫn count theo gameId



Nhưng thay vì sử dụng object thì mình dùng GameSaleDto để nhận dữ liệu

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

 **Constructor Expression:**

* Phần new com.webgame.webgame.dto.GameSalesDTO(...) trong query cho phép JPA gọi trực tiếp constructor của DTO và truyền kết quả query vào.
* Các trường trong SELECT phải trùng với tham số trong constructor của DTO.

 **GROUP BY và COUNT:**

* GROUP BY bảo đảm kết quả được nhóm theo gameId, gameName, gamePrice, và gameImg.
* COUNT(ag.game.gameId) đếm số lượng tài khoản bán được cho từng game.

 **Trả về danh sách DTO:**

* Thay vì danh sách các mảng đối tượng (List<Object[]>), kết quả trả về sẽ là danh sách các đối tượng GameSalesDTO.

Patination

Giải thích phần patination dùng cho admin bên trong file GameServiceImp

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

Phương thức findPaginated thực hiện phân trang và sắp xếp dữ liệu trong cơ sở dữ liệu. Dưới đây là giải thích các thành phần chính:

1. **Tham số đầu vào:**
   * pageNo: Số trang mà người dùng yêu cầu (đếm từ **1**, không phải **0**).
   * pageSize: Số lượng mục (records) trên mỗi trang.
   * sortField: Tên cột trong cơ sở dữ liệu (nếu là truy vấn native, còn tên thực thế nếu truy vấn qua jpa) để sắp xếp .
   * sortDirection: Hướng sắp xếp, có thể là ASC (tăng dần) hoặc DESC (giảm dần).
2. **Xử lý sắp xếp (Sort):**
   * So sánh giá trị của sortDirection với ASC.
   * Nếu là ASC, sắp xếp theo thứ tự tăng dần (Sort.by(sortField).ascending()).
   * Nếu là DESC, sắp xếp theo thứ tự giảm dần (Sort.by(sortField).descending()).
3. **Tạo đối tượng Pageable:**
   * PageRequest.of: Tạo đối tượng Pageable, chứa thông tin về số trang (pageNo - 1), kích thước trang (pageSize), và thứ tự sắp xếp (sort).
   * Lưu ý: Pageable sử dụng chỉ số trang bắt đầu từ **0**, còn mình show lên ho admin bắt đầu từ 1 vì vậy cần trừ **1** từ pageNo.
4. **Truy vấn dữ liệu:**
   * gameRepository.findAll(pageable): Truy vấn tất cả bản ghi từ gameRepository với điều kiện phân trang và sắp xếp được xác định trong Pageable.
   * Kết quả là một đối tượng Page<Game> chứa danh sách các mục thuộc trang yêu cầu, cùng với thông tin phân trang như tổng số trang, tổng số mục, v.v.

Code trong danh sách game hiện có

**Tóm tắt**

Phương thức findPaginated giúp hiển thị danh sách game với dữ liệu được phân trang, và truyền các thông tin cần thiết vào view để xây dựng giao diện phân trang (bao gồm danh sách game, trang hiện tại, tổng số trang, và tổng số bản ghi).

<div th:if="${totalPages > 1}" class="container text-center">  
 <div class="row align-items-start">  
 <div class="col">  
<!--nếu muốn xài variable expression thì phải dùng như phía dưới-->  
 Total Rows: [[${totalItems}]]  
 </div>  
 <div class="col">  
 <span th:each="i : ${#numbers.sequence(1, totalPages)}">  
 <a th:if="${currentPage != i}" th:href="@{'/page/' + ${i}}">[[${i}]]</a>  
<!-- &nbsp; là viết tắt của "non-breaking space", tức là khoảng trắng không ngắt dòng-->  
 <span th:unless="${currentPage != i}">[[${i}]]</span> &nbsp; &nbsp;  
 </span>  
 </div>  
 <div class="col">  
 <a th:if="${currentPage < totalPages}" th:href="@{'/page/' + ${currentPage + 1}}">Next</a>  
 <span th:unless="${currentPage < totalPages}">Next</span>  
 </div>  
  
 <div class="col">  
 <a th:if="${1 < currentPage < totalPages}" th:href="@{'/page/' + ${currentPage - 1}}">Next</a>  
 <span th:unless="${1 < currentPage < totalPages}">Back</span>  
 </div>  
  
 <div class="col">  
 <a th:if="${currentPage < totalPages}" th:href="@{'/page/' + ${totalPages}}">Last</a>  
 <span th:unless="${currentPage < totalPages}">Last</span>  
 </div>  
  
 </div>  
 </div>

Đoạn mã HTML này dùng Thymeleaf để xây dựng thanh phân trang (pagination) trong giao diện, giúp người dùng chuyển giữa các trang của danh sách Employee. Dưới đây là giải thích từng phần:

1. **<div th:if="${totalPages > 1}">**:
   * Toàn bộ khối mã bên trong chỉ hiển thị nếu tổng số trang (totalPages) lớn hơn 1, tức là chỉ hiện thanh phân trang khi có nhiều hơn một trang.
2. **Total Rows: [[${totalItems}]]**:
   * Hiển thị tổng số dòng (bản ghi) có trong toàn bộ danh sách Employee, sử dụng biến totalItems.
3. **<span th:each="i : ${#numbers.sequence(1, totalPages)}">**:
   * Vòng lặp Thymeleaf lặp qua các số từ 1 đến totalPages. Biến i đại diện cho số trang hiện tại trong mỗi lần lặp.
   * #numbers.sequence(1, totalPages) tạo ra một chuỗi số từ 1 đến tổng số trang.
4. **<a th:if="${currentPage != i}" th:href="@{'/page' + ${i}}">[[${i}]]</a>**:
   * Nếu trang hiện tại (currentPage) khác với i, hiển thị số trang i dưới dạng một liên kết (<a>). Liên kết này trỏ tới /page/{i}, cho phép người dùng chuyển tới trang đó khi nhấp vào.
   * Ví dụ, nếu i là 2 và currentPage là 1, đường dẫn sẽ là /page/2.
5. **<span th:unless="${currentPage != i}">[[${i}]]</span>**:
   * Nếu currentPage bằng i (trang hiện tại), số trang sẽ hiển thị dưới dạng văn bản (<span>), không phải liên kết. Điều này giúp hiển thị trang hiện tại mà người dùng đang ở, làm cho nó dễ nhận biết.
6. **<a th:if="${currentPage < totalPages}" th:href="@{'/page/' + ${currentPage + 1}}">Next</a>**:
   * Hiển thị liên kết "Next" để chuyển đến trang tiếp theo nếu currentPage nhỏ hơn totalPages. Đường dẫn sẽ trỏ tới /page/{currentPage + 1}.
7. **<span th:unless="${currentPage < totalPages}">Next</span>**:
   * Nếu currentPage bằng totalPages (tức là đang ở trang cuối cùng), "Next" sẽ hiển thị dưới dạng văn bản (<span>), không phải liên kết, để cho biết rằng không thể chuyển tiếp.
8. **<a th:if="${currentPage < totalPages}" th:href="@{'/page/' + ${totalPages}}">Last</a>**:
   * Hiển thị liên kết "Last" để chuyển đến trang cuối cùng nếu currentPage nhỏ hơn totalPages. Đường dẫn sẽ trỏ tới /page/{totalPages}.
9. **<span th:unless="${currentPage < totalPages}">Last</span>**:
   * Nếu currentPage bằng totalPages, "Last" sẽ hiển thị dưới dạng văn bản (<span>), không phải liên kết, để cho biết đã đến trang cuối cùng.

**Tổng kết**

Đoạn mã này tạo ra thanh phân trang bao gồm:

* Tổng số bản ghi (Total Rows).
* Các số trang (có thể nhấp để chuyển trang).
* Liên kết "Next" và "Last" cho phép di chuyển đến trang tiếp theo hoặc trang cuối cùng, nếu có.

Giải thích dòng query dưới trong GameRepository

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

Câu query này là một **native SQL query** (truy vấn SQL thuần) được sử dụng để lấy thông tin về các **game** cùng với số lượng tài khoản liên kết và danh sách thể loại (category) của từng game. Dưới đây là giải thích chi tiết:

**Cấu trúc của query**

1. **Phần SELECT**:
   * **g.game\_img, g.game\_name, g.description, g.game\_id, g.price**: Lấy các thông tin cơ bản của game, bao gồm:
     + Hình ảnh (**game\_img**),
     + Tên game (**game\_name**),
     + Mô tả (**description**),
     + ID của game (**game\_id**),
     + Giá của game (**price**).
   * **COUNT(ag.account\_game\_id) AS quantity**: Đếm số lượng tài khoản liên kết với game từ bảng **account\_game**.
   * **GROUP\_CONCAT(DISTINCT c.category\_name SEPARATOR ', ') AS category\_list**: Ghép tên các thể loại (**category\_name**) liên kết với game thành một chuỗi, phân tách bằng dấu phẩy (,). Chỉ ghép những tên khác nhau (DISTINCT).
2. **Phần FROM và JOIN**:
   * **FROM game g**: Lấy dữ liệu từ bảng **game** làm bảng chính.
   * **LEFT JOIN account\_game ag ON g.game\_id = ag.game\_id**: Kết nối với bảng **account\_game** để lấy thông tin tài khoản liên kết với game (LEFT JOIN đảm bảo vẫn lấy game ngay cả khi không có tài khoản nào liên kết).
   * **LEFT JOIN category\_game cg ON g.game\_id = cg.game\_id**: Kết nối với bảng **category\_game** để tìm các thể loại liên kết với game.
   * **LEFT JOIN category c ON cg.category\_id = c.category\_id**: Kết nối với bảng **category** để lấy tên thể loại (**category\_name**).
3. **Phần GROUP BY**:
   * **GROUP BY g.game\_img, g.game\_name, g.description, g.game\_id, g.price**: Nhóm dữ liệu theo từng game (dựa trên hình ảnh, tên, mô tả, ID, và giá) để thực hiện các phép tính như **COUNT** và **GROUP\_CONCAT**.
4. **Phần nativeQuery**:
   * Vì sử dụng **nativeQuery = true**, câu lệnh này được thực thi trực tiếp bằng SQL, không qua JPQL, nên các hàm như **GROUP\_CONCAT** (hàm của MySQL) được sử dụng.
5. **Phân trang (Pagination)**:
   * **Pageable**: Truy vấn này hỗ trợ phân trang, trả kết quả theo từng trang thông qua đối tượng **Page<Object[]>**.

**Dữ liệu trả về**

* Mỗi dòng kết quả đại diện cho một game, với các cột:
  1. **Hình ảnh game** (game\_img),
  2. **Tên game** (game\_name),
  3. **Mô tả game** (description),
  4. **ID game** (game\_id),
  5. **Giá game** (price),
  6. **Tổng số tài khoản liên kết** (quantity),
  7. **Danh sách thể loại** (category\_list) được ghép thành chuỗi, phân tách bởi dấu phẩy.

Tóm lại Page chính là toàn bộ dòng trả về còn object[] là 1 mảng object đại diện cho 7 cột trên. Cụ thể object[0] đại diện cho hình ảnh, object [1] đại diện cho tên game, cứ như thế cho đến object[6] đại diện cho danh sách thể loại

Giải thích class ConfigStaticFile: dùng để config đc dẫn đến file đc upload lên

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ

Mô tả được tạo tự động

Cấu hình ConfigStaticFile được dùng để chỉ định các tệp tĩnh (static files) như hình ảnh, CSS, hoặc JavaScript mà Spring Boot có thể phục vụ qua URL. /\*\* chính là chấp nhập tất cả đường dẫn con phía sau đường dẫn đã định nghĩa.

**Ý nghĩa từng dòng:**

1. **@Configuration:**
   * Đánh dấu đây là một lớp cấu hình trong Spring, nơi bạn có thể tùy chỉnh các thiết lập mặc định.
2. **implements WebMvcConfigurer:**
   * Interface này cho phép bạn ghi đè các phương thức để cấu hình thêm cho Spring MVC, chẳng hạn như định nghĩa cách xử lý file tĩnh, định tuyến, hoặc interceptor.
3. **addResourceHandlers:**
   * Phương thức này cho phép bạn định nghĩa các đường dẫn URL để ánh xạ đến các thư mục tệp trên hệ thống.
4. **Cách ánh xạ:**

registry.addResourceHandler("/uploadImgGame/\*\*")

.addResourceLocations("file:src/main/java/com/webgame/webgame/uploadImgGame/");

* + **/uploadImgGame/\*\*:** Đây là đường dẫn mà người dùng sẽ nhập trên trình duyệt để truy cập tệp (VD: http://localhost:8080/uploadImgGame/example.jpg).
  + **file:src/main/java/com/webgame/webgame/uploadImgGame/:** Đây là thư mục thực trên hệ thống nơi các tệp được lưu. "file:" báo cho Spring biết đây là một thư mục tệp, không phải thư mục trong classpath.

**Hoạt động:**

Khi bạn truy cập http://localhost:8080/uploadImgGame/filename, Spring sẽ tìm tệp tương ứng trong thư mục src/main/java/com/webgame/webgame/uploadImgGame/ và trả về file đó.

**Lưu ý:**

* Đường dẫn src/main/java/com/webgame/webgame/uploadImgGame/ này phù hợp cho phát triển nhưng không nên dùng trong môi trường production vì dễ gây nhầm lẫn giữa mã nguồn và tài nguyên tĩnh.
* Nếu triển khai lên server, bạn nên dùng đường dẫn ngoài project, ví dụ: "file:/opt/webgame/uploadImgGame/".

Giải thích câu query này trong file gameRepository

Ảnh có chứa văn bản, Phần mềm đa phương tiện, phần mềm, ảnh chụp màn hình

Mô tả được tạo tự động Câu query này làm những việc sau:

1. **Mục tiêu:**  
   Truy xuất danh sách **thể loại (Category)** thuộc về một game có gameId được chỉ định.
2. **Cách hoạt động:**
   * **cg.category**: Truy xuất trường category từ đối tượng CategoryGame.
   * **WHERE cg.game.gameId = :gameId**: Lọc các bản ghi trong bảng CategoryGame dựa trên gameId của game.
3. **Đầu ra:**  
   Trả về danh sách Category liên kết với gameId đã cho.
4. **Cách sử dụng tham số:**
   * @Param("gameId") ánh xạ tham số gameId trong query với giá trị được truyền vào phương thức.

**Kết quả:**

Phương thức này sẽ trả về danh sách các thể loại (Category) của một game cụ thể trong cơ sở dữ liệu.

**Tại sao Serializable cần thiết cho khóa chính trong JPA?**

**1. Vai trò của Serializable:**

* **Serializable** là một giao diện trong Java giúp các đối tượng có thể được **tuần tự hóa (serialization)**.
* **Tuần tự hóa** là quá trình chuyển đổi trạng thái của một đối tượng thành dạng dữ liệu có thể lưu trữ hoặc truyền qua mạng, và sau đó có thể tái tạo lại đối tượng đó (deserialization).

**2. Yêu cầu của JPA:**

* **JPA** yêu cầu các **khóa chính composite** (được ánh xạ qua @Embeddable hoặc @IdClass) phải thực hiện **Serializable** để đảm bảo:
  + Chúng có thể được sử dụng làm **khóa chính (Primary Key)** trong các hoạt động liên quan đến cơ sở dữ liệu.
  + Đảm bảo trạng thái của các khóa chính có thể được truyền qua các phiên bản Hibernate hoặc EntityManager, ví dụ: trong các **truy vấn, cache hoặc session**.
  + Dễ dàng so sánh hoặc lưu trữ các đối tượng trong bộ nhớ hoặc trong cơ sở dữ liệu.

**3. Lý do thực tế:**

* Khi Hibernate/JPA xử lý một đối tượng thực thể (entity), nó có thể sử dụng trạng thái của khóa chính trong các bối cảnh khác nhau, như:
  + **Cache cấp 1/cấp 2** (First-level cache, Second-level cache).
  + **Serialization** khi truyền dữ liệu giữa các lớp hoặc module.
  + **Hashing** khi dùng trong các cấu trúc dữ liệu như HashMap.

**4. Áp dụng trong CategoryGameId:**

* CategoryGameId là một **khóa chính tổng hợp (composite key)**, vì thế nó cần:
  + **Tuần tự hóa:** để Hibernate có thể lưu hoặc truyền trạng thái của khóa chính giữa các thực thể hoặc session.
  + **So sánh:** để kiểm tra tính duy nhất của cặp khóa chính (categoryId, gameId).

**5. Hậu quả nếu không Serializable:**

* **Lỗi runtime:** Nếu không triển khai Serializable, bạn sẽ gặp lỗi như:
  + org.hibernate.MappingException: Composite-id class must implement Serializable.
* Hibernate không thể xử lý các đối tượng CategoryGameId trong các tình huống tuần tự hóa.

**Tóm lại:**

Serializable trong CategoryGameId đảm bảo khóa chính tổng hợp có thể:

1. **Tuần tự hóa:** Được lưu trữ hoặc truyền qua session hoặc mạng.
2. **Tương thích:** Với các yêu cầu của JPA/Hibernate.
3. **So sánh được:** Đảm bảo hai cặp categoryId và gameId có thể được so sánh hoặc ánh xạ chính xác.

Giả thích phần này trong new-game

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Phần mềm đa phương tiện

Mô tả được tạo tự động

Trong Thymeleaf:

1. **th:object**:
   * Xác định đối tượng (object) được liên kết với form.
   * Mọi thuộc tính của đối tượng này sẽ được ánh xạ vào các trường dữ liệu (field) của form.
   * Trong đoạn code trên, th:object="${newGame}" cho biết form này liên kết với đối tượng **newGame**.
2. **th:field**:
   * Liên kết một trường (input, textarea, checkbox, ...) với một thuộc tính cụ thể của đối tượng đã khai báo trong **th:object**.
   * Trong ví dụ:
     + th:field="\*{gameName}" liên kết trường input với thuộc tính **gameName** của đối tượng **newGame**.
     + Khi người dùng nhập dữ liệu và gửi form, giá trị trong trường này sẽ được gán vào **newGame.gameName**.

**Kết luận:**

* **th:object**: Đặt đối tượng nguồn cho toàn bộ form.
* **th:field**: Ánh xạ một trường cụ thể tới một thuộc tính trong đối tượng.

Tương tự với các trường khác của newGame (GameFormDto)

Giải thích phần /savegame (khi user ấn save game)

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm, Phần mềm đa phương tiện

Mô tả được tạo tự động

Phương thức này xử lý yêu cầu POST khi thêm game mới.

**Giải thích các phần:**

1. **@PostMapping("/saveGame")**:
   * Xử lý các yêu cầu HTTP POST đến URL /saveGame.
2. **@ModelAttribute("newGame")**:
   * @ModelAttribute đúng là để nhận dữ liệu từ đối tượng đã được buộc vào form là newGame và ánh xạ vào đối tượng vào trong code.
   * Các trường trong form phải khớp với thuộc tính trong **GameFormDto** (ví dụ: gameName, price, ...).
3. **RedirectAttributes redirectAttributes**:
   * Dùng để gửi thông báo (success/error) từ controller về view (thông qua redirect).
4. **Logic bên trong**:
   * **gameService.saveGame(gameFormDto)**: Gọi service để lưu thông tin game vào database.
   * **redirectAttributes.addFlashAttribute("success/error", ...)**:
     + Gửi thông báo thành công hoặc thất bại về cho view.
5. **return "redirect:/addGameForm";**:
   * Điều hướng người dùng về trang thêm game sau khi xử lý xong.

**Tóm lại:**

* Lấy dữ liệu từ form, lưu vào database, và gửi thông báo về trang thêm game.

**Giải thích code dưới trong file GameServiceImp**

Ảnh có chứa văn bản, ảnh chụp màn hình, Phông chữ, phần mềm

Mô tả được tạo tự động

Phương thức **getGameFromDto** có nhiệm vụ **chuyển đổi dữ liệu từ GameFormDto sang đối tượng Game**. Cụ thể, nó thực hiện các bước sau:

**Chi tiết chức năng:**

1. **Sao chép các giá trị cơ bản từ GameFormDto sang Game:**
   * **gameName**, **description**, và **price** của GameFormDto được sao chép và gán cho đối tượng Game.
2. **Xử lý lưu ảnh:**
   * **Sinh tên file duy nhất**:  
     Một tên file duy nhất được tạo bằng cách kết hợp một UUID (mã định danh duy nhất) và tên file gốc từ đối tượng MultipartFile.
   * **Tạo đường dẫn lưu ảnh**:  
     Ảnh được lưu trong thư mục uploadImgGame với đường dẫn src/main/java/com/webgame/webgame/uploadImgGame/.
   * **Sao chép ảnh**:  
     File từ input stream của gameFormDto.getGameImg() được sao chép vào vị trí chỉ định.  
     **Lựa chọn thay thế (StandardCopyOption.REPLACE\_EXISTING)** đảm bảo nếu có file trùng tên, file cũ sẽ bị ghi đè.
   * **getInputStream()** trả về một đối tượng InputStream đại diện cho nội dung của file. Bạn có thể sử dụng nó để đọc nội dung file hoặc ghi file vào vị trí lưu trữ, là cách hiệu quả để làm việc với file lớn mà không chiếm nhiều bộ nhớ
3. **Gán tên file vào đối tượng Game:**
   * Sau khi lưu thành công, tên file được lưu vào thuộc tính **gameImg** của đối tượng Game.
4. **Trả về đối tượng Game:**
   * Sau khi hoàn tất cập nhật thông tin từ GameFormDto, phương thức trả về đối tượng Game đã được chỉnh sửa.

**Tóm tắt ý nghĩa:**

Phương thức này giúp:

* Lấy dữ liệu từ form người dùng (thông qua GameFormDto).
* Cập nhật các thuộc tính của Game.
* Lưu file ảnh của game vào thư mục cố định trên server.
* Trả về đối tượng Game đã được cập nhật, sẵn sàng để lưu vào database.

**Giải thích code phía dưới trong file GameServiceImp**

Ảnh có chứa văn bản, ảnh chụp màn hình, phần mềm

Mô tả được tạo tự động

Phương thức **getCategoryGameFromDto** được sử dụng để tạo danh sách **CategoryGame** từ dữ liệu nhận được trong **GameFormDto** với một đối tượng **Game**. Dưới đây là giải thích chi tiết từng phần:

**1. Mục đích:**

* Tạo danh sách các đối tượng **CategoryGame** từ:
  + **Danh sách ID của các category** có trong **GameFormDto**.
  + Đối tượng **Game** liên kết.

**2. Cách hoạt động:**

**Input:**

* **GameFormDto gameFormDto**: Chứa danh sách ID thể loại mà người dùng đã chọn qua form (danh sách **categoryIds**).
* **Game game**: Đối tượng game đã được xử lý (ví dụ: được lưu hoặc cập nhật).

**Output:**

* **Danh sách CategoryGame** được tạo từ các ID thể loại và đối tượng game.

**3. Phân tích đoạn mã:**

public List<CategoryGame> getCategoryGameFromDto(GameFormDto gameFormDto, Game game) {

// Tạo một danh sách để lưu các đối tượng CategoryGame.

List<CategoryGame> categoryGames = new ArrayList<>();

* **categoryGames**: Danh sách sẽ chứa các đối tượng **CategoryGame** sau khi xử lý.

**Vòng lặp qua danh sách ID thể loại:**

for (Long categoryId : gameFormDto.getCategoryIds()) {

Optional<Category> optionalCategory = categoryRepository.findById(categoryId);

* **gameFormDto.getCategoryIds()**: Lấy danh sách ID các thể loại được chọn từ DTO.
* **categoryRepository.findById(categoryId)**: Tra cứu thể loại trong cơ sở dữ liệu bằng **categoryId**.

**Xử lý khi thể loại tồn tại:**

if (optionalCategory.isPresent()) {

Category category = optionalCategory.get();

// hàm constructor categoryGame nhận 2 param game, category còn composite primary key sẽ được lấy ra từ 2 entity này

CategoryGame categoryGame = new CategoryGame(game, category);

categoryGames.add(categoryGame);

}

* **optionalCategory.isPresent()**: Kiểm tra nếu thể loại tồn tại.
* **categoryGame = new CategoryGame(game, category)**:
  + Tạo đối tượng **CategoryGame**, liên kết giữa **game** và **category**.
* **categoryGames.add(categoryGame)**: Thêm vào danh sách.

**Xử lý khi thể loại không tồn tại:**

else {

System.out.println("Category ID " + categoryId + " not found!");

}

* Ghi log thông báo khi không tìm thấy thể loại.

**4. Trả về danh sách:**

return categoryGames;

}

* Trả về danh sách các đối tượng **CategoryGame**.

**5. Tóm tắt:**

* Phương thức này dùng để:
  + Liên kết một **Game** với danh sách **Category** thông qua đối tượng **CategoryGame**.
  + Đảm bảo mỗi liên kết được kiểm tra trước khi tạo.
  + Quản lý các thể loại không tồn tại với thông báo lỗi.