**JPA**

|  |
| --- |
| **Trong JPA, mỗi phiên làm việc với cơ sở dữ liệu (database session) được quản lý bởi một EntityManager. Để đảm bảo tính toàn vẹn của dữ liệu, JPA hỗ trợ khái niệm giao dịch (transaction) - trong đó các thao tác với cơ sở dữ liệu được thực hiện trong một khối giao dịch.**  **Trong JPA, mọi thao tác thay đổi dữ liệu như INSERT, UPDATE, DELETE đều phải được thực hiện trong một giao dịch. Điều này giúp đảm bảo tính nhất quán và độ tin cậy của dữ liệu. Trong đoạn mã trên, em là một đối tượng EntityManager, nó cung cấp các API để tương tác với cơ sở dữ liệu. Phương thức getTransaction() của EntityManager trả về một đối tượng EntityTransaction, đại diện cho một giao dịch với cơ sở dữ liệu. Sau khi có đối tượng EntityTransaction, bạn có thể gọi phương thức begin() để bắt đầu giao dịch, commit() để xác nhận các thay đổi vào cơ sở dữ liệu, và rollback() để hủy bỏ các thay đổi nếu có lỗi xảy ra.**  **Đoạn mã EntityTransaction tr = em.getTransaction(); thực hiện các bước sau:**   1. **em.getTransaction(): Phương thức này trả về một đối tượng EntityTransaction liên kết với EntityManager hiện tại. EntityTransaction là một giao diện cung cấp các phương thức để quản lý giao dịch.** 2. **EntityTransaction tr = ...: Lưu trữ tham chiếu đến đối tượng EntityTransaction vào biến tr. Từ đây, bạn có thể sử dụng tr để điều khiển giao dịch.**   **Sau khi có EntityTransaction tr, bạn có thể:**   * **Bắt đầu giao dịch: tr.begin().** * **Xác nhận (commit) giao dịch: tr.commit().** * **Hủy bỏ (rollback) giao dịch: tr.rollback().** |

**Lưu ý ở mỗi class :**

* + Mỗi class phải implement Serializable
  + Nếu class cha các biến phải có protected
  + Đối với các biến có kiểu là TimeStamp

@Column(name = "Timestamp", columnDefinition = "DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP")

**Cách làm bài tạo mối quan hệ**

**Mối quan hệ 1-1 : có 2 trường hợp**

**TH1: Đối tượng phụ thuộc không có Id**

Đối tượng chủ thể là đối tượng chứa khóa ngoại (foreign key) để liên kết với đối tượng phụ thuộc.

Cả hai đối tượng đều khai báo @OneToOne nếu xét mối quan hệ hai chiều.

Nếu xét một chiều, chỉ đối tượng phụ thuộc khai báo @OneToOne.

Cách làm:

Ta chọn phía class phụ thuộc .Paste đoạn code sau

Ví dụ ta có class ClassProfire phụ thuộc vào class Clazz thì trong class ClassProfire ta code như sau:

**@Id**

**@OneToOne**

**@JoinColumn(name = "class\_id"** **, unique = true, nullable = false)**

**private Clazz clazz**

**TH2: Bảng phụ có thuộc có id rồi .**

Tương tự như trên nhưng bỏ @Id

**Mối quan hệ 1-n**

− Phía One:

Thêm @OneToMany với thuộc tính mappedBy.

− Phía Many

Thêm:

@ManyToOne

@JoinColumn với thuộc tính name

**Nếu xét một hướng, thì hướng đi từ many to one (Hay phía nhiều)**

A computer code with text

Description automatically generated

**Giải thích:**

|  |
| --- |
| **@ManyToOne(fetch = FetchType.*LAZY*)** |

Khi một thực thể (entity) có một mối quan hệ với một hoặc nhiều thực thể khác, JPA cung cấp hai cách để xác định cách các thực thể liên kết được tải:

1. FetchType.EAGER (tải ngay):
   * Khi sử dụng FetchType.EAGER, thực thể liên kết sẽ được tải cùng lúc với thực thể chính.
   * Điều này có nghĩa là khi bạn truy xuất một thực thể, tất cả các thực thể liên kết cũng sẽ được tải vào bộ nhớ.
   * Cách này có thể hữu ích khi bạn biết rằng bạn sẽ luôn cần truy cập các thực thể liên kết, nhưng nó có thể dẫn đến tải dữ liệu không cần thiết và ảnh hưởng đến hiệu suất.
2. FetchType.LAZY (tải khi cần):
   * Khi sử dụng FetchType.LAZY, thực thể liên kết sẽ chỉ được tải khi chúng thực sự được truy cập.
   * Điều này có nghĩa là khi bạn truy xuất một thực thể, các thực thể liên kết sẽ không được tải ngay lập tức. Thay vào đó, chúng sẽ chỉ được tải khi bạn cố gắng truy cập chúng.
   * Cách này có thể cải thiện hiệu suất bằng cách tránh tải dữ liệu không cần thiết, đồng thời giảm sử dụng bộ nhớ.

Ví dụ, trong mối quan hệ giữa Order và Customer, nếu bạn sử dụng @ManyToOne(fetch = FetchType.LAZY) trên trường customer trong thực thể Order, điều này có nghĩa rằng khi bạn truy xuất một Order, thực thể Customer liên kết sẽ không được tải ngay lập tức. Thay vào đó, Customer sẽ chỉ được tải khi bạn cố gắng truy cập nó (ví dụ: order.getCustomer()).

Tuy nhiên, sử dụng FetchType.LAZY cũng có thể dẫn đến các vấn đề, chẳng hạn như "N+1 select problem", khi bạn phải thực hiện nhiều truy vấn để tải các thực thể liên kết. Vì vậy, bạn cần cân nhắc kỹ lưỡng khi sử dụng cách tải dữ liệu này và xem xét các chiến lược tối ưu hóa, chẳng hạn như sử dụng fetch joins hoặc batch loading.

**Mối quan hệ n-n**

-Ví dụ ta có bảng Group và Student là mối quan hệ nhiều nhiều. Chọn đại 1 class và past đoạn code sau vào. Ví dụ chọn class **Group**.

@ManyToMany

@JoinTable(

name = "CourseStudent", //tên của bảng mới

joinColumns = @JoinColumn(name = "CourseID"), //id của bảng được chọn

inverseJoinColumns = @JoinColumn(name = "StudentID") // id của bảng còn lại

)

private Set<Students> students; //nhớ khai báo thuộc tính của class còn lại .Đây là Student.

**Nhúng**

|  |
| --- |
| **@Embedded** |

Chú thích này được sử dụng để đánh dấu một thuộc tính của thực thể (entity) là một tập hợp (collection) các giá trị đơn giản (như String, Integer, etc.).

Nó cho phép bạn lưu trữ một tập hợp các giá trị (như số điện thoại, địa chỉ email, v.v.) liên quan đến một thực thể. **Ví dụ một sinh viên có nhiều số điện thoại**

**Tham chiếu**

**@ElementCollection**

**@CollectionTable(name="phones", joinColumns = @JoinColumn(name="student\_id"))**

**@Column(name="phone", nullable = false)**

**protected Set<String> phones;**

Chú thích này được sử dụng để định nghĩa bảng riêng (collection table) để lưu trữ các giá trị trong tập hợp.

@CollectionTable(name="phones",joinColumns=@JoinColumn(name="student\_id"))

private Set<String> phones;

Ví dụ, khi một sinh viên có nhiều số điện thoại, bạn có thể lưu trữ các số điện thoại của sinh viên đó trong bảng "phones", sử dụng cột "student\_id" để liên kết chúng với bản ghi tương ứng trong bảng "Student".

**Kế thừa**

Có 3 chiến lược để ánh xạ:

−InheritanceType.SINGLE\_TABLE

-InheritanceType.TABLE\_PER\_CLASS

− InheritanceType.JOINED

**\*Đối với class cha**

|  |
| --- |
| **@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)**  **@DiscriminatorColumn(name="Discriminator", discriminatorType = DiscriminatorType.*STRING*)** |

**\*Đối với class con**

|  |
| --- |
| **@DiscriminatorValue("Student")** |

Lưu ý : Kế thừa các lớp con không cần chỉ định id.

Giải thích code:

@Inheritance là một annotation trong Hibernate dùng để định nghĩa chiến lược ánh xạ kế thừa (inheritance mapping) giữa các class Java và bảng trong cơ sở dữ liệu.

InheritanceType.SINGLE\_TABLE là một trong ba chiến lược kế thừa được Hibernate hỗ trợ, cùng với JOINED và TABLE\_PER\_CLASS.

Khi sử dụng InheritanceType.SINGLE\_TABLE:

1. Một bảng duy nhất: Tất cả các lớp con (subclasses) sẽ được lưu trữ trong cùng một bảng với cha (superclass). Bảng này sẽ có thêm một cột "discriminator" để phân biệt các lớp con.
2. Cột "discriminator": Hibernate sẽ tự động thêm một cột "discriminator" vào bảng cha để lưu trữ thông tin về lớp con. Giá trị của cột này sẽ là giá trị được chỉ định bằng **@DiscriminatorValue** ở mỗi lớp con.
3. Truy vấn dễ dàng: Khi truy vấn, Hibernate sẽ tự động phân biệt các lớp con dựa vào giá trị của cột "discriminator" và trả về đối tượng tương ứng.

Ưu điểm của InheritanceType.SINGLE\_TABLE:

* Đơn giản và dễ triển khai
* Truy vấn nhanh vì chỉ có một bảng
* Không cần JOIN nhiều bảng

Nhược điểm:

* Bảng sẽ có nhiều cột để lưu trữ các trường của các lớp con
* Nếu có nhiều lớp con, bảng sẽ trở nên rất lớn và phức tạp

Chiến lược này thích hợp khi có ít lớp con, hoặc khi các lớp con không quá khác biệt về cấu trúc dữ liệu.

**Tự quan hệ (Nhân viên quản lí nhân viên)**

//MỐI QUAN HỆ TUẦN TỰ  
@ManyToOne  
@JoinColumn(name="manager\_id") //manager\_id là tên cột trong bảng students chỉ định khóa ngoại tới bảng students (chính nó)  
protected Student manager;

**Intsall (Quy trình trước khi làm bài)**

B1: Tạo maven project

B2: Tìm hiểu JDK nào thì download JDK đó

**B3: Install các dependences:**

Hibernate : Hibernate bỏ dòng pom

***+Chọn 1 trong 2 :***

Microsoft JDBC Driver for SQL Server -> chọn đúng JDK

MariaDB Java Client

B4: Chuyển sang JPA project

Cách 1 : Chuột phải vào giao diện chọn - configure -> convert to JPA Project (chỉ có khi download Eclipse EE Developer)

Cách 2 : Tạo một Maven Project → Click phải trên project → **Properties** → **Project** **Facets** → **(May be: Convert to faceted from…)** → check vào **JPA** → **Further configuration available …** → JPA Implementation: **Disable Library Configuration** → **OK** → **Apply and Close** (lưu ý ngay Java chọn tương thích version)

**-> Nếu hibernate chọn tự động :**

B5: Chọn **Java Buil Path -> Edit - > chọn JDK -> Installed JREs -> xóa và add lại ở JER home -> chọn folder jdk đó (lưu ý không để C/ECLIPSE gì đó)**

B6: Chọn **Java** **Compiler** -> đổi lại

B7: Cấu hình **Persistence.xml**

**-Nếu lỗi : Chuột phải project -> chọn property ->JPA -> Error -> Multi ->Ignore**

**Lưu ý :** Nếu sai mật khẩu SQL Server vào MySQL chọn Windown -> sa -> đổi passs.

**Các lỗi mắc phải khi làm:**

-Nếu chạy SQL Server bị lỗi vào C:\Program Files\Java\jdk-17\conf\security\java -> chọn edit -> tìm ssl -> bỏ TLSv1 -> save lại và qua project gõ đại gì đó.

**ITELIJ (Gradel)**

**Cài đặt plugin :**

JPA Buddy

Lombok

**Cấu hình :**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

**Sau đó thực hiện tương tư Eclipse**