Héritage

adriencaubel.fr

Table des matières

1. Introduction
    1. Rappels
2. Impedance mismatch
    1. Définition
    2. Impedance mismatch et héritage
    3. Héritage et entités JPA
3. 3 stratégies de mapping
    1. Les 3 stratégies de mapping
    2. L'annotation @Inheritance
4. Stratégie TABLE_PER_CLASS
    1. Représentation
    2. CRUD
5. Stratégie SINGLE_TABLE
    1. Représentation
    2. CRUD
    3. Représentation complète

Introduction

Rappels

- Nous avons étudié les concepts suivants
  - La notion `@Entity`
  - Les associations `@OneToMany` ... et `mappedBy` (bidirectionnelle)

Nous allons maintenant nous intéresser au problème de l'héritage

Impedance mismatch

Définition

L'impédance mismatch fait référence aux incompatibilités ou divergences conceptuelles et techniques entre la POO et les Bases de données relationnelles

(POO)	Bases Relationnelles
Héritage	1:1
List	1:n ou n:1
Map	n:n

Impedance mismatch et héritage

- En OO il est naturel d'avoir des classes qui héritent les unes des autres
- Tandis qu'en base de données relationnelle une table ne va pas hériter d'une autre table

=> Comment représenter l'héritage en BDD ?

Héritage et entités JPA

```java
@Entity
public class User {
    @Id
    private int id
    private String name;
}

@Entity
public class Employee extends User {
    // Pas besoin de id, elle est hérité
    private int salary
}
```

3 stratégies de mapping

Les 3 stratégies de mapping

Nous avons 3 stratégie pour représenter l'héritage

- Héritage avec une table par classe `TABLE_PER_CLASS`
- Héritage avec une seule table `SINGLE_TABLE`
- Héritage avec jointure `JOINED`

L'annotation @Inheritance

On précisera la stratégie d'héritage via l'annotation `@Inheritance`

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class User {

}
```

Étudions en détail les 3 stratégies

Stratégie TABLE_PER_CLASS

Représentation

Chaque entité sera représentée par sa propre classe

```
Employee : ID | Name | Salary
User :     ID | Name
```

CRUD

Pour les opérations CRUD aucun problème :

- `em.persist(...)` suivant le type enregistrera dans la bonne table
- `em.find(User.class, 1)` suivant la valeur du premier paramètre SELECT dans la bonne table
- `user.setName(...)` de même aucun problème, le type de l'objet est précis
- `em.remove(user)` on connaît également le type

Stratégie SINGLE_TABLE

Représentation

Nous allons avoir une table unique pour ces deux entités, avec l'ensemble des colonnes de `User` et `Employee`

```
User : ID | Name | Salary
```

- Si on met des instances de User alors on utilisera que les deux premières colonnes
- Si on met des instances de Employee alors on utilisera les trois colonnes

CRUD

- `em.persist(...)` aucun problème
- `em.find(User.class, 1)`
  - Est-ce que le User qui a l'id 1 est réellement de type User ?
  - La seule façon de s'en sortir est d'ajouter une colonne technique `DTYPE` qui sera complété lors de l'insertion et qui précisera `User` ou `Employee`
- `employee.setSalary(10000)` aucun problème
- `em.remove(employee)` aucun problème

Représentation complète

```
User : ID | Name | Salary | DTYPE
```

Stratégie JOINED

Représentation

Une entité qui étend une autre entité est en relation 1:1 avec cette entité

```
User :     ID | Name
Employee : ID | Salary
```

L'ID dans User et Employee seront les mêmes pour un employee

CRUD

- `em.persist(employee)` deux requêtes `INSERT` nécessaires, une pour la table User et une autre pour Employee
- `em.find(Employee, 2)` il nous faudra une jointure sur les deux tables
- `employee.setSalary(10000)` aucun problème
- `em.remove(employee)` deux requêtes `DELETE` nécessaires