JPA Mapping

adriencaubel.fr

Table des matières

Introduction

Rappels

- Nous avons introduit la spécification JPA
  - EntityManager et opérations persist, remove, find …
  - Cycle de vie d'une entité
  - Gestion des transactions
- Nous avons également étudié `@Entity` et `@Embeddable`
- Maintenant étudions les relations 1:1, 1:n, n:1 et n:m

Impedance mismatch

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

- Par exemple on peut se demander comment de l'héritage en OO va-t-il être transformé en BDD ?
  - C'est au programmeur de résoudre ce problème
  - Heureusement, nous pouvons tirer parti d'un ORM (Object-Relational Mapping)

Les ORM permettent de créer une correspondance entre un modèle objet et un modèle relationnel de base de données

Unidirectionnelles et Bidirectionnelles

Java VS Database

Un premier constat, très important: *en java nous pouvons représenter l'association de trois manières*

- Dans une `Personne` on place un lien vers un `Passeport` (unidir.)
- Dans une `Passeport` on place un lien vers une `Personne` (unidir.)
- On place un lien dans les deux côtés (bidir.)

Rappelons que dans une base relationnelle, le problème ne se pose pas : une association représentée par une clé étrangère est par nature bidirectionnelle.

Unidirectionnelle

```java
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToOne
    private Passeport passeport;
}
```

Unidirectionnelle

```java
@Entity
public class Passeport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String numero;
}
```

- `personne.getPassword()` :
- `password.getPersonne()` :
  - car relation unidirectionnelle

Bidirectionnelle

```java
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne(mappedBy = "passeport") // Référence inverse à la relation dans `Personne`
    private Passeport passport;
}
```

- `personne.getPassword()` :
- `password.getPersonne()` :
  - `Personne` et `Passeport` ont chacun une référence à l'autre.

Relation 1:1

Unidirectionnelle

```
@Entity
public class Person {
    @OneToOne private Passeport passeport;
}
@Entity
public class Passeport {
   ... // aucune relation inverse
}
```

Maintenant si on souhaite aller plus loin et pouvoir naviguer dans les deux directions plusieurs options s'offrent à nous

- avoir deux relations unidirectionnelles
- ou avoir une relation bidirectionnelle

Aparté : Cas impossible

```java
@Entity
public class Person {
    @OneToOne private Passeport passeport;
}
@Entity
public class Passeport {
    private Person person; // aucune annotation
}
```

=> `JdbcTypeRecommendationException` - car `Password` est annoté de `@Entity` - et fait référence à `Person` aussi une entité, sans annotation JPA pour définir le type de la relation

2x OneToOne = 2x unidirectionnelle

```java
@Entity
public class Personne {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToOne
    private Passeport passeport;
}
```

2x OneToOne = 2x unidirectionnelle

```java
@Entity
public class Passeport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String numero;

    @OneToOne
    private Personne personne;
}
```

2x OneToOne : résultat

id	nom	passeport_id
1	John Doe	101

id	numero	personne_id
101	A12345678	1

Dans la plupart des cas, nous n'avons pas besoin de clés étrangères dans les deux tables pour une relation univoque.

OneToOne + mappedBy = Bidir.

```java
/* (Inverse Side) */
@Entity
public class Personne {
    @OneToOne(mappedBy = "personne")  // Inverse side, la FK n'est pas dans Personne
    private Passeport passeport;
}
```

```java
/* (Owning Side) */
@Entity
public class Passeport {
    @OneToOne // Foreign key will be here
    // @JoinColumn(name = "fk_personne_id) nom de colonne explicite
    private Personne personne;
}
```

OneToOne + mappedBy : résultat

id	nom
1	John Doe

id	numero	personne_id
101	A12345678	1

OneToOne + mappedBy : résultat

```
entityManager.getTransaction().begin();

Passeport passeport = new Passeport();
passeport.setNumero("A12345678");

// Create Personne entity and associate it with the Passeport
Personne personne = new Person();
personne.setNom("John Doe");
personne.setPasseport(passeport);

// Persist both entities
entityManager.persist(passeport);
entityManager.persist(personne);

entityManager.getTransaction().commit();
```

OneToOne + mappedBy + Cascade

Nous avons du persister nos deux entités.

```
entityManager.persist(passeport);
entityManager.persist(personne);
```

Il serait intéressant de persister automatiquement l'entité `Passeport` lorsqu'on persiste l'entité `Personne`

OneToOne + mappedBy + Cascade

```java
@Entity
public class Personne {
    @OneToOne(mappedBy = "personne", cascade = CascadeType.ALL)
    private Passeport passeport;
}
```

```java
@Entity // aucun changement pour Passeport
public class Passeport {
    @OneToOne
    private Personne personne;
}
```

OneToOne + mappedBy + Cascade

```
entityManager.getTransaction().begin();

// Crée un passeport
Passeport passeport = new Passeport();
passeport.setNumero("A12345678");

// Crée une personne et associe le passeport
Personne personne = new Personne();
personne.setNom("John Doe");
personne.setPasseport(passeport); // Cascade permettra de persister le passeport

// Persiste uniquement la personne (le passeport sera persisté automatiquement)
entityManager.persist(personne);

entityManager.getTransaction().commit();
```

OneToOne : conclusion

- Utilisation x1 `OneToOne` conduit à une relation unidirectionnelle
- Utiliser x2 `OneToOne` conduit à deux relations unidirectionnelles
  - Donc deux FK
- Utiliser `OneToOne` + `mappedBy` conduit à une relation bidirectionnelle
  - Une FK

Relation 1:n

Relation 1:n

Les choses se compliquent car plusieurs représentation en BDD sont possibles :

- Relation unidirectionnelle (trois tables)
- Relation unidirectionnelle (deux tables)
- Relation bidirectionnelle

Relation 1:n unidirectionnelle (3t)

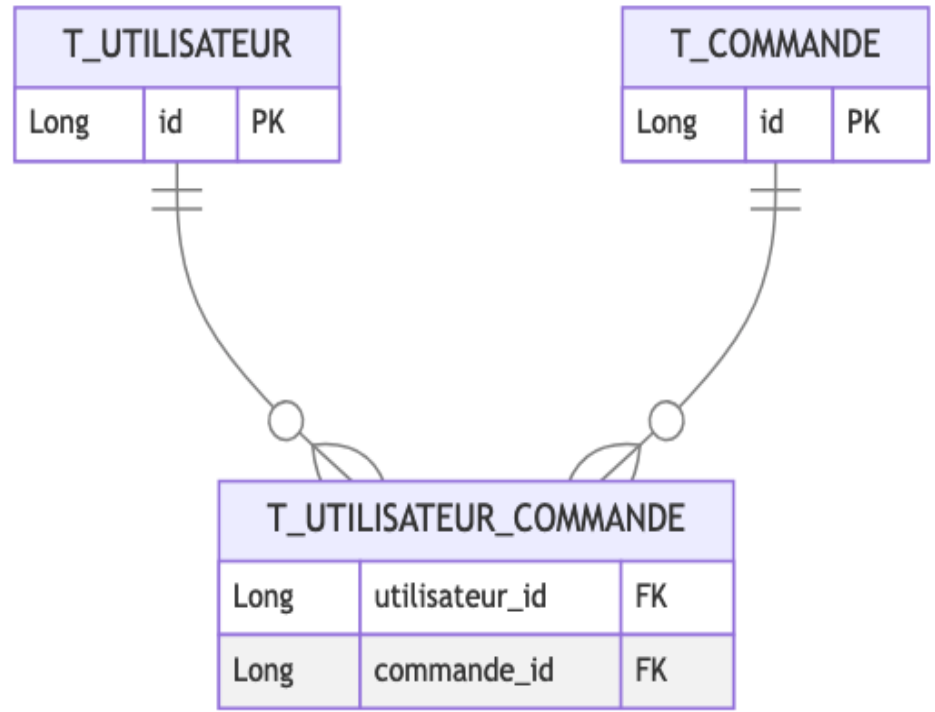
```java
@Entity public class Utilisateur {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToMany( cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Commande> commandes = new ArrayList<>();
}
```

```java
@Entity public class Commande {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

Relation 1:n unidirectionnelle (3t)

Mais si nous exécutons le code suivant, nous n'aurons pas uniquement que deux tables (dont `t_commande` avec la FK) mais 3 tables :

- `t_utilisateur`
- `t_commande`
- `t_utilisateur_commande`



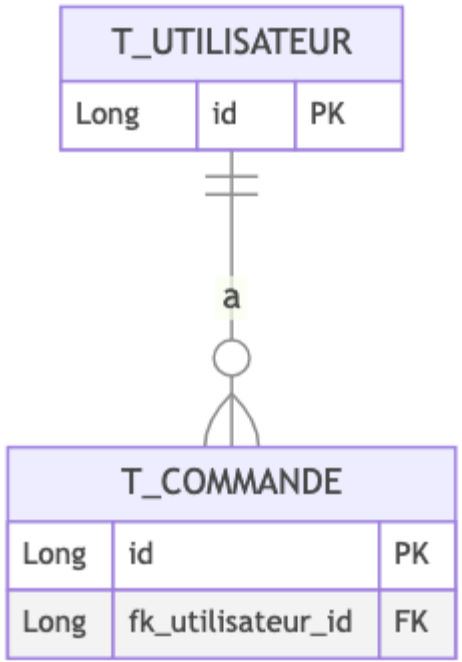
Relation 1:n unidirectionnelle (2t)

Pour résoudre le problème de la table de jointure supplémentaire , il suffit d'ajouter la colonne `@JoinColumn`

```
@Entity public class Utilisateur {
    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name ="fk_utilisateur_id") // <-- ICI
    private List<Commande> commandes = new ArrayList<>();
}
```

```
@Entity public class Commande {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
}
```

Relation 1:n unidirectionnelle (2t)



Nous avons **deux relations unidirectionnelles** != bidirectionnelle

Relation 1:n unidirectionnelle (2t)

Attention : problème de performance

```
Utilisateur utilisateur = new Utilisateur("Adrien");
utilisateur.setCommandes().add(new Commande("Premier Commentaire"));
```

```
insert into utilisateur (nom, id)
values ('Adrien', 1)

-- Création des commentaires
insert into commentaire (desc, id)
values (1, 'Premier Commentaire', 1)

-- Mise à jour des commentaires pour rajouter une valeur dans fk_utilisateur_id
update commentaire set fk_utilisateur_id = 1 where id = 1
```

Relation 1:n unidirectionnelle (2t)

- Problème de performance : L'insertion d'un commentaire se fait en deux étapes.

```
-- 1. Insertion
insert into commentaire (desc, id) values (1, 'Premier Commentaire', 1)

-- 2. Mise à jour des commentaires pour rajouter une valeur dans fk_utilisateur_id
update commentaire set fk_utilisateur_id = 1 where id = 1
```

Souhait

```
insert into commentaire (fk_utilisateur_id, desc, id)
values (1, 'Premier Commentaire', 1)
```

Relation 1:n bidirectionnelle

- Utilisation du `mappedBy`, qui nous permet de dire qui est le owner (i.e quelle classe **ne** contient **pas** la FK)
- Pour assurer la synchronisation bidirectionnelle nous avons ajouté les méthodes `addCommande` et `removeCommande`

Relation 1:n bidirectionnelle

```
@Entity public class Utilisateur {
    @OneToMany(
        mappedBy = "utilisateur", // FK pas dans utilisateur
        cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Commande> commandes = new ArrayList<>();

    public void addCommande(Commande commande) {
        commandes.add(commande);
        commande.setUtilisateur(this);
    }

    public void removeCommande(Commande commande) {
        commandes.remove(commande);
        commande.setUtilisateur(null);
    }
}
```

Relation 1:n bidirectionnelle

```java
@Entity
public class Commande {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY) // par défaut EAGER => moins performant
    @JoinColumn(name = "fk_utilisateur_id") // Optionnel : préciser le nom de la colonne
    private Utilisateur utilisateur;
}
```

- `@OneToMany` avec `mappedBy` + `@ManyToOne` = relation bidirectionnelle

Relation 1:n bidirectionnelle

Si on oublie le `mappedBy` alors :

- Une colonne (`fk_utilisateur_id`) pour la relation @ManyToOne définie dans Commande.
- Une table jointe `utilisateur_commande` avec les colonnes `utilisateur_id` et `commande_id` (du côté `@OneToMany`)

Relation 1:n bidirectionnelle

- TOP performance

```
Utilisateur utilisateur = new Utilisateur("Adrien");
Commande commande1 = new Commande("Premier Commentaire");
utilisateur.addCommande(commande1); // utilisation de la méthode
```

```
insert into utilisateur (nom, id)
values ('Adrien', 1)

-- Insertion directement avec la fk_utilisateur_id
insert into commentaire (fk_utilisateur_id, desc, id)
values (1, 'Premier Commentaire', 1)
```

Relation n:m

Relation n:m : unidirectionnelle

- Il n'y a qu'une seule manière de réaliser une relation de type « Many-To-Many » en base de données : il faut impérativement passer par une table d'association.

```
@Entity public  class Musicien  implements Serializable {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToMany
    private Collection<Instrument> instruments ;
}
```

```
@Entity public  class Instrument  implements Serializable {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
}
```

Relation n:m : bidirectionnelle

```
@Entity public  class Musicien  implements Serializable {
    @ManyToMany
    private Collection<Instrument> instruments ;
}
```

```
@Entity public  class Instrument  implements Serializable {
    @ManyToMany(mappedBy="instruments")
    private Collection<Musicien> musiciens ;
```

Relation n:m

On peut également préciser le nom de la table de jointure et des colonnes

```
// class Musicien
@JoinTable(name = "intrument_musicien",
    joinColumns = @JoinColumn(name = "fk_ins_id"),
    inverseJoinColumns = @JoinColumn(name = "fk_mus_id")
)
private Collection<Instrument> instruments
```

```
// class Instrument
@ManyToMany(mappedBy="instruments")
private Collection<Musicien> musiciens ;
```

Si l'on ne met pas `mappedBy` , alors JPA créera une seconde table de jointure.