Applications orientées données (NSY135)

8 – JPA : le mapping

Auteurs: Raphaël Fournier-S'niehotta et Philippe Rigaux
(philippe.rigaux@cnam.fr,fournier@cnam.fr)

Département d'informatique
Conservatoire National des Arts & Métiers, Paris, France

Plan du cours

Introduction

- Nous en venons maintenant aux associations de type "plusieurs-à-plusieurs", dont deux représentants apparaissent dans le modèle de notre base **webscope** :
  - un film a **plusieurs** acteurs; un acteur joue dans **plusieurs** films;
  - un internaute note **plusieurs** films; un film est noté par **plusieurs** internautes.
- Quand on représente une association plusieurs-plusieurs en relationnel, l'association devient une table dont la clé est la concaténation des clés des deux entités de l'association.
- C'est bien ce qui a été fait pour cette base : une table **Role** représente la première association avec une clé composée **(id_film, id_acteur)** et une table **Notation** représente la seconde avec une clé **(id_film, email)**.
- Et il faut noter de plus que chaque partie de la clé est elle-même une clé étrangère.
- On pourrait généraliser la règle à des associations ternaires, et plus.
- Les clés deviendraient très compliquées et le tout deviendrait vraiment difficile à interpréter.
- La recommandation dans ce cas est de **promouvoir** l'association en entité, en lui donnant un identifiant qui lui est propre.

Introduction (suite)

- Même pour les associations binaires, il peut être justicieux de transformer toutes les associations plusieurs-plusieurs en entité, avec deux associations un-plusieurs vers les entités originales.
- On aurait donc une entité **Role** et une entité **Notation** (chacune avec un identifiant).
- Cela simplifierait le **mapping** JPA qui, comme on va le voir, est un peu compliqué.
- On trouve de nombreuses bases (dont la nôtre) incluant la représentation d'associations plusieurs-plusieurs avec clé composée. Il faut donc savoir les gérer.
- Deux cas se présentent en fait :
  - l'association n'est porteuse d'aucun attribut, et c'est simple;
  - l'association porte des attributs, et ça se complique un peu.
- Nos associations sont porteuses d'attributs, mais nous allons quand même traiter les deux cas par souci de complétude.

Association sans attribut

- Dans notre schéma, le nom du rôle d'un acteur dans un film est représenté dans l'association (il ne peut pas l'être ailleurs).
- Supposons pour l'instant que nous décidions de sacrifier cette information, ce qui nous ramène au cas d'une association sans attribut.
- La modélisation JPA/Hibernate est alors presque semblable à celle des associations un-plusieurs.
- Voici ce que l'on représente dans la classe **Film**.

```
1   @ManyToMany()
2   @JoinTable(name = "Role", joinColumns = @JoinColumn(name = "id_film"),
3                 inverseJoinColumns = @JoinColumn(name = "id_acteur"))
4   Set<Artiste> acteurs = new HashSet<Artiste>();
5   public Set<Artiste> getActeurs() {
6     return acteurs;
7   }
```

Association sans attribut

- C'est ce côté qui est "responsable" du **mapping** avec la base de données.
- On indique donc une association **@ManyToMany**, avec une seconde annotation **JoinTable** qui décrit tout simplement la table représentant l'association, **joinColumn** étant la clé étrangère pour la classe courante (**Film**) et **inverseJoinColumn** la clé étrangère vers la table représentant l'autre entité, **Artiste**.
- De l'autre côté, c'est plus simple encore puisqu'on ne répète pas la description du **mapping**.
- On indique avec **mappedBy** qu'elle peut être trouvée dans la classe **Film**.
- Ce qui donne, pour la filmographie d'un artiste :

```
1   @ManyToMany(mappedBy = "acteurs")
2   Set<Film> filmo;
3   public Set<Film> getFilmo() {
4     return filmo;
5   }
```

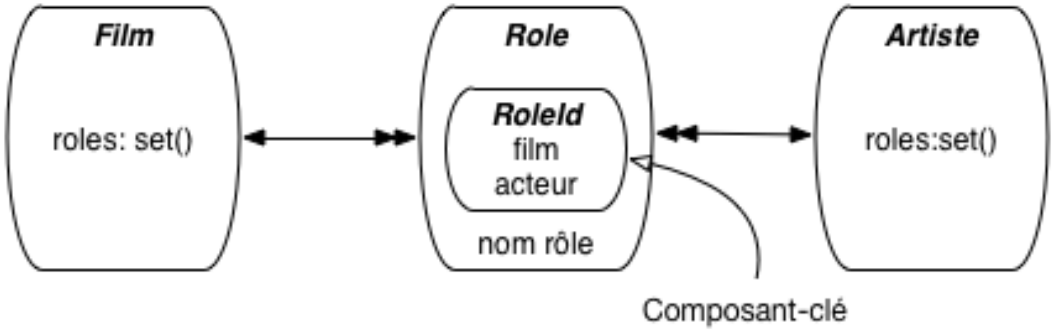
- il est maintenant possible de naviguer d'un film vers ses acteurs et réciproquement.
- L'interprétation de **mappedBy** est la même que dans le cas des associations un-plusieurs.

Association avec attribut

- Avec notre base nous aimerions bien accéder au nom du rôle, ce qui nécessite un accès à la table **Role** représentant l'association.
- Nous allons représenter dans notre modèle JPA cette association comme une **@Entity**, avec des associations un à plusieurs vers, respectivement, **Film** et **Artiste**, et le tour est joué.
- Oui, mais... nous voici confrontés à un problème que nous n'avons pas encore rencontré : la clé de **Role** est composée de **deux** attributs.
- Il va donc falloir tout d'abord apprendre à gérer ce cas.
- Nous avons fait déjà un premier pas dans ce sens en étudiant les **composants** (et les adresses).
- Une clé en Hibernate se représente par un objet.
- Quand cet objet est une valeur d'une classe de base (**Integer**, **String**), tout va bien.
- Sinon, il faut définir cette classe avec les attributs constituant la clé.
- En introduisant un nouveau niveau de granularité dans la représentation d'une entité, on obtient bien ce que nous avons appelé précédemment **un composant**.

Gestion d'un identifiant composé

- La classe **Role** est une entité composée d'une clé et d'un attribut, le nom du rôle.
- La clé est une instance d'une classe **RoleId** constituée de l'identifiant du film et de l'identifiant de l'acteur.



- ce n'est pas une entité, mais un composant annoté avec **@Embeddable**
- Pour le reste on indique les associations **@ManyToOne** de manière standard.

♠ une classe dont les instances vont servir d'identifiants doit hériter de **serializable**, la raison - technique - étant que des structures de **cache** dans Hibernate doivent être sérialisées dans la session, et que ces structures indexent les objets par la clé.

En pratique : RoleId

```java
package modeles.webscope;
import javax.persistence.*;

@Embeddable
public class RoleId implements java.io.Serializable {

  private static final long serialVersionUID = 1L;

  @ManyToOne
  @JoinColumn(name = "id_acteur")
  private Artiste acteur;
  public Artiste getActeur() {
    return acteur;
  }

  public void setActeur(Artiste a) {
    this.acteur = a;
  }

  @ManyToOne
  @JoinColumn(name = "id_film")
  private Film film;
  public Film getFilm() {
    return film;
  }

  public void setFilm(Film f) {
    this.film = f;
  }
}
```

En pratique : Role

```java
package modeles.webscope;

import javax.persistence.*;

@Entity
public class Role {

  @Id
  RoleId pk;
  public RoleId getPk() {
    return pk;
  }

  public void setPk(RoleId pk) {
    this.pk = pk;
  }

  @Column(name="nom_role")
  private String nom;
  public void setNom(String n) {nom= n;}
  public String getNom() {return nom;}
}
```

- On indique donc que l'identifiant est une instance de **RoleId**.
- On la nomme **pk** pour **primary key**, mais le nom n'a pas d'importance.
- La classe comprend de plus les attributs de l'association.

Role (suite)

- On peut donc maintenant accéder à une instance **r** de rôle, afficher son nom avec **r.nom** et même accéder au film et à l'acteur avec, respectivement, **r.pk.film** et **r.pk.acteur**.
- Comme cette dernière notation peut paraître étrange, on peut ajouter le code suivant dans la classe **Role** qui implante un racourci vers le film et l'artiste.

```
1   public Film getFilm() {
2     return getPk().getFilm();
3   }
4
5   public void setFilm(Film film) {
6     getPk().setFilm(film);
7   }
8
9   public Artiste getActeur() {
10    return getPk().getActeur();
11  }
12
13  public void setActeur(Artiste acteur) {
14    getPk().setActeur(acteur);
15  }
```

- Maintenant, si **r** est une instance de **Role**, **r.film** et **r.acteur** désignent respectivement le film et l'acteur.

De l'autre côté de l'association : Film

```java
@OneToMany(mappedBy = "pk.film")
private Set<Role> roles = new HashSet<Role>();
public Set<Role> getRoles() {
  return this.roles;
}
public void setRoles(Set<Role> r) {
  this.roles = r;
}
```

- Seule subitilité : le **mappedBy** indique un **chemin** qui part de **Role**, passe par la propriété **pk** de **Role**, et arrive à la propriété **film** de **Role.pk**.
- C'est là que l'on va trouver la définition du **mapping** avec la base, autrement le nom de la clé étrangère qui référence un film.
- Même chose du côté des artistes :

```java
 @OneToMany(mappedBy = "pk.acteur")
private Set<Role> roles = new HashSet<Role>();
public Set<Role> getRoles() {
  return this.roles;
}
public void setRoles(Set<Role> r) {
  this.roles = r;
}
```

Plan du cours

Résumé

- On a vu en détail le **mapping** ORM pour la plupart des bases de données,
- une exception (rare) étant la représentation de **l'héritage**, pour lequel les bases relationnelles ne sont pas adaptées, mais que l'on va trouver potentiellement dans la modélisation objet de l'application.
- Il faut alors savoir produire une structure de base de données adptée, et la **mapper** vers les classes de l'application. C'est ce que nous verrons dans le prochain chapitre.

- Il reste encore des choses importantes à couvrir pour être complets sur le sujet. Entre autres :
  - les méthodes **equals()** et **hashCode()** sont importantes pour certaines classes; il faudrait les implanter;
  - des options d'annotation, dont celles qui indiquent comment **propager** la persistence en **cascade**, méritent d'être présentées.
- à titre d'exercice : prendre un schéma de base de données connu et produire le mapping adapté