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

Introduction

- Ce chapitre est consacré à une étude plus systématique de JPA/Hibernate
- on commence par la définition du modèle de données, ou plus exactement de l'association (**mapping**) entre la base de données relationnelle et le modèle objet java.
- le but est toujours de transformer **automatiquement** une base relationnelle en graphe d'objets java.
- Cette transformation est effectuée par Hibernate sur des données extraites de la base avec des requêtes SQL.
- On travaille ici avec une base pré-existante
- Il est aussi possible de définir un modèle objet et de laisser Hibernate générer le schéma relationnel correspondant.

Introduction (suite)

- Le but de ce chapitre est principalement de finaliser notre modèle java pour la base **webscope**, ce qui couvrira les options les plus courantes du **mapping** O/R.
- Il existe plusieurs méthodes pour définir un modèle O/R.
  - par un fichier de configuration XML;
  - par des annotations directement intégrées au code java.
- L'option "annotation" présente de nombreux avantages, pour la clarté, les manipulations de fichier, la concision. C'est donc celle que l'on présente ici
- Reportez-vous à la documentation Hibernate si vous voulez inspectez un fichier de configuration XML typique.
  - ♠ Attention, quand vous regardez les (innombrables) documentations sur le Web, à la date de publication; les outils évoluent rapidement, et beaucoup de tutoriaux sont obsolètes
- Pour utiliser les annotations JPA, il faut inclure le **package javax.persistence.***.
- On va suivre JPA le plus possible plutôt que la syntaxe spécifique à Hibernate, la tendance de toute façon étant à la convergence vers la norme.

Plan du cours

Entités et composants

- Une entité, déclarée par l'annotation **@Entity** définit une classe Java comme étant **persistante** et donc associée à une table dans la base de données.
- Cette classe doit être implantée selon les normes des beans, avec des propriétés non publiques (**private** est sans doute le bon choix)
- accesseurs avec **set** et **get**, nommés selon les conventions habituelles.
- une entité n'a pas strictement besoin d'hériter de **Serializable**, sauf si elle doit être transmise à un service distant par le réseau. (option non utilisée ici)
- Par défaut, une entité est associée à la table portant le même nom que la classe.
- Il est possible d'indiquer le nom de la table par une annotation **@Table** :

```
@Entity
@Table(name="Film")
public class Film {
  ...
}
```

- NB : Il existe de nombreuses options JPA, et Hibernate, nous nous limitons à l'essentiel pour comprendre les mécanismes.

Constructeur

- La norme JPA requiert créer un constructeur vide pour chaque entité.
- Le **framework** peut en effet avoir à instancier une entité avec un appel à **Constructor.newInstance()** et le constructeur correspondant doit exister.
- La norme JPA indique que ce constructeur doit être public ou protégé :

```
public Film() {}
```

- Inconvénient : crée un objet dans un état invalide (aucune propriété n'a de valeur).
- Pas de problème quand c'est Hibernate qui utilise ce constructeur (puisqu'il va affecter les propriétés en fonction des valeurs dans la base)
- Mais du point de vue de l'application c'est une source potentielle d'ennuis.
- Avec Hibernate, le constructeur vide peut être déclaré **private** (mais ce n'est pas la recommandation JPA).
- Nous nous soucions essentiellement pour l'instant d'opération de **lecture**, qui suffisent à illustrer et valider la modélisation du **mapping**.

Identifiant d'une entité

- Toute entité doit avoir une propriété déclarée comme étant l'identifiant de la ligne dans la table correspondante.
- Il est beaucoup plus facile de gérer une clé constituée d'une seule valeur qu'une clé composée de plusieurs.
- Les bases de données conçues selon des principes de **clé artificielle** ou **surrogate key** (produite à partir d'une séquence) sont de loin la meilleure solution.
- On peut être confronté à une base qui n'a pas été conçue sur ce principe, auquel cas JPA/Hibernate fournit des méthodes permettant de faire face à la situation, mais c'est plus compliqué. cf fin de ce chapitre.
- L'identifiant est indiqué avec l'annotation **@Id**.

Identifiant (suite)

- Pour les valeurs d'identifiant auto-générées (avec une séquence ou autre) on ajoute une annotation **@GeneratedValue**.
- Hibernate choisit alors automatiquement la méthode de production de l'identifiant unique en fonction du système relationnel utilisé.
- Par exemple, pour **Film** ou **Artiste** dont l'identifiant est auto-généré par MySQL (grâce à l'option **AUTO-INCREMENT**), il suffit d'indiquer :

```
@Id
@GeneratedValue
private Integer id;
private void setId(Integer i) { id = i; }
public Integer getId() { return id; }
```

- **@GeneratedValue** est en fait une abréviation pour :

```
@GeneratedValue(strategy = GenerationType.AUTO)
```

- On indique ici la stratégie de génération de l'identifiant unique, soit en laissant Hibernate faire le meilleur choix (plus simple) avec **AUTO**, soit en indiquant d'autres options (cf doc)

Identifiant : accesseurs

- Faut-il fournir des accesseurs **setId()** et **getId()**?
- Conceptuellement, l'id est utilisé pour lier un objet à la base de données et ne joue aucun rôle dans l'application.
- Il n'a donc pas de raison d'apparaître publiquement. D'où l'absence de méthode pour le récupérer dans la plupart des objets métiers :
  - fournir une méthode **publique setId()** pour un identifiant auto-généré ne semble pas une bonne idée, car dans ce cas l'application pourrait affecter un identifiant en conflit avec la méthode de génération; il est donc préférable de créer une méthode privée;
  - fournir une méthode **getId()** n'est pas nécessaire, sauf si l'application souhaite inspecter la valeur de l'identifiant en base de données, ce qui est de fait souvent utile.
- En résumé, la méthode **setId()** devrait être **private**. Notez qu'Hibernate utilise l'API **Reflection** de java pour accéder aux propriétés des objets, et n'a donc pas besoin de méthodes publiques.
- Avec Hibernate, l'existence de **setId()** ne semble même pas nécessaire. Il semble que JPA requiert des accesseurs pour chaque propriété mappée, donc autant respecter cette règle pour un maximum de compatibilité.

Colonnes

- Par défaut, toutes les propriétés non-statiques d'une classe-entité sont considérées comme devant être stockées dans la base.
- Pour indiquer des options (et aussi pour des raisons de clarté à la lecture du code) on utilise le plus souvent l'annotation **@Column**, comme par exemple :

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

- Cette annotation est utile pour indiquer le nom de la colonne dans la table, quand cette dernière est différente du nom de la propriété en java.
- Dans notre cas nous utilisons des règles de nommage différentes en java et dans la base de données pour les noms composés de plusieurs mots.

Colonnes (suite)

- **@Column** permet alors d'établir la correspondance :

```
@Column(name="annee_naissance")
private Integer anneeNaissance;
public void setAnneeNaissance(Integer a) {anneeNaissance = a;}
public Integer getAnneeNaissance() {return anneeNaissance;}
```

- Principaux attributs pour **@Column** :
  - **name** indique le nom de la colonne dans la table;
  - **length** indique la taille maximale de la valeur de la propriété;
  - **nullable** (avec les valeurs **false** ou **true**) indique si la colonne accepte ou non des valeurs à **NULL** (au sens "base de données" du terme : une valeur à **NULL** est une absence de valeur);
  - **unique** indique que la valeur de la colonne est unique.
- Un objet métier peut très bien avoir des propriétés que l'on ne souhaite pas rendre persistantes dans la base.
- Il faut alors impérativement les marquer avec l'annotation **@Transient**.

Composants

- Une **entité** existe par elle-même indépendamment de toute autre entité, et peut être rendue persistante par insertion dans la base de données, avec un identifiant propre.
- Un **composant**, au contraire, est un objet sans identifiant, qui ne peut être persistant que par rattachement (direct ou transitif) à une entité.
- La notion de composant résulte du constat qu'une ligne dans une base de données peut **parfois** être décomposée en plusieurs sous-ensemble dotés chacun d'une logique autonome.
- Cette décomposition mène à une granularité fine de la représentation objet, dans laquelle on associe **plusieurs** objets à **une** ligne de la table.

Composants (suite)

- On peut sans doute modéliser une application sans recourir aux composants, au moins dans la définition stricte ci-dessus.
- Ils sont cependant également utilisés dans Hibernate pour gérer d'autres situations, et notamment les clés composées de plusieurs attributs, comme nous le verrons plus loin.
- Regardons donc un exemple concret, celui (beaucoup utilisé) de la représentation des **adresses**. Prenons donc nos internautes, et ajoutons à la table quelques champs (au minimum) pour représenter leur adresse.

```
ALTER TABLE Internaute ADD adresse TEXT,
    ADD code_postal VARCHAR(10), ADD ville VARCHAR(100);
```

- Utilisez phpMyAdmin pour insérer quelques adresses aux internautes existant dans la base, et passons maintenant à la modélisation java.

Composants (suite)

- On va considérer ici que l'adresse est un **composant** de la représentation d'un internaute qui dispose d'une unité propre et distinguable du reste de la table.
- Cela peut se justifier, par exemple, par la nécessité de contrôler qu'un code postal est correct, une logique applicative qui n'est pas vraiment pertinente pour l'entité **Internaute**.
- Un autre argument est qu'on peut réutiliser la définition du composant pour d'autres entités, comme par exemple **Société**.

Représentation d'une adresse

- On représente une adresse comme un composant, désigné par le mot-clé **Embeddable** en JPA.

```
1   package modeles.webscope;
2
3   import javax.persistence.*;
4
5   @Embeddable
6   public class Adresse {
7       String adresse;
8     public void setAdresse(String v) {adresse = v;}
9     public String getAdresse() {return adresse;}
10
11    @Column(name="code_postal")
12  String codePostal;
13    public void setCodePostal(String v) {codePostal = v;}
14    public String getCodePostal() {return codePostal;}
15
16    String ville;
17    public void setVille(String v) {ville = v;}
18    public String getVille() {return ville;}
19  }
```

- La principale différence visible avec une entité est qu'un composant n'a pas d'identifiant (marqué **@Id**)

Composants : rattachement à une entité

- Il faut le rattacher à une entité, ici **Internaute**, pour le sauvegarder dans la base

```
1   package modeles.webscope;
2
3   import javax.persistence.*;
4
5   @Entity
6   public class Internaute {
7
8   @Id
9   private String email;
10  public void setEmail(String e) {email = e;}
11
12  @Column
13  private String nom;
14  public void setNom(String n) {nom = n;}
15  public String getNom() {return nom;}
16
17  @Column
18  private String prenom;
19  public void setPrenom(String p) {prenom = p;}
20  public String getPrenom() { return prenom;}
21
22  @Embedded
23  private Adresse adresse;
24  public void setAdresse(Adresse a) {adresse = a;}
25  public Adresse getAdresse() {return adresse;}
26  }
```

Rattachement à une entité (suite)

- Au lieu de l'annotation **Column**, on utilise **@Embedded** et le tour est joué.
- La propriété **adresse** devient, comme **nom** et **prénom**, une propriété **persistante** de l'entité.
- On remarque que la classe **Adresse** contient des annotations qui le **mappent** vers la table, notamment avec les noms de colonne.
- Mais que se passe-t-il alors si on place deux composants du même type dans une entité?
- Par exemple, si on veut avoir une adresse personnelle et une adresse professionnelle pour un internaute?
- On **mapperait** dans ce cas deux propriétés distinctes vers la **même** colonne. Pour illustrer le problème (et la solution), commençons par modifier la table.

```
ALTER TABLE Internaute ADD adresse_pro TEXT,
      ADD code_postal_pro VARCHAR(10), ADD ville_pro VARCHAR(100);
```

Rattachement à une entité (suite)

- On indique alors, **dans le composé** (l'entité), le **mapping** entre la table de l'entité et le nouveau composant, autrement dit les colonnes de la table qui doivent stocker les propriétés du composant.
- C'est une **surcharge** (**override** en anglais), d'où la syntaxe suivante en JPA, à ajouter à la classe **Internaute**.

```
1   @Embedded
2   @AttributeOverrides( {
3         @AttributeOverride(name="adresse", column = @Column(name="adresse_pro") ),
4         @AttributeOverride(name="codePostal", column = @Column(name="code_postal_pro") ),
5         @AttributeOverride(name="ville", column = @Column(name="ville_pro") )
6      }
7   )
8   private Adresse adressePro;
9   public void setAdressePro(Adresse a) {adressePro = a;}
10  public Adresse getAdressePro() {return adressePro;}
```

- L'attribut **codePostal** du composant **adressePro** sera donc par exemple stocké dans la colonne **code_postal_pro**.