Eclipse – JPA – Hibernate

Contenu

- Implémentation de l'héritage
- Implémentation des associations
- JPQL

L'héritage

Pour réaliser l'implémentation d'une relation d'héritage dans une base de données relationnelle, la spécification JPA propose trois stratégies :

Une seule table pour stocker les attributs de toutes les classes d'une arborescence d'héritage (stratégie par défaut). Un champ (le champ discriminant) permet de déterminer le type de la classe (le nom par défaut de ce champ est DTYPE de type varchar et peut être défini par @DiscriminatorColumn (l'attribut DiscriminatorType définit le type de ce champ), la valeur par défaut est le nom de l'entité et cette valeur peut être personnalisée à l'aide de l'annotation @DiscriminatorValue

Remarque:

Les attributs des entités filles doivent être nullable

Jointure des sous classes : Une table par classe de la hiérarchie d'héritage (que la classe de base soit abstraite ou concrète)

Inconvénients : performance

Une table pour chaque classe concrète. (Le support de cette stratégie est optionnel dans la spécification JPA 2.0), les propriétés de la classe de base sont dupliquées dans les classes filles.

Inconvénients : les appels polymorphiques sont plus coûteux que dans les deux autres stratégies.

Les annotations @AttributeOverride et @AttributeOverrides permettent de renommer les attributs dupliqués dans les classes filles.

Atelier

1. Créez les sous classes ClientImportant et ClientRegulier de la classe de base Client

@Entity
@Inheritance
public class Client {
        @Id
        @GeneratedValue
        private long id;
        private String nom;
	@Entity
	public class
	ClientImportant
	extends Client {
	
	        private double
	solde;
		@Entity
		public class
		ClientRegulier extends
		Client {
		
		        private String
		adresse;

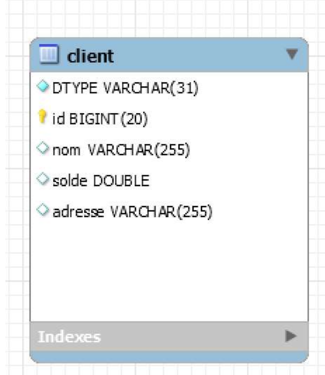
… }	… }	… }

2. Dans le fichier persistence.xml donnez la valeur « drop-and-create » à la propriété "javax.persistence.schema-generation.database.action"

```
<property name="javax.persistence.schema-generation.database.action"
value="drop-and-create"/>
```

3. Dans la méthode main ajouter un client, un client important et un client régulier

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Client cl0=new Client();
cl0.setNom("un Client");
em.persist(cl0);
ClientImportant cl=new ClientImportant();
cl.setNom("Client important 1");
cl.setSolde(3999);
em.persist(cl);
ClientRegulier cl2=new ClientRegulier();
cl2.setNom("Client régulier 1");
cl2.setAdresse("1 10 rue 2");
```




```
em.persist(cl2);
tx.commit();
em.close();
```

4. Exécutez le projet

Après exécution du projet une seule table est crée dont le nom est celui de la classe de base, et qui contient les attributs des trois classes

DTYPE	id	nom	solde	adresse
Client	1	un client	NULL	NULL
ClientImportant	2	Client important 1	3999	NULL
ClientRegulier	3	Client régulier 1	NULL	1 10 rue 2
NULL	NULL	NULL	NULL	NULL

Le champ DTYPE contient le type de client (la valeur de ce champ contient par défaut le nom de la classe).

Le résultat obtenu est équivalent à l'utilisation des annotations JPA suivantes :

<table>
<tr>
<td><pre>@Entity
@Inheritance(strategy
=
InheritanceType.SINGLE
_TABLE)
@DiscriminatorColumn(n
ame = "DTYPE",
discriminatorType =
DiscriminatorType.STRI
NG)
@DiscriminatorValue(va
lue = "Client")
public class Client {
…
}</pre></td>
<td><pre>@Entity
@DiscriminatorValue(value="ClientI
mportant")
public class ClientImportant
extends Client {

…

}</pre></td>
<td><pre>@Entity
@DiscriminatorValue
(value =
"ClientRegulier")
public class
ClientRegulier
extends Client {

…

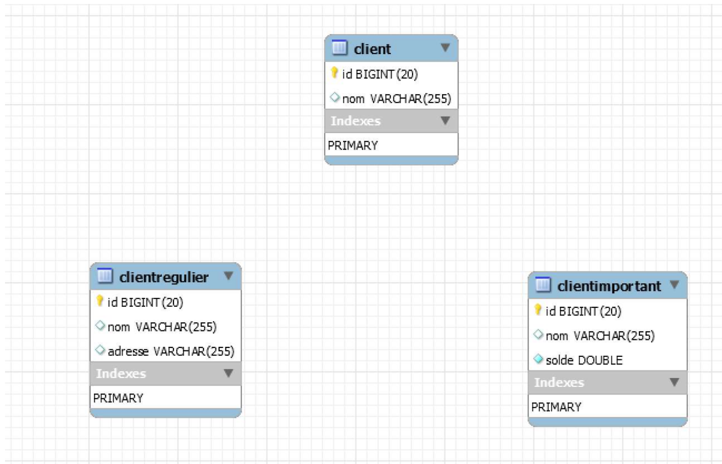
}</pre></td>
</tr>
</table>

Remarque : Si des erreurs de génération se produisent lors de l'exécution du projet (étapes 5 et 6), soit vous réexécutez le projet à nouveau soit vous supprimez manuellement les tables de la base de données

5. Modifiez le projet pour avoir une table pour chaque classe

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

Tables générées

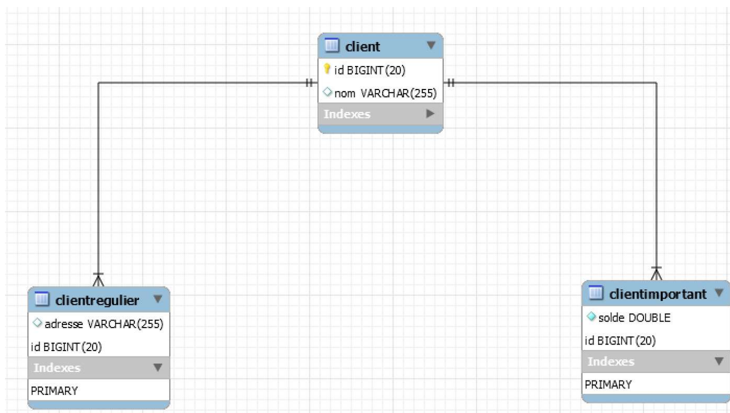


Client	Client Important	Client régulier
id nom 1 un client	id nom 2 Client important 1	id nom adress 3 Client régulier 1 1 10 ru

6. 5 Modifiez le projet pour utliser la stratégie d'héritage « Joined »

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Client {…
```

Tables générées



Client	Client Important	Client régulier

id	nom
1	un client
2	Client important 1
3	Client régulier 1

solde	id
3999	2

adresse	id
1 10 rue 2	3

Les associations

La spécification supporte trois types d'association :

- One to one
- Many to one / One to Many
- Many to Many

Les trois types d'association peuvent être unidirectionnelles (navigables dans un sens) ou bidirectionnelles (navigables dans les deux sens), la navigabilité impacte l'utilisation des classes dans le programme et non pas les tables générées dans la base de données.

Atelier

1. Créez les classes Categorie et Produit (Association : ManyToOne , La navigabilité : Produit → Categorie)

L'association ManyToOne est la plus utilisée dans la pratique, dans la classe Produit on ajoutera un attribut de type Categorie.

Categorie

```
@Entity
public class Categorie {
	@Id
	@GeneratedValue(strateg
y = GenerationType.IDENTITY)
	private long catId;
	private String libelle;


…
}
```

Produit

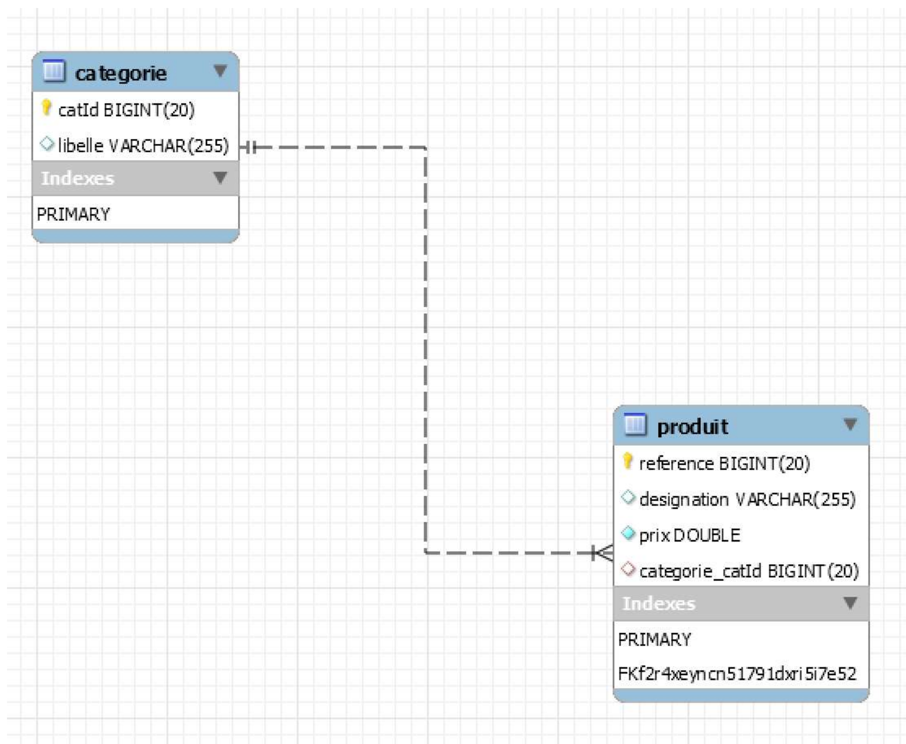
```
@Entity
public class Produit {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY
)
private long reference;
private String designation;
private double prix;
@ManyToOne
private Categorie categorie;

…
}
```

2. Dans la méthode main ajoutez une catégorie et un produit

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu");
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();
        // Ajouter une catégorie
        Categorie c = new Categorie();
        c.setLibelle("PC");
        em.persist(c);
        // Ajout d'un produit
        Produit p = new Produit();
        p.setDesignation("Asus ROG 551");
        p.setPrix(15600);
        p.setCategorie(c);
        em.persist(p);
        tx.commit();
        em.close();
```

3. Exécuter le projet

Tables générées



4. Ajouter la classe Catalogue (association : OneToMany, Navigabilité : Catalogue → Produit)

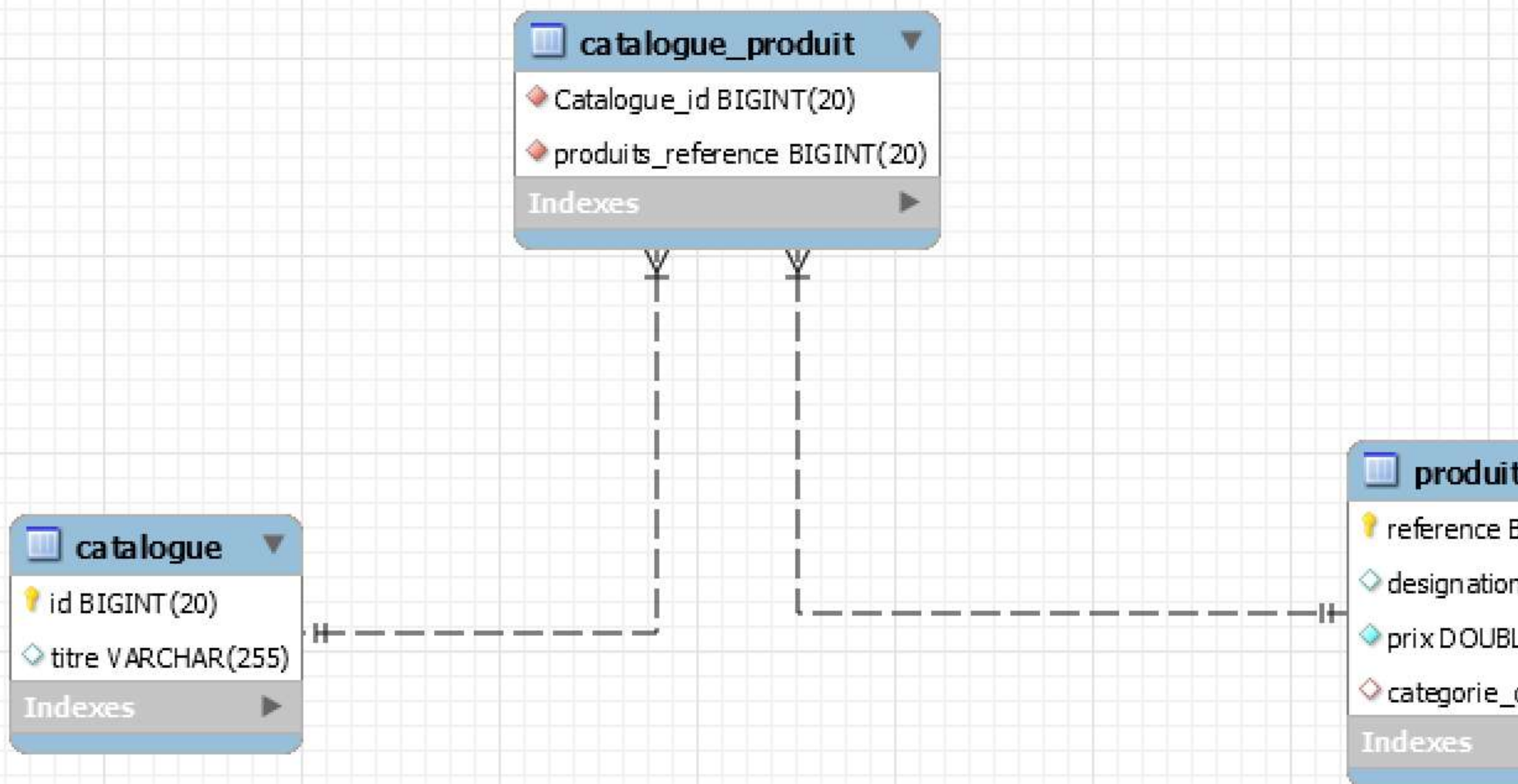
<table>
<tr><th>Catalogue</th><th>Main</th></tr>
<tr><td>

```java
@Entity
public class Catalogue {

	@Id
	@GeneratedValue(strategy =
GenerationType.IDENTITY)
	private long id;
	private String titre;
	@OneToMany
```

</td><td>

```java
//Ajout d'un catalogue
Catalogue cat=new Catalogue();
cat.setTitre("cat1");
cat.getProduits().add(p);
m.persist(cat);
```

</td></tr>
</table>




```
	private List<Produit> 
produits=new ArrayList<>();

	…
}
```

Tables générées

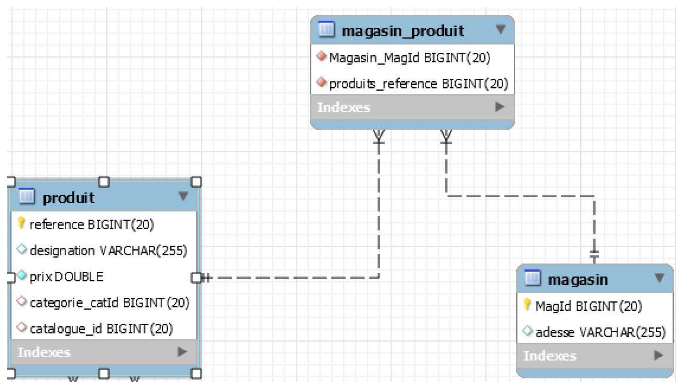
Par défaut l'annotation ManyToOne génère une table association (ce qui est souhaité dans notre exemple) , pour généré une clé étrangère dans la table Produit, il faut ajouter l'annotation JoinColumn pour définir le nom de la clé étrangère dans la table produit :

```
private String titre;
@OneToMany
@JoinColumn(name="catalogue_id")
private List<Produit> produits=new ArrayList<>();
```

5. Nous allons modéliser une association de type plusieurs à plusieurs bidirectionnelle entre Magasin et Produit (Navigabilité : Magasin ←→ Produit)

Magasin	Main

@Entity
public class Magasin {
@Id
private long MagId;
private String adesse;
@ManyToMany
private List<Produit> produits
= new ArrayList<>();
…
 	
}
	//Ajout d'un Magasin
	Magasin m=new Magasin();
	m.setMagId(1L);
	m.setAdesse("rue ure n 1");
	m.getProduits().add(p);
	em.persist(m);



Tables générées

6. Ajoutez la navigabilité Produit → Magasin

```
@ManyToMany(mappedBy = "produits")
private List<Magasin> magasins=new ArrayList<>();
```

Ce qui permettra d'accéder aux magasins à partir d'un produit p.getMagasins()

Requêtes JPQL

JPQL (Java Persistence Query Language) est un langage d'intérogation orienté objet similaire au langage SQL,

Structure d'une requête

SQL	JPQL
SELECT * FROM produit WHERE prix > 100 ORDER BY designation;	**SELECT p** **FROM** Produit p **WHERE p.prix** > 100 **ORDER BY** p.designation

Exemple

```
// Créer une requête
		String req = "Select p from Produit p where p.prix > 400";
		Query r = em.createQuery(req);
		// r.getResultList() pour les requêtes select;
		// r.executeUpdate(): pour les requêtes update, delete, insert;
		List<Produit> produits = (List<Produit>) r.getResultList();

		for (Produit p : produits) {
			System.out.println(p.getDesignation());
		}
```