



Dans cet article


Podcast AXOPEN



AXOPEN

Créati...





26:01

124

AXOPEN - Création...	124
AXOPEN - Concepti...	188
AXOPEN - Le merv...	258
AXOPEN - Spécial ...	223
AXOPEN - La perfo...	311
AXOPEN - L'intégr...	483
AXOPEN - NPM, Nu...	264

Latest tracks by [AXOPEN](#)

Cookie policy

Offres d'emploi

- Développeur applications web
- Développeur .NET / C# – Lyon
- Développeur PHP Symfony
- Développeur Full Stack – Lyon
- Développeur mobile Android – Lyon

Tags

- PLANISWARE (88)
- OPX2 (82)
- P5 (82)
- JBOSS-7 (27)
- JBOSS (24)
- JEE (21)
- OJS (21)
- PERFORMANCE (17)
- PLANISWARE-5 (16)
- HIBERNATE (14)
- JAVA (14)
- REPORTING (13)
- HIBERNATE-4 (12)
- SCRIPT (12)
- BI (11)
- CLIENT-LOURD (11)
- ERREUR (11)
- ARCHITECTURE (9)
- CACHE (8)
- CLIENT-LÉGER (8)
- JBOSS-EAP (8)
- MOBILE (8)
- ETL (7)
- JSF (7)
- REQUÊTE (7)
- VSPHERE (7)
- APACHE (6)
- INTRANET-SERVER (6)
- JAVA-8 (6)
- JBOSS-EAP-6.2 (6)

# Hibernate 4 – Héritage – Mapping et stratégies

Par Florent Tripier Le 14/03/2014

Héritage   HIBERNATE   HIBERNATE 4   JBOSS   JEE



Dans une base de données relationnelle, il est souvent intéressant de faire de l'héritage. Mais comment peut-on représenter cet héritage avec Hibernate 4 ? Plusieurs stratégies existent, qui correspondent chacune à une représentation différente dans le modèle de données.

## 1 – Contexte

Pour détailler les différents mappings proposés par Hibernate, on prendra l'exemple d'une entité « employe » dont héritent deux entités : « technicien » et « ingénieur ». Les classes correspondantes sont Employe, Technicien et Ingenieur. Le modèle théorique d'héritage peut être représenté comme ci-dessous :

## 2 – L'héritage au sens strict

### 2.1 – Principe

Le mode **JOINED** correspond au cas où super-entités et sous-entités sont chacunes représentées par une table en base de données, et sans réplication des champs communs. Ces champs sont uniquement portés par la super-table, et l'id reporté dans les sous-tables permet de joindre pour récupérer les données de ces colonnes communes. Dans notre exemple, cela signifie que les champs « nom » et « prenom » sont uniquement portés par la table « employe » et que dans les tables « technicien » et « ingénieur », le champ « id » constitue une clé étrangère vers « employe » pour récupérer les « nom » et « prenom » à l'aide d'une jointure.

Cela correspond donc au schéma théorique ci-dessus.

### 2.2 – Mapping

Pour mettre en place ce mode d'héritage, il suffit d'annoter la super-classe avec **@Inheritance** et le type **JOINED**. Cela définit le mode d'héritage pour toutes les sous-classes de cette super-classe.

```
1 @Entity
2 @Table(name = "employe")
```

```
3      @Inheritance(strategy = InheritanceType.JOINED)
4      public abstract class Employe implements Serializable {
5
6          private static final long serialVersionUID = 1L;
7
8          @Id
9          @GeneratedValue(strategy = GenerationType.IDENTITY)
10         @Column(name = "id")
11         private int id;
12         @Column(name = "nom")
13         private String nom;
14         @Column(name = "prenom")
15         private String prenom;
```

Comme ce mode d’héritage implique des jointures pour récupérer des instances des sous-classes, il faut préciser dans les classes filles sur quelle colonne s’effectuera la jointure.

```
1      @Entity
2      @Table(name = "ingenieur")
3      @PrimaryKeyJoinColumn(name = "id")
4      public class Ingenieur extends Employe {
5
6          private static final long serialVersionUID = 1L;
7
8          @Column(name = "statut")
9          private String statut;
10         @Column(name = "nb_projets")
11         private int nbProjets;
```

```
1      @Entity
2      @Table(name = "technicien")
3      @PrimaryKeyJoinColumn(name = "id")
4      public class Technicien extends Employe {
5
6          private static final long serialVersionUID = 1L;
7
8          @Column(name = "poste")
9          private String poste;
10         @Column(name = "niveau")
11         private int niveau;
```

Ainsi, si je crée par exemple un Technicien et que je l’enregistre en base de données avec la méthode persist de l’EntityManager, Hibernate créera bien une ligne dans la table « employe » et une ligne dans la table « technicien » avec le même id.

De plus, notons que l’on peut tout-à-fait utiliser le GenerationType.IDENTITY avec cette stratégie d’héritage (toutes les stratégies sont en fait supportées).

### 3 – Un pseudo-héritage : dupliquer les données pour éviter les jointures

#### 3.1 – Principe

Le mode **TABLE\_PER\_CLASS** correspond au cas où super-entités et sous-entités sont chacunes représentées par une table en base de données, et où chaque sous-entité réplique les champs de sa super-entité. Dans notre exemple, cela signifie que les tables « employe », « technicien » et « ingénieur » comportent toutes les champs « id », « nom » et « prenom ».

3.2 – Mapping

Pour mettre en place cette stratégie, il suffit d’annoter la super-classe avec **@Inheritance** et le type **TABLE\_PER\_CLASS**. Cela définit le mode d’héritage pour toutes les sous-classes de cette super-classe.

```
1      @Entity
2      @Table(name = "employe")
3      @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
4      public abstract class Employe implements Serializable {
5
6          private static final long serialVersionUID = 1L;
7
8          @Id
9          @Column(name = "id")
10         private int id;
11         @Column(name = "nom")
12         private String nom;
13         @Column(name = "prenom")
14         private String prenom;
```

Aucune autre information n’est requise pour mapper les sous-classes.

```
1      @Entity
2      @Table(name = "ingenieur")
3      public class Ingenieur extends Employe {
4
5          private static final long serialVersionUID = 1L;
6
7          @Column(name = "statut")
8          private String statut;
9          @Column(name = "nb_projets")
10         private int nbProjets;
```

```
1      @Entity
2      @Table(name = "technicien")
3      public class Technicien extends Employe {
4
5          private static final long serialVersionUID = 1L;
6
7          @Column(name = "poste")
8          private String poste;
9          @Column(name = "niveau")
10         private int niveau;
```

Ici, si je crée par exemple un Technicien et que je l’enregistre en base de données avec la méthode persist de l’EntityManager, Hibernate ne créera qu’une ligne dans la table

« technicien » et rien dans la table « employe ». Si je veux enregistrer aussi un Employe, je dois donc le faire explicitement.

De plus, notons qu’aucune stratégie de génération d’id n’est ici spécifiée : en particulier, les GenerationType AUTO et IDENTITY ne sont pas supportés.

**Remarque :** les annotations **@AttributeOverrides** et **@AttributeOverride** permettent de surcharger le mapping des champs déclarés la super-classe :

```
1      @Entity
2      @Table(name = "technicien")
3      @AttributeOverrides({
4          @AttributeOverride(name="nomDeFamille", column=@Column(name="nom")),
5          @AttributeOverride(name="prenomUsuel", column=@Column(name="prenom"))
6      })
7      public class Technicien extends Employe {
8
9          private static final long serialVersionUID = 1L;
10
11         @Column(name = "poste")
12         private String poste;
13         @Column(name = "niveau")
14         private int niveau;
```

4 – Une alternative intéressante : la table unique

4.1 – Principe

Le mode **SINGLE\_TABLE** correspond au cas où super-entités et sous-entités sont représentées par une seule table en tout et pour tout en base de données. Cette table contient donc tous les champs de la super-classe plus tous ceux de toutes les sous-classes. Ceci implique qu’il y ait des éléments NULL pour chaque tuple de cette table. De plus, une colonne sert de discriminant pour déterminer de quel type doit être tel ou tel tuple. Dans notre exemple, cela signifie qu’il n’y a qu’une table « employe » qui comporte les champs « id », « nom » et « prenom », mais aussi « poste », « niveau », « statut » et « nb\_projets », et enfin une colonne de discrimination (dont le nom par défaut est « DTYPE »).

4.2 – Mapping

Pour mettre en place ce mode d’héritage, il suffit d’annoter la super-classe avec **@Inheritance** et le type **SINGLE\_TABLE**. Cela définit le mode d’héritage pour toutes les sous-classes de cette super-classe.

```
1      @Entity
2      @Table(name = "employe")
3      @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
4      public abstract class Employe implements Serializable {
5
6          private static final long serialVersionUID = 1L;
7
8          @Id
9          @GeneratedValue(strategy = GenerationType.IDENTITY)
10         @Column(name = "id")
```

```
11     private int id;
12     @Column(name = "nom")
13     private String nom;
14     @Column(name = "prenom")
15     private String prenom;
```

Aucune autre information n’est requise pour mapper les sous-classes.

```
1     @Entity
2     @Table(name = "ingenieur")
3     public class Ingenieur extends Employe {
4
5         private static final long serialVersionUID = 1L;
6
7         @Column(name = "statut")
8         private String statut;
9         @Column(name = "nb_projets")
10        private int nbProjets;
```

```
1     @Entity
2     @Table(name = "technicien")
3     public class Technicien extends Employe {
4
5         private static final long serialVersionUID = 1L;
6
7         @Column(name = "poste")
8         private String poste;
9         @Column(name = "niveau")
10        private int niveau;
```

Dans cette stratégie, si je crée par exemple un Technicien et que je l’enregistre en base de données avec la méthode persist de l’EntityManager, Hibernate créera bien une ligne dans la table « employe » avec tous les champs des classes Employe et Technicien, plus le discriminant renseigné automatiquement avec la bonne valeur.

De plus, notons que, là encore, toutes les stratégies de génération d’id sont supportées.

**Remarques :** l’annotation **@DiscriminatorColumn** dans la super-classe permet de surcharger le nom de la colonne du discriminant (valeur par défaut : « DTYPE »). Ci-dessous, on renomme cette colonne « discriminator » :

```
1     @Entity
2     @Table(name = "employe")
3     @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
4     @DiscriminatorColumn(
5         name="discriminator",
6         discriminatorType=DiscriminatorType.STRING
7     )
8     public abstract class Employe implements Serializable {
9
10        private static final long serialVersionUID = 1L;
```

```
12      @Id
13      @GeneratedValue(strategy = GenerationType.IDENTITY)
14      @Column(name = "id")
15      private int id;
16      @Column(name = "nom")
17      private String nom;
18      @Column(name = "prenom")
19      private String prenom;
```

De plus, l’annotation **@DiscriminatorValue** aussi bien dans la super-classe que dans les sous-classes permet de surcharger la valeur du discriminant pour cette classe (valeur par défaut : le nom de la classe). Ci-dessous, on donne la valeur « ING » au discriminant pour le type Ingenieur :

```
1      @Entity
2      @Table(name = "ingenieur")
3      @DiscriminatorValue(value="ING")
4      public class Ingenieur extends Employe {
5
6          private static final long serialVersionUID = 1L;
7
8          @Column(name = "statut")
9          private String statut;
10         @Column(name = "nb_projets")
11         private int nbProjets;
```

5 – Conclusion

La meilleure stratégie a priori est **JOINED** : c’est la seule implémentation de l’héritage au sens strict, tant en Java que dans le modèle de données. Elle implique néanmoins systématiquement des jointures en lecture, et potentiellement plusieurs requêtes en écritures. Les requêtes de recherche peuvent vite devenir lourdes.

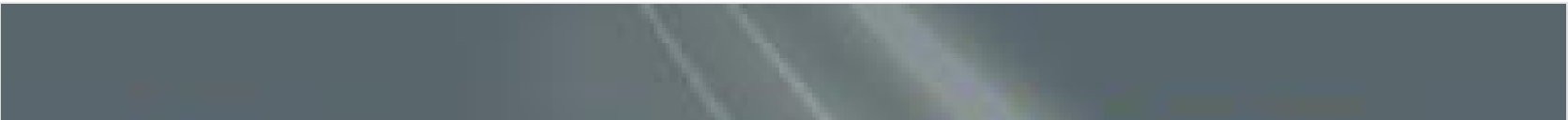
La stratégie **SINGLE\_TABLE** est une alternative très honnête : les performances en lecture et écriture sont très bonnes. En revanche, on a potentiellement un grand nombre de colonnes et beaucoup de valeurs NULL.

Le mode **TABLE\_PER\_CLASS** est un peu le parent pauvre de ce comparatif avec ses allures d’usine à gaz. Il est vrai que l’on perd tout-à-fait l’intérêt de l’héritage dans l’approche base de données (ce qui n’est pas le cas du point de vue Java). On a potentiellement beaucoup de données dupliquées, ce qui signifie qu’une modification sur une des tables impliquées risque de devoir être répercutée sur une ou plusieurs autre(s) table(s). Il propose toutefois des performances correctes en lecture et en écriture (même s’il faut gérer « à la main » la cohérence des données entre la super-classe et les sous-classes).

L’équipe [AXOPEN](#)

[Héritage](#)   [HIBERNATE](#)   [HIBERNATE 4](#)   [JBOSS](#)   [JEE](#)

Voir aussi les articles suivants







**La clause HAVING en Hibernate 4**

Le 27/12/2013 par Florent Tripier

Comment utiliser la clause SQL HAVING avec Hibernate 4 ? Rappel théorique En SQL, lorsqu’une requête possède une condition sur une colonne sur laquelle porte une clause GROUP BY, cette condition n’est pas exprimée dans la clause WHERE mais dans la clause HAVING. En pratique Si vous savez faire une requête simple avec Hibernate 4, écrire une condition simple et utiliser la clause GROUP BY, la mise en place d’un HAVING ne vous posera pas de problème.

[Lire l'article](#)



**Le multiselect avec Hibernate 4**

Le 05/11/2013 par Florent Tripier

Nous avons vu dans l’article « Les requêtes avec Hibernate 4 » comment réaliser une requête ramenant soit un champ soit tous les champs (SELECT \*). Voyons à présent comment ramener plusieurs champs mais pas tous. Le code côté Hibernate Pour ce faire, nous avons recours à la fonction multiselect() de la classe CriteriaQuery. Cette fonction prend en paramètres soit une List de champs, soit n paramètres représentant des champs ou des fonctions.

[Lire l'article](#)



**Les requêtes avec Hibernate 4**

Le 02/10/2013 par Florent Tripier

Hibernate 4 propose une syntaxe structurée qui permet d’exploiter la plupart des fonctionnalités du SQL. Cette syntaxe s’articule autour de quelques classes clés. Les classes clés pour construire une requête EntityManager L’EntityManager vous donne accès à votre PersistenceUnit, c’est-à-dire à votre base de données. C’est donc à

partir de lui que seront construits tous les objets suivants. La configuration de la PersistenceUnit fera l’objet d’un autre article. CriteriaBuilder Le CriteriaBuilder est généré par l’EntityManager.

[Lire l'article](#)



**Agence Lyon**  
44 Avenue Paul Kruger  
69100 Villeurbanne - France



2007 - 2019 © AXOPEN

**La société**

- Qui sommes-nous ?
- Notre histoire
- Nos valeurs
- L'équipe
- Nous rejoindre
- Où nous trouver
- Mentions légales
- Agence de Lyon
- ESN
- Développement web
- Développement mobile

**Expertise**

- Développement Web
- Développement Mobile
- SI & Data

**Métier**

- Développement spécifique
- Maintenance applicative
- Big Data

**Réalisations**

**Formations**

**Blog**

[Contactez-nous](#)