

Introduction à Hibernate

Université de Nice Sophia-Antipolis
Version O 0.5 – 23/10/05
Richard Grin

- ❑ Hibernate est un outil ORM (*Object-Relational Mapping*) open source pour Java
- ❑ Ce cours est un rapide survol de ses possibilités

Outil ORM

- ❑ Automatise ou facilite la correspondance entre des données stockées dans des objets et une base de données relationnelle
- ❑ Le plus souvent les données sont décrites dans des fichiers de configuration (souvent XML)
- ❑ Les principaux produits : TopLink d'Oracle (commercial) et Hibernate (*open source*)

Fonctionnalités de base

- ❑ Recherche et enregistre les données associées à un objet dans une base de données
- ❑ Détecte quand un objet a été modifié et l'enregistre en optimisant les accès à la base

Avantages

- ❑ Évite l'écriture de code répétitif, inintéressant et source d'erreurs difficiles à déceler
- ❑ Gain de 30 à 40 % du nombre de lignes de certains projets
- ❑ Améliore la portabilité du code pour des changements de SGBD

Avantages

- ❑ Le développeur pense en termes d'objet et pas en termes de lignes de tables
- ❑ Sans outil ORM le développeur peut hésiter à concevoir un modèle objet « fin » afin d'éviter du codage complexe pour la persistance
- ❑ Le *refactoring* du schéma de la base de données ou du modèle objet est facilité

Pas toujours bénéfique

- ❑ Un type d'applications ne bénéficie pas de l'utilisation d'un outil ORM : celles qui modifient un grand nombre de lignes pour chaque update ou qui ne comportent essentiellement que des requêtes select de type « group by »
- ❑ En effet, en ce cas la manipulation d'un grand nombre d'objets nuit aux performances
- ❑ Par exemple les applications OLAP (*online analytical processing*), le *data mining*

R. Grin

Hibernate

page 7

Les classes persistantes dans Hibernate

R. Grin

Hibernate

page 8

Persistence pour POJOs

- ❑ Au contraire d'autres outils ou framework, les objets persistants sont des POJOs (*Plain Old Java Objects*)
- ❑ Leur classe n'a pas besoin d'implémenter certaines interfaces ou d'hériter de certaines classes
- ❑ Quelques contraintes sont tout de même recommandées ou obligatoires

R. Grin

Hibernate

page 9

Contraintes obligatoires pour les classes persistantes

- ❑ Elles doivent avoir un constructeur sans paramètre (il peut être privé, mais il est préférable qu'il soit accessible par le packaging)
- ❑ Les collections qui représentent des associations doivent être typées avec des interfaces et pas des classes ; par exemple **List** et pas **ArrayList**

R. Grin

Hibernate

page 10

Contraintes optionnelles mais recommandées

- ❑ Un des champs doit identifier une instance parmi toutes les autres de la même classe
- ❑ Tous les champs qui sont persistants doivent avoir un modificateur (*setter*) et un accesseur (*getter*) ; ils peuvent être privés
- ❑ Les classes ne doivent pas être **final** ; les méthodes public ne doivent pas être **final**

R. Grin

Hibernate

page 11

Fichiers de mapping

R. Grin

Hibernate

page 12

Fichier de mapping

- ❑ Décrit comment se fera la persistance des objets d'une classe
- ❑ Format XML
- ❑ Se place dans le même répertoire que la classe et se nomme *Classe.hbm.xml* si la classe s'appelle *Classe*

Exemple de fichier de mapping (en-tête)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

Exemple de fichier de mapping (suite)

```
<hibernate-mapping>
  <class name="Event" table="EVENTS">
    <id name="id" column="EVENT_ID">
      <generator class="increment"/>
    </id>
    <property name="date"
      type="timestamp"
      column="EVENT_DATE"/>
    <property name="title"/>
  </class>
</hibernate-mapping>
```

nom accesseurs : {get|set}Date

Attributs du tag property

- ❑ **column** indique le nom de la colonne dans la table relationnelle
- ❑ Par défaut, elle a le même nom que la propriété (attribut « **name** »)
- ❑ **type** indique le type « Hibernate » de la propriété ; Hibernate en déduit le type SQL et le type Java
- ❑ Par défaut, il est déterminé par introspection de la classe

Identificateur

- ❑ Le tag **id** est obligatoire pour chaque classe qui est représentée par une table (une classe « composant » n'est pas représentée par une table)
- ❑ Il indique quelle sera la clé primaire de la table
- ❑ Remarque : les sous-classes héritent des identificateurs de leur classe mère

Générateur de clés

- ❑ Le tag **generator**, sous-tag de **id**, indique comment seront générées les clés des objets
- ❑ Le plus souvent les clés sont de type **long**, **int** ou **short** et n'ont pas de signification pour l'application (une bonne pratique)

Valeurs pour l'attribut **class**

- ❑ **increment** : ne convient que si aucun autre processus n'ajoute des données dans la table
- ❑ **identity, sequence** : la clé est une colonne « identity » ou est générée par une séquence
- ❑ **hilo** : une table contient la prochaine valeur
- ❑ **native** : utilise un des moyens ci-dessus, selon les possibilités du SGBD

Clés significatives

- ❑ Bien que les clés significatives ne soient pas recommandées, ce choix peut être imposé par les bases de données héritées d'autres applications
- ❑ **class="assigned"** indique que c'est l'application qui donnera la valeur à la clé
- ❑ C'est la valeur par défaut pour l'attribut **class**

États d'un objet

Les différents états

- ❑ Nouveau : il vient d'être créé dans le code Java ; il n'est pas persistant
- ❑ Persistant : il a été rendu persistant par **session.persist(objet)** ou bien il a été chargé depuis la base de données par une requête (**session.get(...)**)
Toute modification des propriétés de l'objet sera *automatiquement* répercutée dans la base

Les différents états

- ❑ Détaché : il a été persistant mais la session qui le gère a été fermée ; on peut créer une autre session et le rattacher à cette nouvelle session (par **session.update(objet)**)
- ❑ Transient : un objet persistant ou détaché peut être rendu non persistant par l'appel de **session.delete(objet)**

session.persist(objet)

- ❑ Rend l'objet passé en paramètre persistant
- ❑ Une commande SQL INSERT ne sera exécutée qu'au moment où la méthode **session.commit()** sera lancée

Récupérer l'identificateur d'un objet

- ❑ `persist` renvoie un `Serializable` qui est l'identificateur utilisé pour sauvegarder l'objet dans la base
- ❑ Si l'identificateur est un long on peut le récupérer par :

```
Long id =  
    (Long)session.persist(objet);
```

Persistence par transitivité (ou par référence)

Le concept

- ❑ Les objets référencés par un objet persistant sont automatiquement persistants
- ❑ Ainsi si on rend un objet persistant, tous les objets référencés par cet objet sont rendus persistants
- ❑ C'est un comportement logique : un objet ne serait pas vraiment persistant si une partie des valeurs de ses propriétés n'était pas persistante

Un problème difficile

- ❑ Quand on rend un objet non persistant, on risque de supprimer de la base des valeurs utilisées par un autre objet persistant
- ❑ Donc pratiquement la persistance par transitivité n'est pas si simple à mettre en œuvre

Le choix d'Hibernate

- ❑ Par défaut, Hibernate n'effectue pas de persistance par transitivité
- ❑ Pour que les objets associés à un objet persistant deviennent automatiquement persistants, il faut l'indiquer dans le fichier de mapping de la classe de l'objet persistant
- ❑ Ce comportement est moins sûr mais permet plus de souplesse, souvent pour obtenir de meilleures performances

Attribut cascade

- ❑ Les tags qui décrivent les associations entre objets peuvent avoir un attribut `cascade` qui indique le comportement d'Hibernate pour la persistance par transitivité
- ❑ Par défaut, la valeur de l'attribut est `none` : l'objet ou les objets référencés par cette association ne seront pas rendus persistants automatiquement par un `persist` de l'objet référençant

Possibilités de cascade (1)

- ❑ **save-update** : Hibernate suivra l'association lors des appels à persist ou update (persistance automatique des objets référencés)
- ❑ **delete** : Hibernate suivra l'association lors des appels à delete (les objets référencés seront automatiquement rendus non persistants)
- ❑ **all** : les 2 précédents

Possibilités de cascade (2)

- ❑ **delete-orphan** : tout objet associé qui est retiré de l'association sera rendu non persistant
- ❑ **all-delete-orphan** : comme **all** et **delete-orphan** réunis (convient bien si les objets référencés ne peuvent exister en dehors de l'association)

Exemple

- ❑ Si on suppose qu'une adresse n'existe pas sans une personne, et qu'une personne peut avoir plusieurs adresses, on aura dans le fichier de mapping de la classe **Personne** :

```
<set name="adresses"
      table="ADRESSE"
      cascade="all-delete-orphan"
      . . .
```

Les associations

Une partie complexe...

- ❑ C'est la partie la plus complexe si on veut écrire des applications performantes avec Hibernate

Types d'association

- ❑ Uni ou bidirectionnelle
- ❑ Cardinalités correspondant aux associations
1-1, 1-N, N-1, M-N

Tags Hibernate pour les associations

- ❑ Pour les collections : `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` et `<primitive-array>`
- ❑ Les cardinalités avec les tags suivants : `<one-to-one>`, `<one-to-many>`, `<many-to-one>`, `<many-to-many>` (on ajoutera `<join>` pour un cas particulier)

Détails tags pour les collections

- ❑ `<bag>` (sac) : une collection non ordonnée qui peut contenir plusieurs fois le même élément (au contraire d'un `set`)
- ❑ `<primitive-array>` : un tableau de type primitif (`int[]` par exemple)

Fichiers de mapping

- ❑ On se place du point de vue de la classe dont on décrit le mapping
- ❑ Si la classe contient une collection, on utilise un tag de collection qui contient lui-même un tag indiquant la cardinalité de l'association
- ❑ Sinon, on utilise un des tags qui indique la cardinalité

Exemple d'association N-1

- ❑ Cas de l'association unidirectionnelle Employé → Département :

```
<class name="Employe">
  <id name="id" column="employeeId">
    . . .
  </id>
  . . .
  <many-to-one name="dept"
    column="NUM_DEPT"
    class="Departement"/>
</class>
```

correspond au {get|set}Dept du transparent suivant

En Java

```
public class Employe {
    private Departement dept;
    . . .
    public Departement getDept() {
        return dept;
    }
    public void setDept(Departement dept) {
        this.dept = dept;
    }
}
```

Si l'association est bidirectionnelle

```
public class Departement {
    private Set employes = new HashSet();
    . . .
    public Set getEmployes() {
        return employes;
    }
    public void setEmployes(Set employes) {
        this.employes = employes;
    }
}
```

Fichier de mapping de Departement

```
<class name="Departement"
  table="DEPARTEMENT">
  . . .
  <set name="employes">
    <key column="NUM_DEPT" />
    <one-to-many class="Employe"/>
  </set>
</class>
```

correspond au
`getEmployes` du
transparent précédent

colonne de la table
EMPLOYE (pas
DEPARTEMENT) clé
étrangère vers
DEPARTEMENT

Attention, ce fichier est
incomplet

R. Grin

Hibernate

page 43

Pas de gestion automatique des associations bidirectionnelles

- ❑ Au contraire des containers d'EJB entités, Hibernate n'automatise pas la synchronisation des 2 bouts d'une association
- ❑ Si une association est bidirectionnelle, le programmeur doit gérer cette synchronisation

R. Grin

Hibernate

page 44

Problème 1

- ❑ Dans le code Java, il ne faut donc pas oublier de gérer les 2 côtés de l'association
- ❑ Par exemple, si on ajoute un employé dans un département :

```
employe.setDept(dept);
dept.getEmployes().add(employe);
```

- ❑ Pour faciliter la programmation, on ajoute souvent cette méthode dans **Departement** :

```
public void addEmploye(Employe emp) {
  employes.add(emp);
  emp.setDept(this);
}
```

R. Grin

Hibernate

page 45

Problème 1 (suite)

- ❑ On peut maintenant rendre **private** la méthode `getEmployes` de la classe **Departement** pour une meilleure conception (encapsulation)
- ❑ En effet, si Hibernate conseille d'avoir des accesseurs/modificateurs pour les propriétés persistantes, ceux-ci peuvent être **private**

R. Grin

Hibernate

page 46

Problème 2

- ❑ Pour représenter l'association en relationnel, il suffit de positionner une clé étrangère ; il n'y a pas 2 opérations à faire comme en Java
- ❑ On indique à Hibernate de ne pas faire 2 fois la même opération en ajoutant un attribut **inverse** dans le mapping de **Departement** :

```
<set name="employes" inverse="true">
  <key column="NUM_DEPT" />
  <one-to-many class="Employe"/>
</set>
```

R. Grin

Hibernate

page 47

Le fichier de mapping complet

```
<class name="Departement"
  table="DEPARTEMENT">
  . . .
  <set name="employes"
    inverse="true">
    <key column="NUM_DEPT" />
    <one-to-many class="Employe"/>
  </set>
</class>
```

R. Grin

Hibernate

page 48

Association M-N

- ❑ Les associations M-N sont traduites en relationnel par une table association
- ❑ En objet elles sont traduites par des collections
- ❑ Dans le fichier de mapping on aura donc un tag « collection » pour indiquer l'association ; ce tag collection indiquera les noms des colonnes de la table association
- ❑ Le nom de la colonne qui référence la table correspondant à la classe du fichier de mapping est indiqué par le tag **key**

R. Grin

Hibernate

page 49

Exemple d'association M-N

- ❑ Association M-N qui traduit la participation d'un employé à un projet
- ❑ Pour enregistrer une nouvelle participation, le code
 - doit ajouter le projet dans la collection de la classe **Employé** des projets auxquels l'employé participe
 - **et** il doit aussi ajouter l'employé dans la collection de la classe **Projet** des employés qui participent au projet

R. Grin

Hibernate

page 50

Code Java

- ❑ Méthode de la classe **Employé** qui ajoute la participation d'une personne à un événement (il faut choisir une des 2 classes pour ajouter ce type de méthode) :

```
public void addProjet(Projet projet) {  
    this.getProjets().add(projet);  
    projet.getEmployes().add(this);  
}
```

R. Grin

Hibernate

page 51

Fichiers de mapping

```
<class name="Employe">  
    . . .  
    <set name="projets"  
        table="PARTICIPATION">  
        <key column="CODE_PROJET"/>  
        <many-to-many  
            class="Projet"  
            column="CODE_PROJET"  
        />  
    />  
</class>
```

colonne de la table association qui référence la table **PROJET**

colonne de la table association qui référence la table **EMPLOYE**

R. Grin

Hibernate

page 52

Contrainte pour le code Java

- ❑ Les associations doivent être typées avec des interfaces : **Set**, **Collection**, **Map** ou **SortedMap**
- ❑ En effet, Hibernate utilisera ses propres classes pour implémenter ces interfaces
- ❑ Il pourra ainsi faire des chargements retardés (*lazy*) et suivre les ajouts et les suppressions dans les associations pour gérer la persistance transitive (attributs « cascade »)

R. Grin

Hibernate

page 53

Héritage

R. Grin

Hibernate

page 54

- ❑ Hibernate supporte les 3 stratégies principales pour la correspondance en relationnel des hiérarchies d'héritage

R. Grin

Hibernate

page 55

Stratégie

« une seule table par hiérarchie »

- ❑ Il faut utiliser les sous-tags suivant du tag `<class>` dans les fichiers de mapping :
 - `<discriminator>` indique la colonne de la table qui contient la valeur qui permet de différencier les lignes correspondant aux sous-classes dans la table
 - `<subclass>` introduit une sous-classe de la racine de la hiérarchie ; ce tag peut contenir lui-même des sous-tags `<subclass>`

R. Grin

Hibernate

page 56

Exemple

```
<hibernate-mapping>
  <class name="Paielement"
    table="PAIEMENT"
    discriminator-value="P">
    <id name="id" column="PAIEMENT_ID"
      type="long">
      <generator class="native"/>
    </id>
    <discriminator column="TYPE_PAIELEMENT"
      type="string"/>
    <property name="proprietaire"
      column="PROPRIETAIRE" type="string"/>
    . . .
```

R. Grin

Hibernate

page 57

Exemple (suite)

```
  <subclass name="CarteCredit"
    discriminator-value="CC">
    <property name="type"
      column="TYPE_CC"/>
    . . .
  </subclass>
  . . .
</class>
</hibernate-mapping>
```

R. Grin

Hibernate

page 58

Stratégie « une table par classe »

- ❑ On utilise le sous-tag `<joined-subclass>` du tag `<class>` ; il peut contenir lui-même des sous-tags `<joined-subclass>`
- ❑ Ce sous-tag contient lui-même un sous-tag `<key>` qui permet d'indiquer la clé primaire de la sous-classe, qui est aussi une clé étrangère vers la clé primaire de la classe mère

R. Grin

Hibernate

page 59

Exemple

```
<hibernate-mapping>
  <class name="Paielement"
    table="PAIEMENT">
    <id name="id" column="PAIEMENT_ID"
      type="long">
      <generator class="native"/>
    </id>
    <property name="proprietaire"
      column="PROPRIETAIRE" type="string"/>
    . . .
```

R. Grin

Hibernate

page 60

Exemple (suite)

```
<joined-subclass name="CarteCredit"
  table="CARTE_CREDIT">
  <key column="CARTE_CREDIT_ID">
  <property name="type"
    column="TYPE_CC"/>
  ...
</joined-subclass>
...
</class>
</hibernate-mapping>
```

Nom de la colonne
qui est clé primaire
de la table
CARTE_CREDIT et
clé étrangère vers la
table **PAIEMENT**

R. Grin

Hibernate

page 61

Stratégie

« une table par classe concrète »

- ❑ Chaque classe concrète est définie à part
- ❑ Les fichiers de mapping ne font pas référence aux autres classes

R. Grin

Hibernate

page 62

Retrouver des données

R. Grin

Hibernate

page 63

Requêtes

- ❑ Hibernate offre plusieurs façons de retrouver des données enregistrées dans la base
- ❑ 2 de ces façons utilisent une vision totalement objet des données
- ❑ Une 3^{ème} façon permet un accès direct par SQL

R. Grin

Hibernate

page 64

3 façons de retrouver des données

- ❑ Langage HQL :

```
session.createQuery("from Employe e where  
e.nome like 'Dup%'");
```

- ❑ L'API Criteria pour QBC (*query by criteria*) et pour QBE (*query by example*) :

```
session.createCriteria(Employe.class).add  
(Expression.like("nome", "Dup%"));
```

R. Grin

Hibernate

page 65

3 façons de retrouver des données

- ❑ SQL direct avec mapping automatique du résultat avec les objets :

```
session.createSQLQuery("select {e.*} from  
EMPLOYE {e} where NOM like 'Dup%', 'e',  
Employe.class);
```

R. Grin

Hibernate

page 66

Quelle façon utiliser ?

- ❑ HQL est préconisé pour les requêtes complexes

HQL

```
Criteria crit = session.createQuery("from  
    Employe e where e.nom like 'Dup%'");  
crit.addOrder(Order.asc("nom"));  
crit.setFirstResult(0);  
crit.setMaxResults(20);  
List noms = crit.list();
```

HQL – Chaînage des méthodes

```
List noms = session.createQuery("from  
    Employe e where e.nom like 'Dup%'")  
    .addOrder(Order.asc("nom"))  
    .setFirstResult(0)  
    .setMaxResults(20)  
    .list();
```

Danger !

- ❑ Il est dangereux de construire une requête à partir d'une chaîne de caractères dont une partie est constituée de chaînes saisies par un utilisateur
- ❑ Il faut plutôt utiliser les possibilités de paramétrage offertes par Hibernate

Paramètres

- ❑ 2 possibilités pour paramétrer les requêtes
 - À la JDBC (avec des « ? »)
 - Avec des paramètres nommés (:nomparam)
- ❑ Les paramètres nommés sont recommandés :
 - les paramètres de position sont plus sensibles aux modifications (si on ajoute un paramètre par exemple)
 - un paramètre nommé peut apparaître plusieurs fois dans une requête

À la JDBC

```
String query =  
    "from Employe e where e.nom like ? and  
    e.dateEmbauche > ? ";  
List noms =  
    session.createQuery(query)  
        .setString(0, "Dup%") // 0 et pas 1 !  
        .setDate(1, dateDebut)  
        .list();
```

de type Date



Paramètres nommés

```
String query =
    "from Employe e where e.nom like
    :modeleNom";
List noms =
    session.createQuery(query)
        .setString("modeleNom", "Dup%")
        .list();
```

setEntity

- ❑ Il est possible d'attacher un objet entier à un paramètre :

```
session.createQuery(
    "from Employe e where e.dept = :dept")
    .setEntity("dept", dept)
    .list();
```

lazy

- ❑ Lorsqu'une requête récupère des objets qui contiennent une collection qui traduit une association, cette collection n'est pas initialisée sauf si on le demande explicitement
- ❑ Exemple : si on récupère un département, la collection des employés n'est pas initialisée par défaut
- ❑ La collection ne sera initialisée que lorsqu'elle sera absolument nécessaire, par exemple si on veut des informations sur un des employés

Mode de récupération *eager*

- ❑ Avec ce mode on peut indiquer à Hibernate d'initialiser une collection
- ❑ Dans les exemples suivants on indique à Hibernate d'initialiser les collections **employes** des instances de **Département** récupérées
- ❑ Avec HQL :

```
from Departement d
left join fetch d.employes e
```
- ❑ Avec Criteria :

```
criteria.setFetchMode("employes",
    FetchMode.EAGER)
```

Quelques problèmes et solutions pour l'implémentation

Dirty checking

- ❑ Lorsque la transaction est validée il est inutile d'enregistrer tous les objets enregistrés dans la session
- ❑ Il faut donc un moyen de connaître les objets qui ont été modifiés
- ❑ Essentiellement 2 solutions :
 - on intercepte les modifications pour enregistrer si un objet a été modifié ou non
 - on compare les valeurs des objets au moment où on les récupère et au moment où on devrait les enregistrer

Dirty checking

- ❑ La 1^{ère} solution (interception) implique une modification du code source ou, le plus souvent, du *bytecode*
- ❑ La 2^{ème} solution implique que l'on puisse faire une copie de l'objet (ou des valeurs de l'objet) et que l'on puisse comparer les valeurs (souvent avec la méthode `equals`)

Dirty checking

- ❑ Pour savoir si on doit suivre les associations 1-N ou M-N qui partent d'un objet persistant, on doit aussi utiliser ses propres classes de collection pour enregistrer les informations nécessaires

Lazy loading

- ❑ Le plus souvent on a besoin de modifier le code de l'application
- ❑ Par exemple, on remplace l'objet par un proxy qui représente l'objet mais ne contient pas tout l'état de l'objet
- ❑ Dans le cas où on veut gérer le chargement ou pas des objets associés à un objet (associations 1-N ou M-N), on doit aussi utiliser ses propres classes de collection

Configuration d'Hibernate

Configuration d'Hibernate

- ❑ Il faut décrire en particulier le SGBD utilisé pour la persistance
- ❑ La configuration est contenue dans un fichier `hibernate.cfg.xml` placé dans le classpath

Exemple de configuration (en-tête)

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

session-factory

- ❑ Un tag **session-factory** par base de données (le plus souvent, une seule base) :

```
<hibernate-configuration>
  <session-factory>
    . . .
  </session-factory>
  <session-factory>
    . . .
  </session-factory>
</hibernate-configuration>
```

R. Grin

Hibernate

page 85

session-factory

- ❑ Chaque session-factory contient les informations pour obtenir une connexion à la base, le dialecte SQL (Oracle, DB2,...) et divers autres informations

R. Grin

Hibernate

page 86

Connexion à la base

```
<property name="connection.driver_class">
  oracle.jdbc.driver.OracleDriver
</property>
<property name="connection.url">
  jdbc:oracle:thin:@si.unice.fr:1521:INFO
</property>
<property name="connection.username">
  toto
</property>
<property name="connection.password">
  xxxxx
</property>
```

R. Grin

Hibernate

page 87

Autres informations

```
<!-- JDBC connection pool -->
<property name="connection.pool_size">1
</property>
<property name="dialect">
  org.hibernate.dialect.OracleDialect
</property>
<!-- Affiche les ordres SQL exécutés -->
<property name="show_sql">true</property>
<!-- Recrée la base au démarrage -->
<property
  name="hbm2ddl.auto">create</property>
<mapping resource="Event.hbm.xml"/>
```

R. Grin

Hibernate

page 88

Mise au point

R. Grin

Hibernate

page 89

Fichier de log

- ❑ Le fichier de configuration log4j.properties doit se trouver dans le fichier src et recopié pour l'exécution dans le répertoire d'exécution
- ❑ Si le fichier jar « log4jxxx.jar » n'est pas visible, l'API standard de logging sera utilisé par Hibernate
- ❑ Il faut configurer le logging pour qu'il renvoie les erreurs dans un fichier de log

R. Grin

Hibernate

page 90

Exemple pour log4j (1)

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file hibernate.log ###
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=hibernate.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

R. Grin

Hibernate

page 91

Exemple pour log4j (2)

```
### set log levels - for more verbose logging
change 'info' to 'debug'
log4j.rootLogger=warn, file
log4j.logger.net.sf.hibernate=info

### enable the following line if you want to track
down connection
### leakages when using
DriverManagerConnectionProvider
log4j.logger.net.sf.hibernate.connection.DriverManagerConnectionProvider=trace

### log JDBC bind parameters ###
log4j.logger.net.sf.hibernate.type=info

### log prepared statement cache activity ###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

R. Grin

Hibernate

page 92

Requêtes SQL

- ❑ Hibernate affiche sur la console les requêtes SQL qu'il lance
- ❑ Exemple d'affichage :
Hibernate: insert into EVENTS
(EVENT_DATE, title, EVENT_ID) values (?, ?, ?)

R. Grin

Hibernate

page 93

schemaexport

- ❑ Il peut aussi être intéressant d'étudier les commandes DDL (create table...) lancées par Hibernate pour créer les tables
- ❑ On peut ainsi voir si les fichiers de configuration sont correctement écrits

R. Grin

Hibernate

page 94

Bibliographie

- ❑ Hibernate In Action par Christian Bauer et Gavin King, édition Manning
- ❑ Traduction française : Hibernate, édition CampusPress, collection Référence

R. Grin

Hibernate

page 95