9

L'outillage d'Hibernate

Selon votre cycle d'ingénierie, il est possible d'améliorer la productivité de vos applications en utilisant l'outillage proposé par Hibernate. Les outils que vous utiliserez le plus dépendront de votre méthode de conception et de l'existant pour une application donnée.

La finalité première des différents outils proposés par Hibernate est de faciliter la génération de code :

- génération des classes java pour les POJO ;
- génération des fichiers de mapping ;
- génération des scripts de création de schéma de base de données (DDL).

Hibernate 3 couvre une grande partie de ces besoins, notamment grâce aux outils Hibernate Explorer et SchemaExport.

Il est en outre possible d'interfacer Hibernate avec des mécanismes extérieurs de gestion de pool de connexions ou de cache de second niveau ainsi que de monitorer les statistiques.

Ajoutées à l'outillage spécifique d'Hibernate, ces extensions facilitent l'interaction entre les couches logicielles et matérielles constituant vos applications.

L'outillage relatif aux métadonnées

Comme vous avez pu le voir au chapitre 3, les métadonnées constituent un dictionnaire assez volumineux de paramètres à maîtriser. Nativement décorrélées de vos classes Java car externalisées dans des fichiers XML de mapping, elles peuvent paraître difficiles à écrire et à maintenir.

Les annotations proposées par Hibernate 3 et l'outil XDoclet apportent de la souplesse dans la gestion des métadonnées en permettant de les définir au cœur même des sources de vos classes.

Les annotations

Les annotations représentent un sous-ensemble de la spécification des EJB 3.0 (JSR 220). Elles seront probablement extraites de la JSR afin d'être partagées par tout type d'application Java. Cette section s'inspire en grande partie de la documentation officielle écrite par Emmanuel Bernard, membre de l'équipe Hibernate *(http://www.hibernate.org/247.html)*.

Les annotations sont donc un moyen de décrire les métadonnées. Rappelons que ces dernières sont indispensables pour gérer la transformation du monde relationnel vers le monde objet et *vice versa.* Nous avons vu que les métadonnées étaient écrites dans des fichiers XML.

Tout comme XDoclet *(voir la section suivante),* les annotations proposent de stocker les métadonnées dans les sources des classes persistantes. Cela permet de limiter la maintenance tout en fournissant au développeur les informations de mapping directement sur la source de la classe persistante qu'il manipule. Contrairement à XDoclet, cependant, les annotations ne sont pas un moyen de générer les fichiers XML, l'analyse des informations qu'elles décrivent étant complètement transparente.

Annotations Hibernate et JSR 220

Hibernate propose plus de fonctionnalités en relation avec les annotations que celles spécifiées par la JSR 220. À moyen terme, cette dernière implémentera l'ensemble des fonctionnalités avancées proposées par Hibernate.

Le prérequis pour utiliser les annotations de manière transparente est de disposer d'un JDK 1.5. Si vous ne pouvez mettre à jour votre version du JDK, JBoss propose un compilateur permettant de travailler avec les annotations sous un JDK 1.4.

Installation du JDK 1.5 sous Eclipse

Le lecteur maîtrisant le paramétrage d'Eclipse peut passer directement à la section suivante.

Après avoir téléchargé et installé le JDK 1.5, ouvrez Eclipse, allez dans Windows, Preference, Java et Installed Jres, et configurez un nouveau JRE, comme indiqué à la figure 9.1.

Au besoin, dans Windows, Preference, Java et Compiler, sélectionnez la compatibilité par défaut avec la version 5.0, comme indiqué à la figure 9.2.

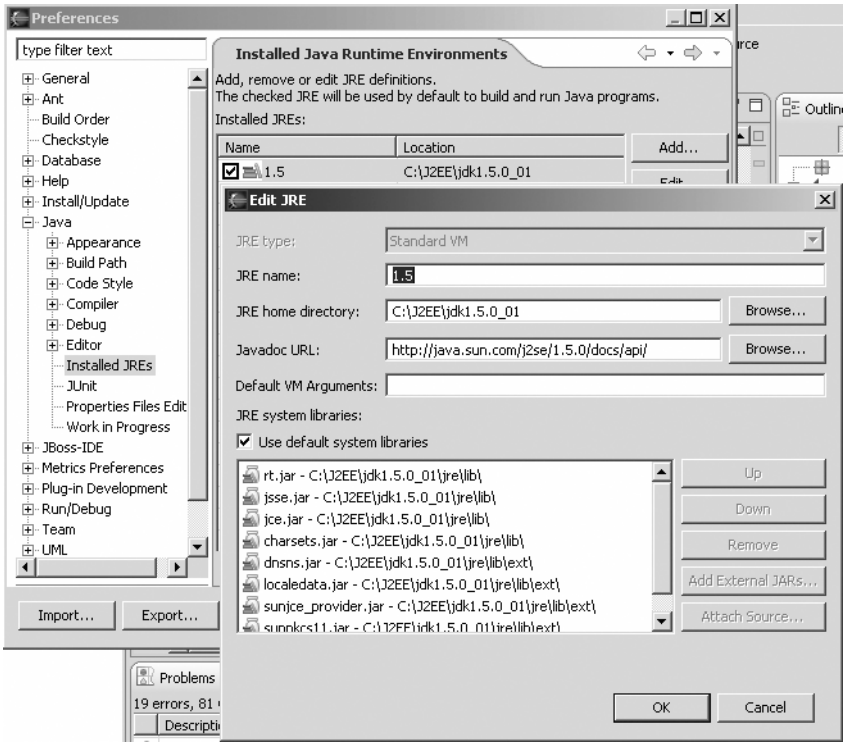


Figure 9.1
Installation du JDK 1.5 sous Eclipse

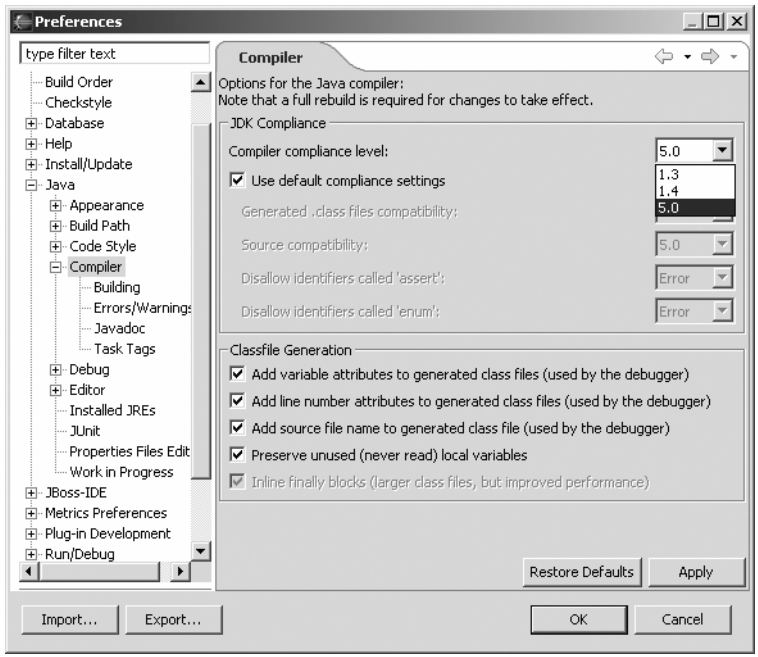


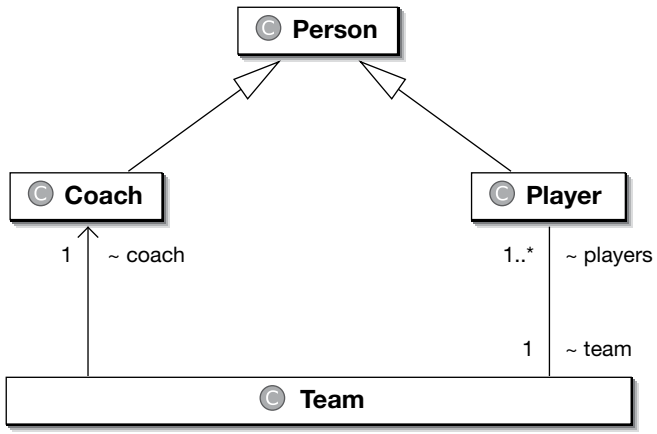
Figure 9.2
Compatibilité par défaut avec la version 5.0

À l'heure où nous écrivons ces lignes, certaines instabilités relatives aux annotations sont constatées dans Eclipse. Cela devrait vite être corrigé.

Exemple d'utilisation

Une fois de plus, nous allons partir d'un diagramme de classes exemple *(voir figure 9.3)*.

Figure 9.3
Diagramme de classes exemple



Notre objectif n'est pas de passer en revue l'intégralité des annotations mais de détailler des exemples d'utilisation courants, tels que id, associations, chargement, cascade et héritage.

Les annotations nécessitent l'import préalable du package **javax.persistence.*** pour les classes où elles sont placées.

Les annotations sont placées juste au-dessus de l'écriture des getters ou de la déclaration de la classe. Par défaut, chaque propriété de classe persistante est persistante.

Une classe est déclarée persistante grâce à `@Entity` :

```
// Team.java
@Entity
@Table(name="TEAM")
public class Team implements Serializable{
…}
```

Les propriétés non persistantes sont déclarées par `@Transient` :

```
@Transient
public Integer getCalculatedField () {
  return calculatedField;
}
```

Une classe persistante doit obligatoirement définir la propriété identifiante *via* l'annotation `@Id` :

```
// Team.java
@Id(generate = GeneratorType.AUTO)
public Long getId() {
  return id;
}
```

Entre parenthèses, vous pouvez spécifier la stratégie de génération :

- `AUTO` : selon la base de données relationnelle utilisée, sélectionne la colonne `identity` ou l'utilisation d'une séquence.
- `TABLE` : table contenant l'id.
- `IDENTITY` : colonne de type `identity`.
- `SEQUENCE` : génération *via* une séquence.
- `NONE` : la clé est à renseigner par l'application.

Certains générateurs prennent un paramètre. C'est notamment le cas d'une génération par séquence, dans laquelle il faut spécifier le nom de la séquence :

```
// Team.java
@Entity
@Table(name="TEAM")
@javax.ejb.SequenceGenerator(
    name="SEQ_TEAM",
    sequenceName="my_sequence"
)
public class Team implements Serializable{
…
  @Id(generate = GeneratorType.AUTO, generator="SEQ_TEAM")
  public Long getId() {
    return id;
  }
}
```

Une association `one-to-one` pure (partageant la même clé primaire) se définit comme suit :

```
// Team.java
@OneToOne
public Coach getCoach() {
  return coach;
}
```

Une association `one-to-one` fondée sur des clés étrangères doit définir la colonne de jointure :

```
// Team.java
@OneToOne
@JoinColumn(name="COACH_ID")
public Coach getCoach() {
  return coach;
}
```

Une association `many-to-one` reprend la même logique :

```
// Player.java
@ManyToOne
@JoinColumn(name="TEAM_ID")
public Team getTeam() {
  return team;
}
```

Les incidences des spécifications des annotations JSR 220 sur les collections sont les suivantes :

- Seules les collections et les set sont acceptés. Pour le moment, les list et map ne sont pas supportées.
- Si votre collection n'est pas un `generic` (type Java 5), vous devez spécifier le type des éléments de la collection avec `targetEntity`.
- Les associations `one-to-many` sont toujours bidirectionnelles. L'autre extrémité (`many-to-one`) doit donc être déclarée. L'extrémité `one-to-many` est forcément `inverse` (dans les fichiers de mapping cela correspond au nœud `inverse="true"`).

Voici un exemple de déclaration de collection :

```
// Team.java
@OneToMany(targetEntity="Player")
@JoinColumn(name="TEAM_ID")
public Set getPlayers() {
  return players;
}
```

```
// Team.java
@OneToMany
@JoinColumn(name="TEAM_ID")
public Set<Player> getPlayers() {
  return players;
}
```

La définition d'une relation `many-to-many` demande la déclaration de la table d'association. Dans le cadre d'une relation bidirectionnelle, vous devez en outre définir l'extrémité inverse.

Dans notre diagramme de classes, nous n'avons pas d'exemple d'association `many-to-many`. Évoquons donc simplement une relation entre un `Article` et une `Catégorie` :

```
// Article.java
@Entity()
public class Article implements Serializable {
  @ManyToMany(targetEntity="Categorie")
  @AssociationTable(
    table=@Table(name="ARTICLE_CATEGORIE"),
    joinColumns={@JoinColumn(name="ARTICLE_ID")},
    inverseJoinColumns={@JoinColumn(name="CATEGORIE_ID")}
```

```
  )
  public Collection getCategories() {
    return categories;
  }
  …
}
```

```
// Categorie.java
@Entity()
public class Categorie implements Serializable {
  @ManyToMany(isInverse=true)
  @AssociationTable(
    table=@Table(name="ARTICLE_CATEGORIE"),
    joinColumns={@JoinColumn(name="CATEGORIE_ID")},
    inverseJoinColumns={@JoinColumn(name="ARTICLE_ID")}
  )
  public Collection getArticles() {
    return articles;
  }
}
```

Trois stratégies d'héritage sont disponibles *via* les annotations :

- Une table par classe fille, qui limite notamment le polymorphisme : `strategy=InheritanceType. TABLE_PER_CLASS`.
- Une table pour la hiérarchie de classes, avec utilisation d'une colonne discriminante : `strategy=InheritanceType.SINGLE_TABLE`.
- Une table par classe, plus connue sous le nom de *joined subclass strategy :* `strategy=InheritanceType. JOINED`.

Prenons la stratégie par colonne discriminante comme exemple. La classe mère doit définir quelle est la colonne discriminante et si elle est concrète, ainsi que la valeur de la colonne pour son type :

```
// Person.java
@Entity
@Table(name="PERSON")
@Inheritance(
  strategy=InheritanceType.SINGLE_TABLE,
  discriminatorType=DiscriminatorType.STRING,
  discriminatorValue="PERSON"
)
@DiscriminatorColumn(name="PERSON_TYPE")
public class Person {
…
}
```

Les classes héritées ne définissent que la valeur de la colonne discriminante :

```
// Coach.java
@Entity
```

```
@Table(name="COACH")
@Inheritance(discriminatorValue="COACH")
public class Coach extends Person implements Serializable{
..
}
```

```
// Player.java
@Entity
@Table(name="PLAYER")
@Inheritance(discriminatorValue="PLAYER")
public class Player extends Person implements Serializable{
…
}
```

Comme dans les fichiers de mapping, la persistance transitive peut aussi être spécifiée par la notion de cascade au niveau des associations :

- `CascadeType.PERSIST` est équivalent à `cascade="persist"`.
- `CascadeType.MERGE` est équivalent à `cascade="merge"`.
- `CascadeType.REMOVE` est équivalent à `cascade="delete"`.
- `CascadeType.ALL` cumule les trois paramétrages précédents.

Voici un exemple d'annotations permettant de spécifier la persistance transitive selon les actions de création et de suppression :

```
// Team.java
@OneToMany(
  targetEntity="Player",
  cascade={CascadeType.PERSIST, CascadeType.MERGE}
)
@JoinColumn(name="TEAM_ID")
public Set getPlayers() {
  return players;
}
```

Notez la possibilité de spécifier plusieurs options pour `cascade` en utilisant la syntaxe entre accolades.

Le chargement des propriétés et associations est paramétrable par le biais de l'attribut `fetch`, qui prend comme valeur `FetchType.LAZY` ou `FetchType.EAGER`.

Complétons notre association précédente en spécifiant un chargement à la demande :

```
// Team.java
@OneToMany(
  targetEntity="Player",
  cascade={CascadeType.PERSIST, CascadeType.MERGE},
  fetch = FetchType.LAZY
)
@JoinColumn(name="TEAM_ID")
public Set getPlayers() {
```

```
    return players;
  }
```

Les premières critiques faites à l'encontre des annotations sont leur potentielle nuisance à la lisibilité et la pollution du source de vos classes. Une fois essayées, les annotations seront cependant probablement adoptées. Leur utilisation est en effet très intuitive, et l'outillage de l'IDE améliore la rapidité de leur écriture. Concrètement, les annotations faciliteront et accéléreront le paramétrage de la persistance de vos applications.

Les annotations n'en sont évidemment qu'à leur début. Actuellement, l'accent est mis sur l'implémentation des spécifications EJB 3.0 (JSR 220). Progressivement, l'équipe d'Hibernate et son leader pour les annotations Emmanuel Bernard enrichiront le jeu d'annotations pour l'affiner.

Nous vous recommandons de consulter régulièrement le site consacré aux annotations ainsi que le forum dédié d'Hibernate 3.0 *(http://forum.hibernate.org/viewforum.php?f=9 et http://www.hibernate.org/247.html).*

XDoclet

XDoclet est un projet Open Source à part entière, qui propose des artifices de génération de code. En ce qui concerne Hibernate, XDoclet s'appuie sur le même concept que les annotations, puisque son rôle est d'enrichir les sources de vos classes persistantes avec des métadonnées.

L'inconvénient de taille de XDoclet vient de ce qu'il n'est pas interprété à l'exécution et qu'il permet juste de générer les fichiers de mapping. Il semble donc *a priori* moins puissant que les annotations.

Pour autant, si vous avez déjà pris l'habitude de l'utiliser ou si vous ne pouvez travailler avec le JDK 1.5, XDoclet reste une solution éprouvée et fiable. Malheureusement, XDoclet n'est pour l'instant synchronisé qu'avec les versions 2 d'Hibernate.

Utilisation des tags XDoclet

Vous devez dans un premier temps télécharger XDoclet et ses dépendances et les importer dans votre projet *(voir le site de référence http://xdoclet.sourceforge.net/xdoclet/install.html).*

Comme pour les annotations, nous allons reprendre les fonctionnalités principales de XDoclet en décrivant les sources des classes persistantes illustrées à la figure 9.3.

Les tags XDoclet doivent être placés dans les javadoc. Comme pour les annotations, l'autocomplétion peut être installée dans votre IDE.

Pour déclarer une classe persistante, le tag `@hibernate-class` doit être spécifié dans la javadoc au niveau classe. Ce tag prend divers attributs, comme `table` pour le nom de la table.

```
// Team.java
/**
```

```
   * @hibernate.class
   *  table="TEAM"
   */
public class Team implements Serializable{
  …
}
```

Le tag `@hibernate-id` permet de paramétrer la propriété identifiante et se place dans la javadoc au niveau de la méthode `getId()` :

```
// Team.java
/**
 * @hibernate.id
 *  generator-class="native"
 *  column="TEAM_ID"
 */
public Long getId() {
  return id;
}
```

Contrairement aux annotations, il faut spécifier manuellement chacune des propriétés comme étant persistante :

```
// Team.java
/**
 * @hibernate.property
 *  column="TEAM_NAME"
 */
public String getName() {
  return name;
}
```

La déclaration d'association se fait dans la javadoc au niveau getter.

Exemple de `one-to-one` :

```
// Coach.java
/**
 * @hibernate.one-to-one
 *  property-ref="TEAM"
 */
public Team getTeam() {
  return team;
}
```

Exemple de `many-to-one` :

```
// Player.java
/**
 * @hibernate.many-to-one
 *  column="TEAM_ID"
 */
public Team getTeam() {
  return team;
}
```

Exemple de `one-to-many` inverse :

```
// Team.java
/**
 * @hibernate.set
 *  inverse="true"
 * @hibernate.collection-key
 *  column="TEAM_ID"
 * @hibernate.collection-one-to-many
 */
public Set getPlayers() {
  return players;
}
```

Contrairement aux annotations, le type de la collection n'est pas déduit. Il faut donc spécifier `@hibernate.set`, `@hibernate.bag`, `@hibernate.map`, `@hibernate.list`, etc.

La déclaration de l'héritage s'effectue au niveau de la classe. D'après notre diagramme de classes, `Person` est la classe mère, et `Player` et `Coach` héritent de `Person`. Nous choisissons donc la stratégie par discriminateur pour implémenter cet héritage en base de données :

```
// Person.java
/**
 * @hibernate.class
 *  table="PERSON"
 *  discriminator-value="PERSON"
 * @hibernate.discriminator
 *  column="PERSON_TYPE"
 */
public class Person {
…
}
```

Ici, deux tags XDoclet sont utilisés. Tout d'abord `@hibernate.class`, avec l'attribut `discriminator-value`, puis `@discriminator`, avec l'attribut `column` pour déclarer la colonne de la table, qui permettra de déduire le type de l'instance retournée par un enregistrement donné.

Les classes héritées doivent utiliser le tag `@hibernate.subclass`, accompagné de l'attribut `discriminator-value` :

```
// Coach.java
/**
 * @hibernate.subclass
 *  discriminator-value = "COACH"
 */
public class Coach extends Person implements Serializable{
  …
}
```

La classe `Player` reprend le même type de spécification :

```
// Player.java
/**
 * @hibernate.subclass
 *  discriminator-value = "PLAYER"
 */
public class Player extends Person implements Serializable{
  …
}
```

L'enrichissement des sources de vos classes persistantes n'est que la première étape dans l'utilisation de XDoclet.

Génération des fichiers de mapping

Une fois les sources complétées, il vous faut déclencher la génération des fichiers de mapping. Pour cela, vous utilisez Ant avec le paramétrage suivant, notamment la cible `"generate"` :

```
<project name="SportTracker" default="schemaexport" basedir=".">
  <property name="lib.dir" value="WEB-INF/lib"/>
  <property name="src.dir" value="WEB-INF/src"/>
  <property name="generated" value="WEB-INF/src"/>
  <property name="build.dir" value="WEB-INF/build"/>
  <property name="classes.dir" value="WEB-INF/classes"/>
  <property name="generated.forced" value="true"/>
  <property name="javadoc"
    value="http://java.sun.com/j2se/1.3/docs/api"/>
  <property name="javac.debug" value="on"/>
  <property name="javac.optimize" value="off"/>
…
  <path id="lib.class.path">
    <fileset dir="${lib.dir}">
      <include name="**/*.jar"/>
    </fileset>
    <pathelement path="${clover.jar}"/>
  </path>

  <target name="generate"
    description="Runs all auto-generation tools.">
    <taskdef name="hibernatedoclet"
      classname="xdoclet.modules.hibernate.HibernateDocletTask">
      <classpath>
        <fileset dir="${lib.dir}/xdoclet-1.2.2">
          <include name="*.jar"/>
        </fileset>
      </classpath>
    </taskdef>
    <hibernatedoclet
      destdir="${src.dir}"
      excludedtags="@version,@author,@todo"
      force="${generated.forced}"
```

```
      mergedir="${src.dir}"
      verbose="false">

      <fileset dir="${src.dir}">
        <include name="**/*.java"/>
      </fileset>
      <hibernate version="2.0"/>
    </hibernatedoclet>
  </target>
</project>
```

Si vous avez correctement écrit vos sources et paramétré votre tâche Ant, l'exécution devrait tracer les informations suivantes :

```
generate:
[hibernatedoclet] (XDocletMain.start                    47  )
  Running <hibernate/>
[hibernatedoclet] Generating mapping file for
  com.eyrolles.sportTracker.model.Team.
[hibernatedoclet] com.eyrolles.sportTracker.model.Team
[hibernatedoclet] Generating mapping file for
  com.eyrolles.sportTracker.model.Person.
[hibernatedoclet] com.eyrolles.sportTracker.model.Person
[hibernatedoclet] com.eyrolles.sportTracker.model.Player
[hibernatedoclet] com.eyrolles.sportTracker.model.Coach
BUILD SUCCESSFUL
Total time: 7 seconds
```

Dans le cas où vous souhaiteriez ajouter des informations qui ne pourraient être générées par XDoclet, vous avez à votre disposition une fonction de fusion. Utilisez pour cela le répertoire défini dans la tâche Ant par `mergedir`, et suivez les informations apparaissant en commentaire dans les fichiers de mapping générés.

Voici, par exemple, le résultat partiel de la génération de **Team.hbm.xml :**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping
>
  <class
    name="com.eyrolles.sportTracker.model.Team"
    table="TEAM"
    dynamic-update="false"
    dynamic-insert="false"
    select-before-update="false"
    optimistic-lock="version"
    >
    <id
      name="id"
      column="TEAM_ID"
```

```
      type="java.lang.Long"
    >
      <generator class="native">
        <!--
          To add non XDoclet generator parameters, create a
          file named hibernate-generator-params-Team.xml
          containing the additional parameters and place it
          in your merge dir.
        -->
      </generator>
    </id>
    <one-to-one
      name="coach"
      class="com.eyrolles.sportTracker.model.Coach"
      cascade="none"
      outer-join="auto"
      constrained="false"
    />
    <property
      name="name"
      type="java.lang.String"
      update="true"
      insert="true"
      access="property"
      column="TEAM_NAME"
    />
…
    <set
      name="players"
      lazy="false"
      inverse="true"
      cascade="none"
      sort="unsorted"
    >
      <key
        column="TEAM_ID"
      >
      </key>
      <one-to-many
        class=""
      />
    </set>
    <!--
      To add non XDoclet property mappings, create a file named
      hibernate-properties-Team.xml
      containing the additional properties and place it in
      your merge dir.
    -->
  </class>
</hibernate-mapping>
```

Nous avons volontairement laissé le formatage de XDoclet. Remarquez les commentaires, qui vous permettent de fusionner les mappings générés avec d'autres fichiers. Il est important de noter que les fichiers générés respectent la DTD Hibernate 2.0.

Nous vous conseillons de visiter le site XDoclet pour d'éventuelles mises à jour.

Correspondance entre systèmes de définition des métadonnées

Le tableau 9.1 récapitule les correspondances entre les différents systèmes de définition des métadonnées. Les annotations adoptent une écriture composée d'une partie logique (votre modèle d'entité) et d'une partie physique (votre structure relationnelle cible), ce qui rend parfois la correspondance délicate à lire.

Tableau 9.1. Correspondance des systèmes de définition des métadonnées

Hibernate	Annotations	XDoclet
`<hibernate-mapping>`		`@hibernate.mapping`
schema		schema
catalog		
default-cascade		default-cascade
default-access		
default-lazy		
auto-import		auto-import
package		
`<class>`	`@Entity`	`@hibernate.class`
name		Implicite
table	`@Table(name="")`	table
discriminator-value	discriminatorValue	discriminator-value
mutable		mutable
schema		schema
catalog		
proxy	`@Proxy(lazy=false, proxyClassName="my.Interface")`	proxy
dynamic-update		dynamic-update
dynamic-insert		dynamic-insert
select-before-update		select-before-update
polymorphism		polymorphism
where	`@Where(clause="")`	where
persister		

Tableau 9.1. Correspondance des systèmes de définition des métadonnées *(suite)*

Hibernate	Annotations	XDoclet
batch-size	@BatchSize(size=n)	
optimistic-lock		optimistic-lock
lazy		lazy
entity-name		
catalog		
check		
rowid		
subselect		
abstract		
<subclass>	**@Inheritance(strategy="")**	**@subclass**
<joined-subclass>	**@Inheritance(strategy="")**	**@joined-subclass**
<union-subclass>		Non supporté
name		Implicite
entity-name		
proxy		proxy
table		table
schema		schema
catalog		
subselect		
dynamic-update		dynamic-update
dynamic-insert		dynamic-insert
select-before-update		
extends		
lazy		lazy
abstract		
persister		
check		
batch-size		
node		
<key…/> (pour joined-subclass)		@ joined-sublass-key
<id>	**@Id**	**@id**
name		Implicite
type		type
column	@Column(name="")	column

Tableau 9.1. Correspondance des systèmes de définition des métadonnées *(suite)*

Hibernate	Annotations	XDoclet
unsaved-value		unsaved-value
access		access
<generator class	generate	generator-class
<composite-id>		
class		
name		
node		
unsaved-value		
<discriminator>	**@DiscriminatorColumn**	**@discriminator**
column	name	column
type	discriminatorType	type
force		force
insert		insert
formula		
<version>	**@Version**	**@version**
name		Implicite
column	@Column(name="")	column
type		type
access		access
unsaved-value		unsaved-value
<timestamp>		**@timestamp**
name		Implicite
column		column
type		type
access		access
unsaved-value		unsaved-value
<property>	**@Column**	**@property**
name		Implicite
column	name	column
type		type
update	updatable	update
insert		insert
formula		formula
access		access

Tableau 9.1. Correspondance des systèmes de définition des métadonnées *(suite)*

Hibernate	Annotations	XDoclet
lazy	*Cf* @Basic(fetch="")	
unique	@UniqueConstraint	unique
not-null	nullable	not-null
optimistic-lock		
<many-to-one>	**@ManyToOne**	**@many-to-one**
name	Implicite	Implicite
column	@JoinColumn	column
class	Implicite	class
cascade	cascade	cascade
fetch	fetch	
update		update
insert		insert
property-ref		property-ref
access		
unique	@UniqueConstraint	unique
not-null		not-null
optimistic-lock		
<one-to-one>	**@OneToOne**	**@one-to-one**
name	Implicite	Implicite
class	Implicite	class
cascade	cascade	cascade
constrained		constrained
fetch		
property-ref	@JoinedColumn	property-ref
access		
<component>	**@Dependent**	**@component**
name		Implicite
class		class
insert		
update		
access	access	
lazy	*Cf* fetch	
optimistic-lock		
<property …../>	@DependantAttribute	

Tableau 9.1. Correspondance des systèmes de définition des métadonnées *(suite)*

Hibernate	Annotations	XDoclet
<many-to-one …. />		
<join>		Non supporté
table		
schema		
catalog		
fetch		
inverse		
optional		
<key … />		
<property … />		
<bag>	Prochainement	@bag
<set>	Implicite	@set
<list>	Prochainement	@list
<map>	Prochainement	@map
<array>	Prochainement	@array
<idbag>		Non supporté
name	Implicite	Implicite
table		table
schema		schema
lazy	*Cf* fetch	lazy
inverse	isInverse	inverse
cascade		cascade
sort		where
order-by		order-by
where	@Where(clause="")	where
fetch	fetch	
batch-size	@BatchSize(size=n)	
access		
<key …. />	@JoinColumn	
<index …. />		
<index-many-to-many …/>		
<element …. />		
<composite-element …. />		
<one-to-many … />	@OneToMany	

Tableau 9.1. Correspondance des systèmes de définition des métadonnées *(suite)*

Hibernate	Annotations	XDoclet
<many-to-many … />		
<element>		**@collection-element**
column		column
node		
formula		
type		type
length		length
precision		
scale		
not-null		not-null
unique	@UniqueConstraint	unique
<one-to-many>	**@OneToMany**	**@collection-one-to-many**
class	targetEntity	class
node		
entity-name		
<many-to-many>	**@ManyToMany**	**@collection-many-to-many**
class	targetEntity	class
node		
embed-xml		
entity-name		
column		column
formula		
outer-join		outer-join
fetch		
foreign-key		
unique	@UniqueConstraint	
<composite-element>		**@collection-composite-element**
class		class
<key>	**@JoinColumn**	**@collection-key**
column	name	column
property-ref		
foreign-key		
on-delete		
not-null		

Tableau 9.1. Correspondance des systèmes de définition des métadonnées *(suite)*

Hibernate	Annotations	XDoclet
update		
unique	@UniqueConstraint	
<index>		**@collection-index**
column		column
type		type
length		length
<column>	**@Column**	**@column**
name	name	name
length	length	length
precision		
scale		
not-null	nullable	not-null
unique	@UniqueConstraint	unique
unique-key		unique-key
sql-type		sql-type
index		index
check		

Ce tableau couvre 99 % des besoins en matière de métadonnées. Si certains éléments n'y figurent pas, c'est parce qu'ils relèvent d'une utilisation particulière ou extrêmement poussée ou que leur utilisation est semblable à un autre élément figurant déjà dans le tableau. Ces éléments sont : `any`, `dynamic-component`, `composite-id`, `properties`, `primitive-array`, `list-index`, `map-key`, `map-key-many-to-many`, `index-many-to-many`, `composite-map-key`, `composite-index`, `many-to-any`, `index-many-to-any` et `collection-id`. La consultation de la DTD devrait vous permettre de les spécifier rapidement.

Pour conclure sur les métadonnées, rappelons qu'Hibernate 3 est encore tout nouveau et que l'outillage autour des métadonnées s'étoffera avec le temps, l'enrichissement des annotations étant un chantier à part entière.

Le meilleur conseil à donner aux adeptes des annotations et de XDoclet est de consulter régulièrement les sites dédiés pour obtenir les dernières versions.

En résumé

Partie intégrante de la spécification de persistance EJB 3.0, les annotations tendent à standardiser le mapping objet-relationnel. Encore en phase d'enrichissement pour proposer non seulement les métadonnées spécifiées par la JSR 220 mais aussi prendre en

compte les spécificités d'Hibernate, elles représenteront à moyen terme la meilleure manière de gérer vos métadonnées.

En attendant, utilisez les traditionnels fichiers de mapping, et tirez profit de l'outillage intégré aux environnements de développement, notamment l'éditeur de mapping disponible dans la suite d'outils Hibernate 3, qui vous permettra de gagner un temps précieux.

L'outillage Hibernate Tools

L'outillage disponible autour d'Hibernate sous le nom d'Hibernate Tools permet d'accroître la productivité.

L'éditeur de mapping simplifie l'écriture des fichiers de mapping. De leur côté, les requêtes sont testées rapidement grâce à Hibernate Console tandis que le schéma de base de données est généré rapidement avec SchemaExport.

Les cycles d'ingénierie

Globalement, il existe deux moyens principaux de développer un projet informatique selon que votre entreprise possède un historique de conception orientée objet fondé sur la modélisation du schéma relationnel ou que votre application est destinée à utiliser ou enrichir un schéma relationnel existant. Dans le premier cas votre point de départ est la conception fonctionnelle et dans le second le schéma de base de données.

Si UML est ancré dans votre organisation, après les spécifications fonctionnelles commencent les spécifications techniques. Parmi les différents diagrammes entrant en jeu pendant les phases de modélisation techniques, le diagramme de classes vous permet de modéliser votre modèle métier.

Votre outil de modélisation UML générera assez facilement les sources de vos classes persistantes. En le personnalisant, vous pourrez générer directement les fichiers de mapping ou les sources comportant les annotations. Cette personnalisation passe par un métamodèle définissant les stéréotypes et tagged-values destinés à enrichir le spectre de spécifications couvert par le diagramme de classes.

La figure 9.4 illustre les différentes possibilités de génération de code ou de métadonnées selon votre cycle d'ingénierie.

Les outils permettant d'opérer à partir du schéma de base de données sont les suivants :

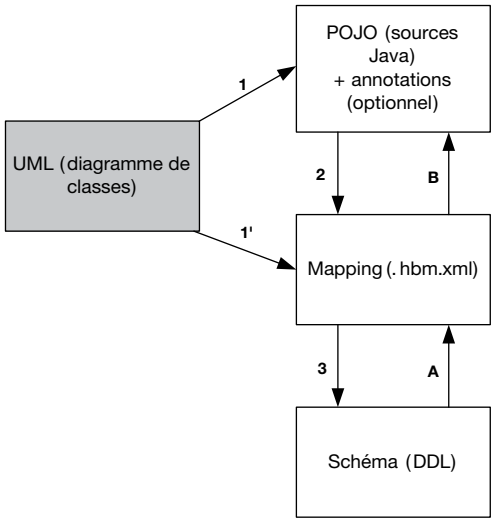
- Pour la génération des fichiers de mapping : Hibernate Explorer ou Middlegen.
- Pour générer les sources des classes persistantes : Hibernate Explorer ou HBM2JAVA.

Les outils disponibles à partir du diagramme de classes standard sont les suivants :

- Pour la génération des fichiers de mapping : vous pouvez personnaliser votre atelier UML ou adopter AndroMDA, qui travaille à partir de l'export XMI.

- Pour générer les fichiers de mapping : XDoclet.
- Pour générer le schéma de base de données : SchemaExport.

Figure 9.4
Possibilités de génération de code et métadonnées



Les outils disponibles à partir du diagramme de classes avec atelier UML personnalisé (enrichi par des stéréotypes et tagged-values) sont les suivants :

- Pour générer des sources Java des classes persistantes : annotations ou fichiers de mapping (les fichiers de mapping ne sont plus nécessaires si vous utilisez les annotations).
- Pour générer le schéma de base de données : SchemaExport.

Nous avons déjà abordé les annotations ainsi que XDoclet. Les sections qui suivent se concentrent sur la nouvelle suite d'outils proposée par Hibernate 3. Pour les autres outils, référez-vous à la page *http://www.hibernate.org/256.html.*

L'outillage d'Hibernate 3

Hibernate 3 est désormais doté d'un jeu complet d'outils. Un sous-ensemble du site Hibernate leur est dédié, à l'adresse *http://www.hibernate.org/255.html.*

Le minimum requis pour utiliser ces outils est de disposer d'Eclipse 3.1 M4, téléchargeable à l'adresse *http://www.eclipse.org.*

L'éditeur de mapping ne fonctionne qu'en présence de JBoss IDE *(http://www.jboss.org/products/jbosside).*

L'éditeur de mapping

Une fois les éléments précédents installés, les fichiers de mapping ainsi que le fichier de configuration global devraient être automatiquement ouverts par l'éditeur de mapping d'Hibernate 3. Si ce n'est pas le cas, il suffit de le redéfinir dans Windows, Preferences et Files associations, comme l'illustre la figure 9.5.

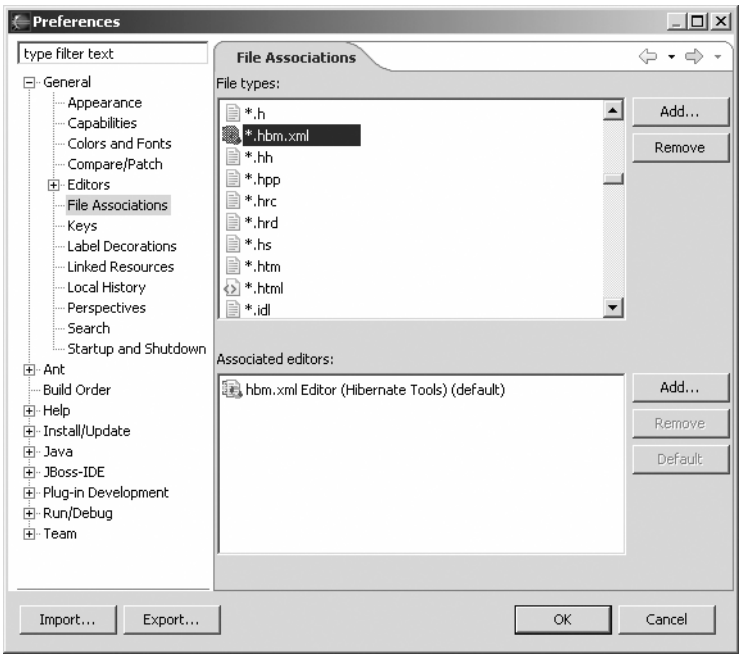


Figure 9.5
Paramétrage des éditeurs sous Eclipse

L'éditeur de mapping ne fait pas qu'appliquer un jeu de couleurs en fonction des éléments du fichier XML. Il permet aussi l'autocomplétion des nœuds, des noms de classes et des propriétés, comme l'illustre la figure 9.6.

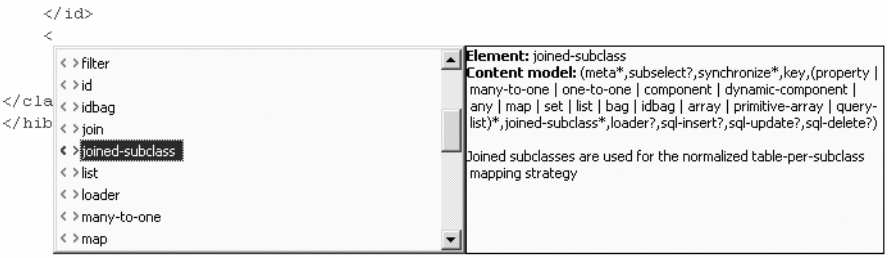


Figure 9.6
Autocomplétion des fichiers de mapping

Cet éditeur est d'une grande aide pour les débutants. Non seulement il accélère l'écriture des fichiers de mapping mais, grâce à son aide intuitive, il permet d'assimiler rapidement la structure des fichiers de mapping.

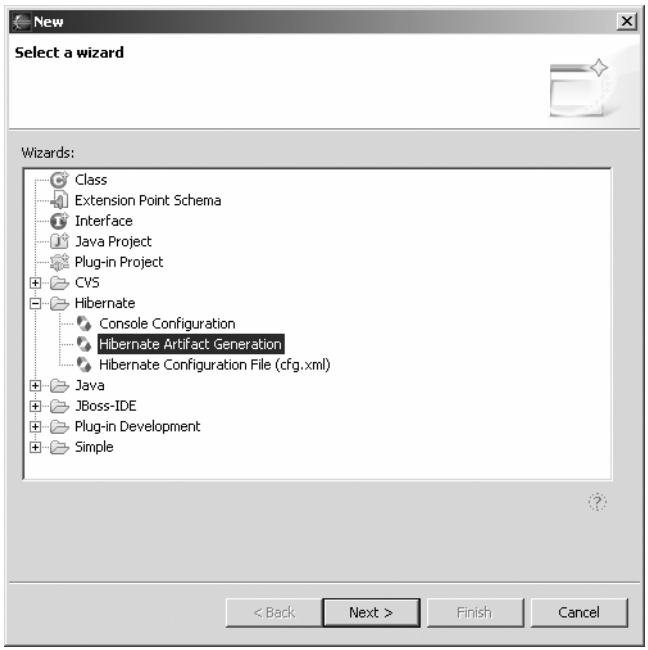
Dans ses prochaines évolutions, l'éditeur offrira les fonctionnalités suivantes :

- navigation entre les sources des classes et les fichiers de mapping ;
- possibilité de génération des sources relatives aux associations et propriétés ;
- participation à l'option de refactoring d'Eclipse.

Hibernate Console

Ce plug-in Eclipse propose trois assistants *(voir figure 9.7)* pour générer les fichiers de mapping depuis la base de données, configurer votre fichier **hibernate.cfg.xml** puis paramétrer Hibernate Console.

Figure 9.7
Assistants de la console Hibernate

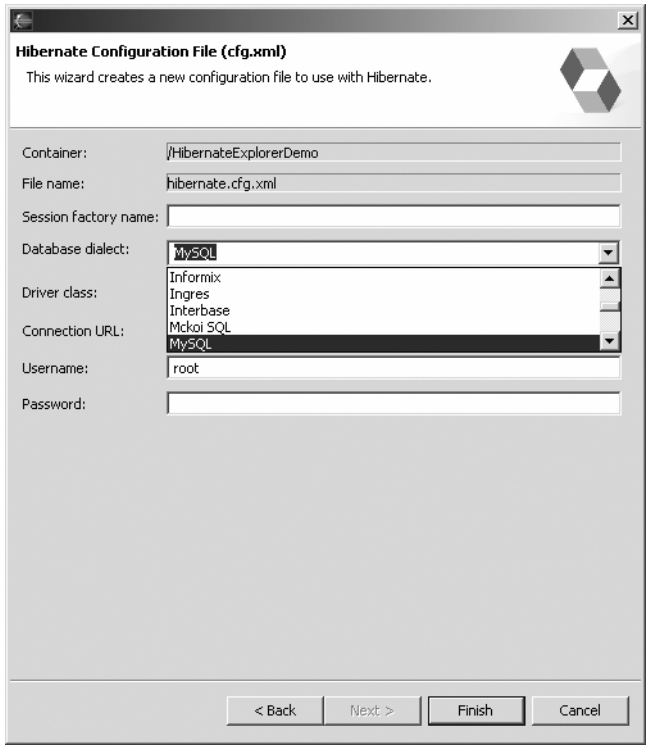


En choisissant l'assistant de fichier de configuration, vous pouvez rapidement spécifier les informations de configuration de haut niveau nécessaires à votre application, comme l'illustre la figure 9.8.

Par exemple, le choix du dialecte se fait simplement dans une liste déroulante.

La console Hibernate vous permet principalement de tester vos requêtes. Pour la configurer, choisissez Console Configuration, puis spécifiez le paramétrage nécessaire dans la fenêtre suivante *(voir figure 9.9).*

Figure 9.8
L'assistant Hibernate Configuration File



Prenez garde de ne pas spécifier les JAR relatifs à Hibernate dans la partie classpath. Vous ne devez renseigner que les pilotes JDBC et les classes persistantes. Une fois votre configuration établie, vous pouvez basculer dans la perspective Hibernate Console et créer une `SessionFactory`, comme l'illustre la figure 9.10.

La figure 9.11 illustre la représentation visuelle générée de vos classes persistantes, avec leur id et leurs propriétés.

L'intérêt principal de la console Hibernate réside dans la vue HQL Editor View, dans laquelle vous pouvez tester vos requêtes HQL *(voir figure 9.12)*.

À chaque exécution de requête, un onglet apparaît pour retourner les résultats de la requête. Il sera possible dès la version bêta de tester les requêtes à base de `Criteria` ainsi que les SQLQuery. L'éditeur HQL sera en outre enrichi d'un mécanisme de coloration et disposera d'une fonction d'autocomplétion. Chaque graphe d'objets retourné par la console pourra être exploré, et il sera possible d'analyser comment les stratégies de chargement sont appliquées.

Hibernate Console se verra enfin adjoindre des fonctions de visualisation du modèle relationnel (modèle physique de données) ainsi que du modèle de classes (UML).

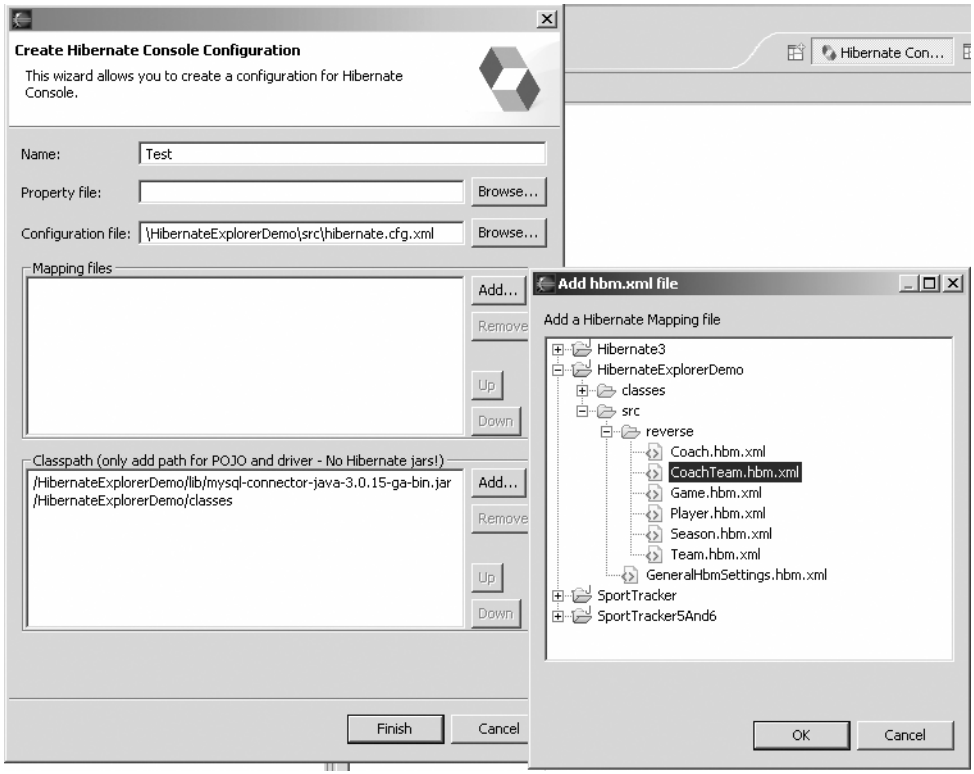


Figure 9.9
L'assistant Console Configuration

Figure 9.10
La perspective Hibernate Console

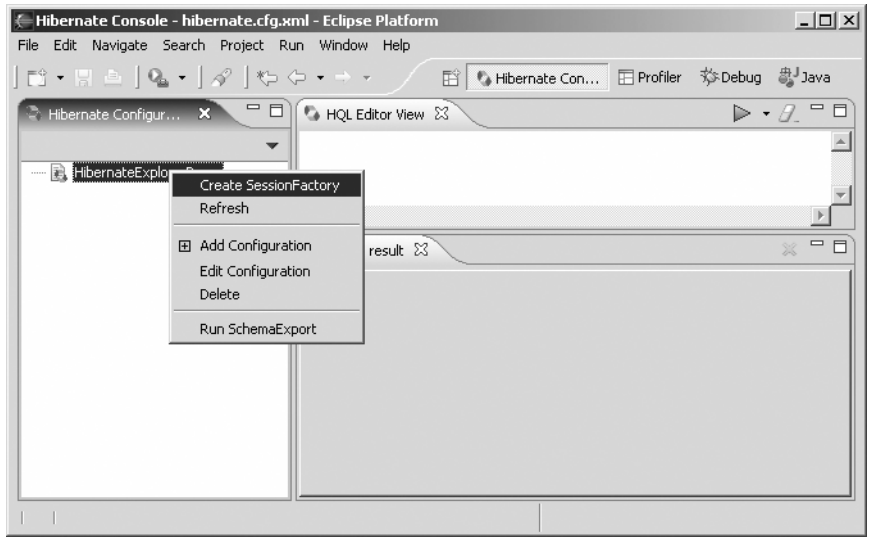


Figure 9.11
La vue Hibernate Configurations

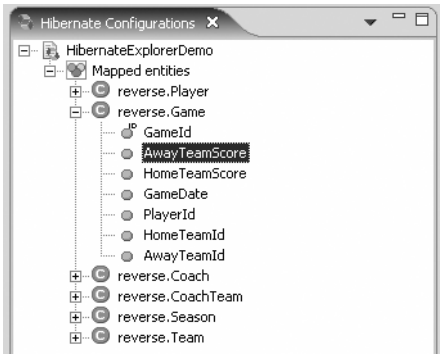
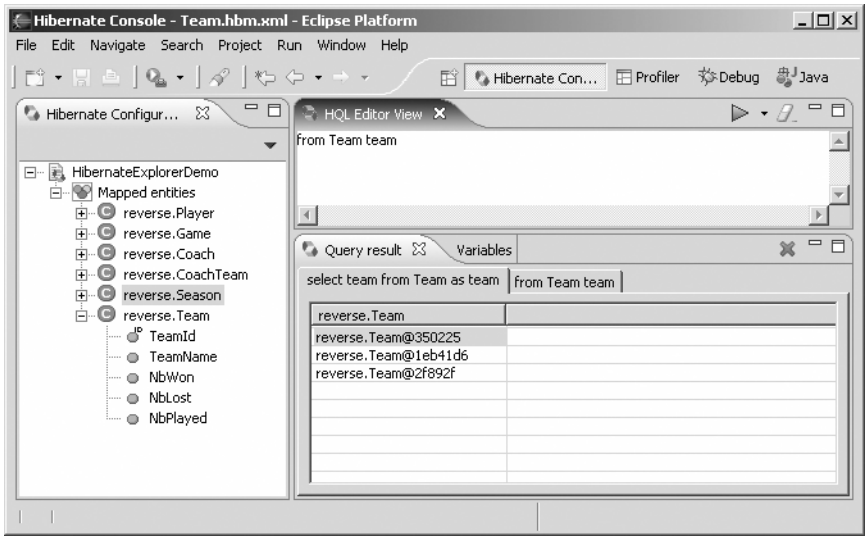


Figure 9.12
Test des requêtes



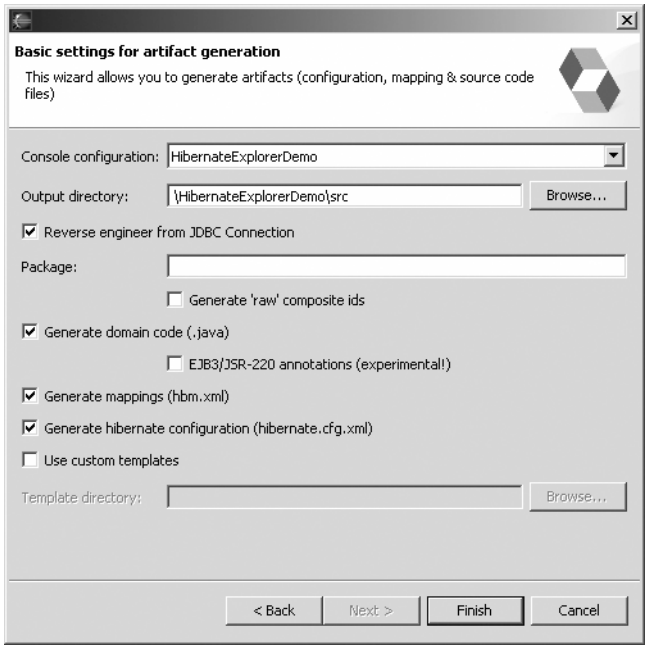
Hibernate Artifact Generation

Cette suite d'outils propose plusieurs assistants de génération. Il vous suffit de choisir Hibernate Artifact Generation depuis l'écran de la figure 9.7. L'assistant illustré à la figure 9.13 apparaît.

Il vous permet d'effectuer les tâches suivantes :

- reverse engineering depuis la connexion JDBC : génération de métadonnées depuis un schéma de base de données existant ;
- génération du fichier **hibernate.cfg.xml ;**
- génération de vos fichiers de mapping et de vos sources Java ;
- génération directe de vos sources Java complétés d'annotations (expérimental).

Figure 9.13
Options de génération



Il est d'ores et déjà prévu une personnalisation de la génération *via* des templates et un outil de migration automatique des fichiers de mapping vers les annotations.

Génération du schéma SQL avec SchemaExport

La suite d'outils proposée par Hibernate contient une fonction de génération de scripts DDL appelée SchemaExport.

Pour générer le schéma de votre base de données, il vous faut d'abord spécifier le dialecte de la base de données cible. Il existe plusieurs façons pratiques d'exécuter l'outil, et il est même possible d'appliquer le schéma généré directement à l'exécution.

Pour faire comprendre l'avantage de cette fonctionnalité, nous allons présenter une organisation de projet applicable depuis la fin de la conception jusque vers le milieu des développements.

La figure 9.14 illustre une organisation traditionnelle d'un projet pendant les phases de développement.

Chaque développeur est connecté à la même base de données. Chacun peut donc potentiellement utiliser les mêmes données de test. Ces données évoluent au fil du temps, de même que la structure relationnelle.

Cette organisation demande l'intervention régulière d'une personne responsable de la base de données pour la mise à jour des scripts de création de base de données ainsi que

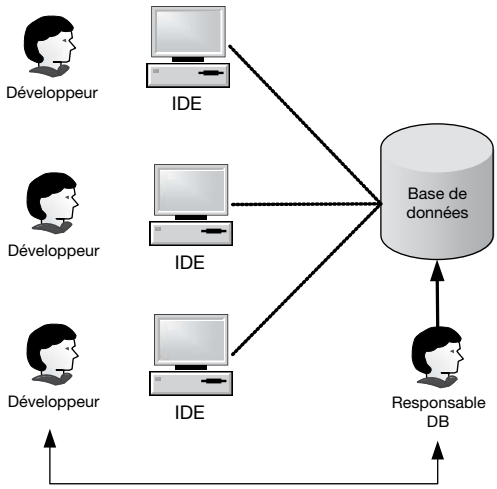


Figure 9.14
Exemple d'organisation traditionnelle d'une base de données partagée

pour l'injection de données de test. Les scripts d'injection de données de test sont à maintenir en fonction de l'évolution du schéma.

Les charges liées à cette maintenance de la base de données de développement sont d'autant plus importantes que la structure relationnelle est instable.

A priori, si votre projet s'articule autour d'Hibernate pour la persistance, il est probable qu'une phase de conception solide orientée objet a été identifiée. Si tel est le cas, l'outillage UML permet aisément de générer un squelette de fichier de mapping plus ou moins fin en fonction des stéréotypes et tagged-values mis en place.

AndroMDA propose une génération des fichiers de mapping depuis un export XMI de votre modélisation UML. Cela permet d'économiser une partie de la charge nécessaire à l'écriture des fichiers de mapping.

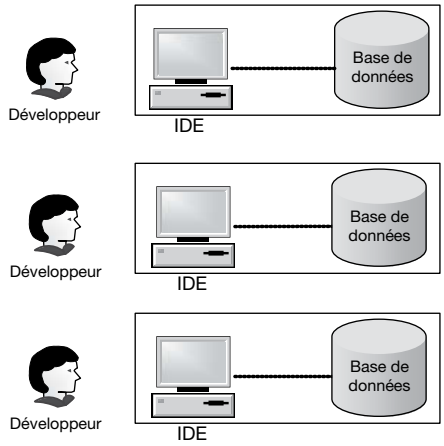
Les fichiers de mapping contiennent, par définition, énormément d'informations sur la structure du modèle relationnel. En exagérant, ils contiennent l'intégralité des éléments structurant des tables amenés à stocker les données métier de votre application.

Hibernate propose pour sa part un outil de génération de schéma. Avec une organisation différente *(voir figure 9.15),* vous pourrez améliorer la qualité et la productivité de vos applications pendant au moins la première moitié de la phase de réalisation.

Globalement, chacun des développeurs dispose d'une base de données locale. La meilleure solution est d'utiliser HSQLDB, qui présente les avantages suivants :

- Ne nécessite pas de licence.
- Est très simple à installer et utiliser avec Hibernate.
- Consomme très peu de ressources.

Figure 9.15
Proposition d'organisation de la base de données locale



Les avantages offerts par cette organisation sont multiples. Tout d'abord, le schéma de la base de développement est à jour tant que les fichiers de mapping restent le référentiel des informations relatives à la structure de données.

De plus, chaque développement devra coder ses propres instances de classes persistantes pour ses tests, ce qui garantit la présence systématique de tests. Si les classes évoluent, le développeur sera forcé de modifier ces fichiers de mapping pour que son application démarre. Par ailleurs, ces méthodes d'injection d'objets persistants seront, elles aussi, constamment à jour.

Les développeurs étant isolés les uns des autres, chacun est maître de ses objets de test. De même, ils ne peuvent être plusieurs à manipuler les mêmes données. À chaque démarrage de l'application, l'intégralité des jeux de test est injectée en base. Le développeur n'a donc plus à se soucier de leur validité.

Cette organisation peut être mise en place pendant la première moitié des développements sans problème. Une bonne gestion de configuration doit être utilisée pour fusionner les classes responsables de la gestion des instances de test.

Une fois la structure relationnelle suffisamment stable, il est très facile de migrer vers une base répondant à la cible de production. Il suffit juste de paramétrer le nouveau dialecte. Le schéma est alors généré dans un fichier SQL ou injecté directement dans la base de données.

Après cette phase d'initialisation de la base de données, le DBA peut intégrer l'équipe pour insérer les tables système, tuner les index et réaliser les autres paramétrages fins dont il est le seul garant. Bien entendu, le DBA doit être consulté en amont, dès la conception, pour définir les règles et principes à respecter.

Le cas le plus typique est le choix de la stratégie d'héritage. Il est important que le DBA participe aux choix en fonction des contraintes de performance et d'intégrité des données.

Enrichissement des fichiers de mapping pour la génération

Par défaut, les fichiers de mapping contiennent 80 % des éléments métier structurants de la base de données. Vous pouvez enrichir le schéma généré en utilisant certains tags dans les fichiers de mapping. Ces tags n'ont généralement pas de grand intérêt pendant l'exécution de votre application. Pour autant, si votre organisation se rapproche de celle décrite précédemment, il peut être intéressant d'enrichir au maximum les fichiers de mapping.

Le tableau 9.2 récapitule les diverses options utiles à la génération de schéma.

Tableau 9.2. Options de génération de schéma

Attribut	Valeur	Interprétation
`length`	Numérique	Précision d'une colonne (longueur ou décimal)
`not-null`	`true/false`	Spécifie que la colonne doit être non nulle.
`unique`	`true/false`	Spécifie que la colonne doit avoir une contrainte d'unicité.
`index`	Nom de l'index	Spécifie le nom d'un index (multicolonne).
`unique-key`	Nom de clé d'unicité	Spécifie le nom d'une contrainte d'unicité multicolonne.
`foreign-key`	Nom de la clé étrangère	Nom d'une contrainte de clé étrangère générée pour une association. Utilisez-la avec les éléments de mapping `<one-to-one>`, `<many-to-one>`, `<key>` et `<many-to-many>`. Notez que les extrémités `inverse="true"` ne sont pas prises en compte par SchemaExport.
`sql-type`	Type de la colonne	Surcharge le type par défaut (attribut de l'élément `<column>` uniquement).
`check`	Expression SQL	Créé une contrainte de vérification sur la table ou la colonne.

Exécution de SchemaExport

Avant d'aborder les différentes manières d'exécuter SchemaExport, le tableau 9.3 récapitule les options disponibles à l'exécution.

Tableau 9.3. Options d'exécution de SchemaExport

Option	Description
`quiet`	Ne pas écrire le script vers la sortie standard
`drop`	Supprime seulement les tables.
`text`	Ne pas exécuter sur la base de données
`output=my_schema.ddl`	Écrit le script DDL vers un fichier.
`config=hibernate.cfg.xml`	Lit la configuration Hibernate à partir d'un fichier XML.
`properties=hibernate.properties`	Lit les propriétés de la base de données à partir d'un fichier.
`format`	Formate proprement le SQL généré dans le script.
`delimiter=x`	Paramètre un délimiteur de fin de ligne pour le script.

Les sections qui suivent détaillent les différents moyens de déclencher SchemaExport.

☛ Déclenchement de SchemaExport *via* Ant

La tâche Ant vous permet d'automatiser la création du schéma ou de l'appeler de manière ponctuelle. Cela peut se révéler utile lors de la mise à jour du script SQL :

```
<project name="SportTracker" default="schemaexport" basedir=".">
  <property name="lib.dir" value="WEB-INF/lib"/>
  <property name="src.dir" value="WEB-INF/src"/>
  <property name="build.dir" value="WEB-INF/build"/>
  <property name="classes.dir" value="WEB-INF/classes"/>
  <property name="javac.debug" value="on"/>
  <property name="javac.optimize" value="off"/>

  <path id="hibernate-schema-classpath">
    <fileset dir="${lib.dir}">
      <include name="**/*.jar"/>
      <include name="**/*.zip"/>
    </fileset>
    <pathelement location="${classes.dir}" />
  </path>

  <path id="lib.class.path">
    <fileset dir="${lib.dir}">
      <include name="**/*.jar"/>
    </fileset>
    <pathelement path="${clover.jar}"/>
  </path>

  <!-- Tasks -->
…
  <target name="schemaexport">
    <taskdef name="schemaexport"
      classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
      classpathref="hibernate-schema-classpath"/>

    <schemaexport
      config="${classes.dir}/hibernate.cfg.xml"
      quiet="no"
      text="yes"
      drop="no"
      delimiter=";"
      output="schema-export.sql">
    </schemaexport>
  </target>
</project>
```

☛ Déclenchement de SchemaExport *via* JUnit

Afin de jouer les tests unitaires de votre application, il vous faut un jeu de fichiers de mapping précis ainsi que des données de test. Le schéma évoluant au fur et à mesure que votre projet avance, il est conseillé de s'appuyer sur des instances de classes de tests qui seront rendues persistantes après exécution de SchemaExport.

Pour ce faire, faites hériter vos classes de test unitaire de la classe ci-dessous :

```
package xxx;

import org.hibernate.tool.hbm2ddl.SchemaExport;
import utils.HibernateUtil;


public abstract class TestCase extends junit.framework.TestCase {
  public TestCase(String s) {
    super(s);
  }
  protected void runTest() throws Throwable {
    …
  }
  protected void setUp() throws Exception {
    super.setUp();
    SchemaExport ddlExport =
      new SchemaExport(HibernateUtil.getConfiguration());
    ddlExport.create(false, true);
  }
  protected void tearDown() throws Exception {
    super.tearDown();
    SchemaExport ddlExport =
      new SchemaExport(HibernateUtil.getConfiguration());
    ddlExport.drop(false, true);
  }
}
```

☛ Déclenchement de SchemaExport *via* un filtre de servlet

Si vous adoptez l'organisation de développement où chacun des développeurs dispose de sa base de données (par exemple HSQLDB) et que vous deviez réaliser une application Web, le filtre de servlet ci-dessous vous permettra de générer le schéma à chaque démarrage de l'application puis de le détruire à son arrêt.

Le filtre vous permet d'invoquer une méthode `init()`, qui aura en charge la persistance des instances de test :

```
public class DevFilter implements Filter {
  public void init(FilterConfig filterConfig) throws ServletException {
    SchemaExport ddlExport;
    try {
      ddlExport = new SchemaExport(HibernateUtil.getConfiguration());
      ddlExport.create(false, true);
```

```
    } catch (HibernateException e) {
      // TODO Auto-generated catch block
      …
    }
    // appel de la méthode de création d'instances persistantes
    init();
    }

  public void doFilter(ServletRequest request,
    ServletResponse response,
    FilterChain chain)
    throws IOException, ServletException {
  }

  public void destroy() {
    SchemaExport ddlExport;
    try {
      ddlExport = new SchemaExport(HibernateUtil.getConfiguration());
      ddlExport.drop(false, true);
    } catch (HibernateException e) {
      // TODO Auto-generated catch block
      …
    }
  }
  public static void init(){
    // créer les instances persistantes pour les tests unitaires
    }
}
```

Pour rappel, le paramétrage d'un filtre de servlet se fait dans le fichier **web.xml,** qui doit contenir les lignes suivantes :

```
<filter>
  <filter-name>DevFilter</filter-name>
  <filter-class>dev.utils.DevFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>DevFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
```

En résumé

La suite Hibernate Tools, comme les annotations, n'en est qu'à ses débuts. Les évolutions prochaines enrichiront les fonctionnalités proposées, ainsi que leur intégration à Eclipse. Il est d'ores et déjà prévu une déclinaison de l'outillage indépendante d'Eclipse ainsi que l'exécution par des tâches Ant.

SchemaExport était disponible dans les versions précédentes d'Hibernate. Cet outil, couplé à une organisation de développement où chaque développeur travaille avec sa

propre base de données, réduira notablement certaines charges de développement, comme la gestion et la maintenance de la base de données et des jeux de test par un DBA.

Extensions et intégration

Hibernate permet d'utiliser des extensions pour couvrir notamment la gestion des connexions JDBC ainsi que le cache de second niveau.

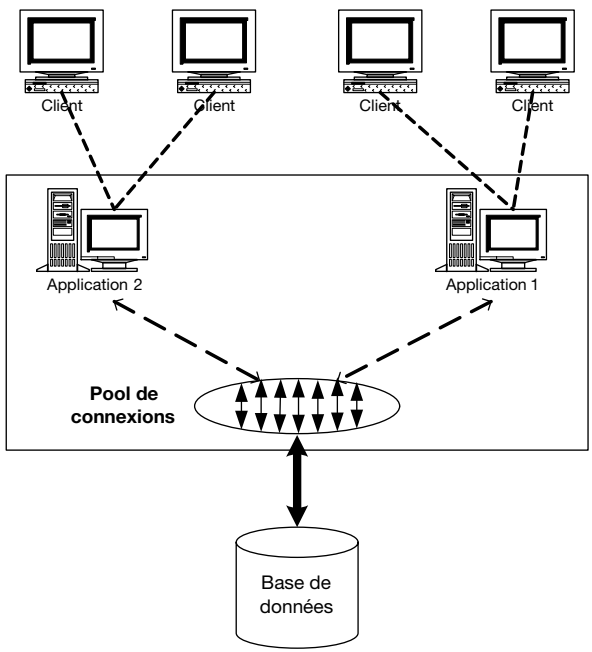
Hibernate n'ayant pas vocation a réinventer des composants existants, il propose dans sa livraison certains caches et pools de connexions éprouvés.

Utilisation d'un pool de connexions JDBC

Si vous ne disposez pas d'une datasource, vous allez devoir brancher un pool de connexions JBDC. Le principe du pool est illustré à la figure 9.16. La problématique inhérente est que l'établissement d'une connexion JDBC est très coûteux en terme de performances. Il ne faut donc pas que la demande d'accès aux données depuis le client final ouvre sa connexion au fil du temps.

Chaque connexion ouverte doit le rester du point de vue du pool, et non du client final. Lorsqu'elle n'est plus utilisée, elle reste disponible, prête à l'emploi pour les applications clientes du pool.

Figure 9.16
Principe d'utilisation du pool de connexions



Les pools de connexions offrent généralement un certain nombre de services, comme le contrôle du nombre maximal de connexions ouvertes ou de la validité des connexions ouvertes et le délai maximal de veille et de vie des connexions.

Hibernate se branche par défaut à C3P0 et Proxool, le support du pool de connexions DBCP étant terminé faute de stabilité. Si vous souhaitez brancher un autre pool de connexions, vous n'avez qu'à fournir une implémentation de l'interface `org.hibernate.connection.ConnectionProvider`, dont voici le source :

```
/**
 * A strategy for obtaining JDBC connections.
 * <br><br>
 * Implementors might also implement connection pooling.<br>
 * <br>
 * The <tt>ConnectionProvider</tt> interface is not intended to be
 * exposed to the application. Instead it is used internally by
 * Hibernate to obtain connections.<br>
 * <br>
 * Implementors should provide a public default constructor.
 *
 * @see ConnectionProviderFactory
 * @author Gavin King
 */
public interface ConnectionProvider {
 /**
 * Initialize the connection provider from given properties.
 * @param props <tt>SessionFactory</tt> properties
 */
public void configure(Properties props) throws HibernateException;
/**
 * Grab a connection
 * @return a JDBC connection
 * @throws SQLException
 */
public Connection getConnection() throws SQLException;
/**
 * Dispose of a used connection.
 * @param conn a JDBC connection
 * @throws SQLException
 */
public void closeConnection(Connection conn) throws SQLException;
/**
 * Release all resources held by this provider. JavaDoc requires a second sentence.
 * @throws HibernateException
 */
public void close() throws HibernateException;
}
```

Cette interface n'est pas des plus complexes à implémenter, les pools de connexions ne devant offrir que des méthodes de type fermeture et ouverture de connexion ainsi qu'un paramétrage minimal.

Pour en savoir plus Pour avoir la liste des paramètres disponibles pour le fichier **hibernate.cgf.xml,** reportez-vous à la javadoc de `org.hibernate.cfg.Environnement 'doc\api\org\hibernate\cfg\Environment.html`.

Les sections qui suivent montrent comment utiliser les pools livrés avec Hibernate.

C3P0

Le pool de connexions C3P0 présente deux atouts majeurs. Tout d'abord il est reconnu fiable puisque, en deux ans, aucun bogue réel n'a été déploré. Son autre avantage est qu'il propose des possibilités de paramétrage étendues.

Le tableau 9.4 recense les paramètres de `org.hibernate.cfg.Environnement` relatifs à C3P0 qui sont configurables.

Tableau 9.4. Paramètres configurables de *org.hibernate.cfg.Environnement* relatifs à C3P0

`C3P0_ACQUIRE_INCREMENT` Nombre de connexions acquises lorsque la taille du pool augmente
`C3P0_IDLE_TEST_PERIOD` Temps de veille avant qu'une connexion soit validée
`C3P0_MAX_SIZE` Taille maximale du pool
`C3P0_MAX_STATEMENTS` Taille maximale du cache des statements
`C3P0_MIN_SIZE` Taille minimale du pool
`C3P0_TIMEOUT` Durée de vie d'une connexion

En naviguant dans la javadoc, nous obtenons les paramètres que nous pouvons placer dans **hibernate.cfg.xml.**

D'abord les chaînes de caractères :

```
public static final String C3P0_ACQUIRE_INCREMENT
  = "hibernate.c3p0.acquire_increment"
public static final String C3P0_IDLE_TEST_PERIOD
  = "hibernate.c3p0.idle_test_period"
public static final String C3P0_MAX_SIZE
  = "hibernate.c3p0.max_size"
public static final String C3P0_MAX_STATEMENTS
  = "hibernate.c3p0.max_statements"
public static final String C3P0_MIN_SIZE
  = "hibernate.c3p0.min_size"
public static final String C3P0_TIMEOUT
  = "hibernate.c3p0.timeout"
```

Ensuite les lignes :

```
<property name="c3p0.acquire_increment">1</property>
<property name="c3p0.idle_test_period">100</property> <!-- seconds -->
<property name="c3p0.max_size">100</property>
<property name="c3p0.max_statements">0</property>
<property name="c3p0.min_size">10</property>
<property name="c3p0.timeout">100</property> <!-- seconds -->
```

Ces paramètres ne représentent qu'un sous-ensemble des possibilités de C3P0. Si vous souhaitez tirer parti de la totalité des paramètres, il vous suffit de préciser le `ConnectionProvider` et le paramètre `max_size` dans **hibernate.cfg.xml** puis de placer le fichier de propriété **c3p0.properties** dans le classpath de votre application.

Dans ce fichier, vous avez le loisir d'utiliser la totalité des paramètres décrits au tableau 9.5.

Tableau 9.5. Paramétrage du pool de connexions C3P0

Paramètre	Défaut	Définition
`initialPoolSize`	3	Taille initiale du pool
`minPoolSize`	3	Taille minimale du pool
`maxPoolSize`	15	Taille maximale du pool
`idleConnectionTestPeriod`	0	Une connexion non utilisée pendant ce délai (en secondes) sera déclarée comme `idle` (en veille).
`maxIdleTime`	0	Une connexion `idle` sera relâchée du pool au bout de ce délai (en secondes) d'inactivité.
`checkoutTimeout`	0	Délai (en millisecondes) d'attente au bout duquel une tentative d'acquisition de connexion sera considérée en échec. Un délai survient lorsque le pool a atteint sa taille maximale. Cet échec soulève une `SQLException`. `0` signifie un délai d'attente infini.
`acquireIncrement`	3	Nombre de connexion à ouvrir lorsque la taille du pool doit être augmentée.
`acquireRetryAttempts`	30	Nombre de tentatives d'acquisition d'une connexion avant abandon définitif. Si cette valeur est inférieure ou égale à `0`, il y aura un nombre infini de tentatives.
`acquireRetryDelay`	1000	Délai (en millisecondes) entre chaque essai
`autoCommitOnClose`	`false`	Si `false`, un rollback sur les transactions ouvertes d'une connexion qui se ferme sera exécuté. Si `true`, un commit sur les transactions ouvertes d'une connexion qui se ferme sera exécuté.

Pour plus de détails sur C3P0, reportez-vous à son site de référence, à l'adresse *http://sourceforge.net/projects/c3p0*.

Proxool

Proxool est un pool de connexions proposé par défaut lorsque vous téléchargez Hibernate. Il offre sensiblement les mêmes fonctionnalités que C3P0 et dispose d'une servlet, qui permet de visualiser l'état du pool de connexions.

Le tableau 9.6 récapitule le paramétrage d'un pool de connexions Proxool

Tableau 9.6. Javadoc relative à Proxool

`static String PROXOOL_EXISTING_POOL` Configurer Proxool à partir d'un pool existant
`static String PROXOOL_POOL_ALIAS` Alias à utiliser pour l'utilisation de Proxool (requis par `PROXOOL_EXISTING_POOL`, `PROXOOL_PROPERTIES` ou `PROXOOL_XML`)
`static String PROXOOL_PREFIX` Préfixe Proxool/Hibernate
`static String PROXOOL_PROPERTIES` Définition du fournisseur de configuration Proxool *via* un fichier de propriété (`/path/to/proxool.properties`)
`static String PROXOOL_XML` Définition du fournisseur de configuration Proxool *via* un fichier XML de propriété (`/path/to/proxool.xml`)

En naviguant dans la javadoc, vous obtenez les chaînes de caractères que vous pouvez placer dans **hibernate.cfg.xml :**

```
public static final String PROXOOL_EXISTING_POOL
  = "hibernate.proxool.existing_pool"
public static final String PROXOOL_POOL_ALIAS
  = "hibernate.proxool.pool_alias"
public static final String PROXOOL_PREFIX
  = "hibernate.proxool"
public static final String PROXOOL_PROPERTIES
  = "hibernate.proxool.properties"
public static final String PROXOOL_XML
  = "hibernate.proxool.xml"
```

Comme pour C3P0, il ne vous reste plus qu'à insérer ces paramètres dans **hibernate.cfg.xml.**

Si vous souhaitez tirer profit de l'ensemble du paramétrage *(voir tableau 9.7),* il vous faut externaliser le paramétrage dans un fichier de propriétés **proxool.properties.** Cependant, ce paramétrage est un peu plus délicat qu'avec C3P0 *(voir l'article dédié sur le wiki d'Hibernate, à l'adresse http://www.hibernate.org/222.html).*

Pour disposer de la servlet de visualisation des statistiques Proxool, paramétrez une application Web comme suit :

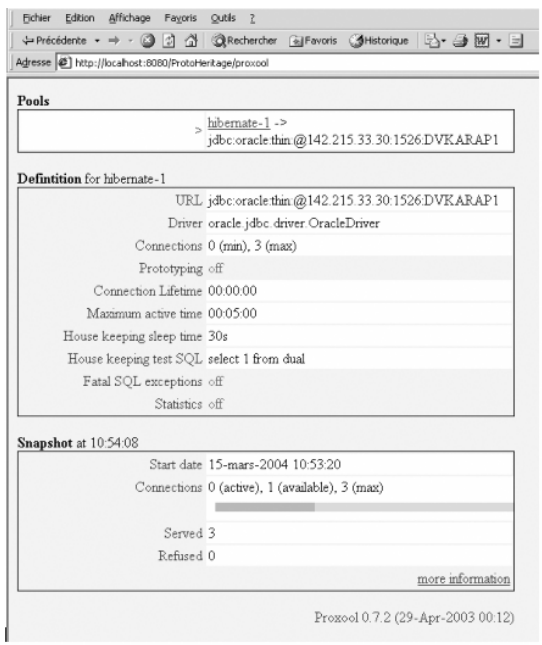
```
<servlet>
  <servlet-name>proxool</servlet-name>
  <servlet-class> org.logicalcobwebs.proxool.admin.servlet.AdminServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>proxool</servlet-name>
  <url-pattern>/proxool</url-pattern>
</servlet-mapping>
```

Tableau 9.7. Paramétrage de Proxool

Paramètre	Défaut	Définition
`house-keeping-sleep-time`	`30`	Définit le délai de test du thread (en secondes).
`house-keeping-test-sql`		Si le thread de test trouve une connexion `idle`, il exécute cette requête pour déterminer si la connexion est toujours valide. Cette requête doit être rapide à utiliser. Exemple : `select 1 from dual`.
`test-before-use`	`false`	Si `true`, chaque utilisation d'une connexion force un test de celle-ci. Si le test échoue, la connexion est abandonnée et une autre est acquise. Si toutes les connexions disponibles échouent, une nouvelle connexion est ouverte. Si celle-ci échoue aussi, une `SQLException` est levée.
`test-after-use`	`false`	Si `true`, chaque fermeture (la connexion est rendue au pool) entraîne un test de celle-ci. Si le test échoue, la connexion est abandonnée et une autre est acquise.
`maximum-connection-count`	`15`	Nombre maximal de connexions à la base de données.
`maximum-connection-lifetime`	`4 heures`	Durée de vie d'une connexion en millisecondes. Après ce délai, la connexion est détruite.
`maximum-new-connections`	`10`	Nombre maximal de connexions à la base de données
`maximum-active-time`	`5 minutes`	Durée de vie de la connexion. Si le thread de test tombe sur une connexion active depuis plus longtemps que ce délai, il la détruit. Cette valeur doit être supérieure au temps de réponse le plus long que vous prévoyez.
`trace`	`false`	Si `true`, le SQL sera logué (niveau debug) tout au long de l'exécution.
`maximum-connection-count`	`10`	Nombre maximal de connexions pouvant être ouvertes simultanément

La figure 9.17 illustre la servlet permettant de visionner les statistiques de Proxool.

Figure 9.17
Statistiques de Proxool



Pour plus d'informations sur Proxool, référez-vous au site de référence, à l'adresse *http://proxool.sourceforge.net.*

Utilisation du cache de second niveau

Le cache de premier niveau est la session Hibernate. Comme vous le savez désormais, la session a une durée de vie relativement courte et n'est pas partageable par plusieurs traitements.

Pour améliorer les performances, des caches de second niveau sont disponibles. N'allez toutefois pas croire que le cache de second niveau soit la solution ultime à vos problèmes de performances. Le cache relève en effet d'une problématique plus complexe à appréhender qu'il n'y paraît.

Gérer un cache demande des ressources et donc du temps serveur. Il demande encore plus de temps si votre environnement de production répartit la charge sur plusieurs serveurs, le temps réseau venant s'ajouter au temps machine.

Une règle simple à retenir est que moins le cache est manipulé en écriture, plus il est efficace. À ce titre, les données les plus propices à être mises en cache sont les données de référence, de paramétrage et celles rarement mises à jour par l'application. Il est évidemment inutile de mettre en cache des données rarement consultées. De bons exemples de données pour le cache sont les codes postaux, les pays ou les articles d'un catalogue qui ne serait mis à jour que trimestriellement.

Ayez toujours à l'esprit que l'utilisation du cache à l'exécution se produit lorsque Hibernate tente de récupérer une instance d'une classe particulière avec un id particulier. Ces prérequis ne sont donc pas respectés lorsque vous exécutez des requêtes et en obtenez les résultats par invocation de la méthode `query.list()`. Nous reviendrons sur l'utilisation spécifique du cache pour les requêtes.

Les caches livrés avec Hibernate

Les caches livrés avec Hibernate sont recensés au tableau 9.8. Comme pour les pools de connexions, il s'agit de projets à part entière, qui ne sont pas liés à Hibernate.

Les paramètres à prendre en compte lorsque vous choisissez un cache sont les suivants :

- Votre application fonctionne-t-elle en cluster ? Si oui, souhaitez-vous une stratégie par invalidation de cache ou par réplication ?
- Souhaitez-vous tirer profit du cache des requêtes ?
- Souhaitez-vous utiliser le support physique disque pour certaines conditions ?

Voici les définitions des différentes stratégies de cache :

- **Lecture seule (read-only).** Si votre application a besoin de lire mais ne modifie jamais les instances d'une classe, un cache read-only peut être utilisé. C'est la stratégie la plus simple et la plus performante. Elle est même parfaitement sûre dans un cluster.

- **Lecture/écriture (read-write).** Si l'application a besoin de mettre à jour des données, un cache read-write peut être approprié. Cette stratégie ne devrait jamais être utilisée si votre application nécessite un niveau d'isolation transactionnelle sérialisable. Si le cache est utilisé dans un environnement JTA, vous devez spécifier `hibernate.transaction.manager_lookup_class`, fournissant une stratégie pour obtenir le `TransactionManager` JTA. Dans d'autres environnements, vous devriez vous assurer que la transaction est terminée à l'appel de `session.close()` ou `session.disconnect()`. Si vous souhaitez utiliser cette stratégie dans un cluster, assurez-vous que l'implémentation de cache utilisée supporte le verrouillage.
- **Lecture/écriture non stricte (nonstrict-read-write).** Si l'application a besoin de mettre à jour les données de manière occasionnelle, c'est-à-dire s'il est très peu probable que deux transactions essaient de mettre à jour le même élément simultanément, et qu'une isolation transactionnelle stricte ne soit pas nécessaire, un cache nonstrict-read-write peut être approprié. Si le cache est utilisé dans un environnement JTA, vous devez spécifier `hibernate.transaction.manager_lookup_class`. Dans d'autres environnements, vous devriez vous assurer que la transaction est terminée à l'appel de `session.close()` ou `session.disconnect()`.
- **Transactionnelle (transactional).** La stratégie de cache transactional supporte un cache complètement transactionnel, par exemple, JBoss TreeCache. Un tel cache ne peut être utilisé que dans un environnement JTA, et vous devez spécifier `hibernate.transaction.manager_lookup_class`.

Tableau 9.8. Les caches livrés avec Hibernate

Cache	*Provider Class org.hibernate.cache.*	Type	Cluster Safe	Support cache de requête
Hashtable	`HashtableCacheProvider`	Mémoire		Oui
EHCache	`EhCacheProvider`	Mémoire, disque		Oui
OSCache	`OSCacheProvider`	Mémoire, disque		Oui
SwarmCache	`SwarmCacheProvider`	Clustérisé (IP multicast)	Oui (invalidation clustérisée)	
JBossTreeCache	`TreeCacheProvider`	Clustérisé (IP multicast), transactionnel	Oui (réplication)	Oui (synchronisation des horloges nécessaire)

Comme le montre le tableau 9.9, aucun cache ne supporte la totalité de ces stratégies.

Tableau 9.9. Support des stratégies de cache

Cache	Read-only	Nonstrict-read-write	Read-write	Transactionnal
Hashtable	Oui	Oui		
EHCache	Oui	Oui	Oui	Non
OSCache	Oui	Oui	Oui	Non
SwarmCache	Oui	Oui	Non	Non
JBossTreeCache	Oui	Non	Non	Oui

Mise en œuvre d'EHCache

EHCache est particulièrement simple à configurer. La première étape consiste à extraire **ehcache-x.x.jar** et à le placer dans le classpath de votre application.

Il faut ensuite spécifier dans **hibernate.cfg.xml** l'utilisation d'EHCache en ajoutant la ligne suivante :

```
<property name="hibernate.cache.provider_class">
  org.hibernate.cache.EhCacheProvider
</property>
```

EHCache nécessite un fichier de configuration **(ehcache.xml)** spécifique, dans lequel vous spécifiez, classe par classe, les paramètres principaux suivants :

- Nombre maximal d'instances qui peuvent être placées dans le cache.
- Délai maximal de veille, qui correspond à la durée maximale au terme de laquelle une instance non utilisée sera supprimée du cache.
- Durée de vie maximale, qui correspond à la durée, depuis la mise en cache, au bout de laquelle une instance, même fréquemment consultée, sera supprimée du cache.

Voici un exemple de fichier **ehcache.xml :**

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>

  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    overflowToDisk="true"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    diskPersistent="false"
    diskExpiryThreadIntervalSeconds="120"
  />

  <cache name="com.eyrolles.sportTracker.model.Team"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
  />
  <cache name="com.eyrolles.sportTracker.model.Player"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
  />
<cache name="com.eyrolles.sportTracker.model.Coach"
```

```
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="false"
  />
</ehcache>
```

Vous pouvez vérifier qu'EHCache est actif sur votre application en scrutant les traces au démarrage de votre application. Vous devriez retrouver les lignes suivantes :

```
INFO SettingsFactory:259 - Cache provider:
  org.hibernate.cache.EhCacheProvider
INFO SettingsFactory:186 - Second-level cache: enabled
```

Pour toute information supplémentaire concernant EHCache, consultez le site de référence, à l'adresse *http://ehcache.sourceforge.net/documentation/.*

Activation du cache pour les classes et collections

Vous venez de voir comment configurer globalement le cache au niveau applicatif en utilisant, par exemple, EHCache. Il ne vous reste plus qu'à définir la stratégie dans vos fichiers de mapping ou au niveau de **hibernate.cfg.xml.**

Cette définition s'effectue au niveau à la fois des classes et des collections, les deux notions étant distinctes du point de vue du cache.

Activation dans les fichiers de mapping

Un premier moyen pour définir la stratégie de cache consiste à utiliser l'élément XML `<cache/>` directement dans les fichiers de mapping.

Voici, par exemple, comment configurer les classes `Player`, `Team` et `Coach` :

```
<hibernate-mapping package="com.eyrolles.sportTracker.model">
  <class name="Player" table="PLAYER" >
    <cache usage="read-write"/>
    …
  </class>
</hibernate-mapping>
```

```
<hibernate-mapping package="com.eyrolles.sportTracker.model">
  <class name="Coach" table="COACH">
    <cache usage="read-write"/>
    …
  </class>
</hibernate-mapping>
```

Le dernier exemple suivant permet de visualiser comment déclarer la mise en cache d'une collection :

```
<hibernate-mapping package="com.eyrolles.sportTracker.model">
  <class name="Team" table="TEAM" >
    <cache usage="read-write"/>
    …
    <bag name="players" inverse="true" fetch="select"
      cascade="create,merge">
      <cache usage="read-write"/>
      <key column="TEAM_ID"/>
      <one-to-many class="Player" />
    </bag>
  </class>
</hibernate-mapping>
```

☛ Activation dans *hibernate.cfg.xml*

Une autre méthode pour déclarer la mise en cache de vos classes consiste à ajouter les déclarations dans le fichier **hibernate.cfg.xml.** Dans ce cas, il est bien sûr inutile d'ajouter les déclarations `<cache/>` dans les fichiers de mapping.

Dans **hibernate.cfg.xml,** les élément XML à utiliser sont les suivants :

- `<class-cache/>` et les attributs suivants :
  - `usage`, pour définir la stratégie ;
  - `class`, pour spécifier le nom de la classe ;
- `<collection-cache/>` et les attributs suivants :
  - `usage`, pour définir la stratégie ;
  - `collection`, pour spécifier le nom d'une collection.

Voici un exemple de déclaration de mise en cache :

```
<class-cache usage="read-write"
  class="com.eyrolles.sportTracker.model.Team"/>
<class-cache usage="read-write"
  class="com.eyrolles.sportTracker.model.Player"/>
<class-cache usage="read-write"
  class="com.eyrolles.sportTracker.model.Coach"/>
<collection-cache usage="read-write"
  collection="com.eyrolles.sportTracker.model.Team.players"/>
```

Exemples de comportements selon la stratégie retenue

Il est important de saisir les nuances qui existent entre les différentes stratégies.

Le code ci-dessous permet de mettre en œuvre le cache. En l'exécutant, nous nous rendons compte qu'aucune requête n'est exécutée à la récupération de l'entité déjà chargée par une session précédente :

```
tx = session.beginTransaction();
// première touche, mise en cache
```

```
Team t = (Team)session.get(Team.class,new Long(1));
tx.commit();
HibernateUtil.closeSession();

// nouvelle session
session = HibernateUtil.getSession();
tx = session.beginTransaction();

//récupération du cache
Team t2 = (Team)session.get(Team.class,new Long(1));
tx.commit();
```

L'idée est la suivante : une première session permet d'alimenter le cache, tandis qu'une seconde permet de vérifier qu'aucune requête n'est exécutée à la récupération de la même entité.

L'exemple de code suivant permet de mesurer les impacts d'une écriture sur une entité présente dans le cache :

```
tx = session.beginTransaction();
//mise en cache
Team t = (Team)session.get(Team.class,new Long(1));

// modification de l'instance
t.setName("another name");
tx.commit();
HibernateUtil.closeSession();

// nouvelle session
session = HibernateUtil.getSession();
tx = session.beginTransaction();
//récupération du cache ?
Team t2 = (Team)session.get(Team.class,new Long(1));
tx.commit();
```

Voici ce que nous observons dans les traces pour la stratégie read-only :

```
Hibernate: select team0_.TEAM_ID as TEAM1_0_, team0_.NB_LOST as NB2_0_0_,
  team0_.TEAM_NAME as TEAM3_0_0_, team0_.COACH_ID as COACH4_0_0_
  from TEAM team0_
  where team0_.TEAM_ID=?
15:57:17,139 ERROR ReadOnlyCache:42 - Application attempted to edit read only item:
com.eyrolles.sportTracker.model.Team#1
java.lang.UnsupportedOperationException: Can't write to a readonly object
at org.hibernate.cache.ReadOnlyCache.lock(ReadOnlyCache.java:43)
```

Une exception est soulevée. La modification d'une entité configurée pour un cache avec la stratégie read-only ne peut être modifiée.

Modifions donc la stratégie en la paramétrant en nonstrict-read-write, et observons les traces :

```
Hibernate: select team0_.TEAM_ID as TEAM1_0_, team0_.NB_LOST as NB2_0_0_,
  team0_.TEAM_NAME as TEAM3_0_0_, team0_.COACH_ID as COACH4_0_0_
  from TEAM team0_
where team0_.TEAM_ID=?
Hibernate: update TEAM
  set NB_LOST=?, TEAM_NAME=?, COACH_ID=?
  where TEAM_ID=?
Hibernate: select team0_.TEAM_ID as TEAM1_0_, team0_.NB_LOST as NB2_0_0_,
  team0_.TEAM_NAME as TEAM3_0_0_, team0_.COACH_ID as COACH4_0_0_
  from TEAM team0_
  where team0_.TEAM_ID=?
```

La modification est autorisée mais entraîne l'invalidation du cache pour l'entité modifiée. Le cache n'est pas mis à jour, l'entité y étant simplement supprimée, ce qui engendre un ordre SQL `SELECT` à la récupération suivante de l'entité.

Cette stratégie n'a aucun intérêt pour des entités fréquemment mise à jour.

Testons désormais la dernière stratégie, read-write :

```
Hibernate: select team0_.TEAM_ID as TEAM1_0_, team0_.NB_LOST as NB2_0_0_,
  team0_.TEAM_NAME as TEAM3_0_0_, team0_.COACH_ID as COACH4_0_0_
  from TEAM team0_
  where team0_.TEAM_ID=?
Hibernate: update TEAM
  set NB_LOST=?, TEAM_NAME=?, COACH_ID=?
  where TEAM_ID=?
```

Cette fois, le cache est mis à jour en même temps que la base de données.

Cette dernière stratégie est plus efficace pour les entités que l'on peut mettre à jour. Elle ne peut toutefois être utilisée que si le risque de modification concourante est nul.

Contrôler l'interaction avec le cache

Il est possible de contrôler l'interaction avec le cache au niveau de la session. Cela passe par l'invocation de la méthode `session.setCacheMode()`.

Les différents modes d'interaction sont les suivants :

- `CacheMode.NORMAL` : récupère les objets depuis le cache de second niveau et les place dans le cache de second niveau.
- `CacheMode.GET` : récupère les objets du cache de second niveau mais n'y accède pas en écriture, excepté lors de la mise à jour de données.
- `CacheMode.PUT` : ne récupère pas les objets depuis le cache de second niveau (lecture directe depuis la base de données) mais écrit les objets dans le cache de second niveau.
- `CacheMode.REFRESH` : ne récupère pas les objets depuis le cache de second niveau (lecture directe depuis la base de données) mais écrit les objets dans le cache de second niveau, tout en ignorant l'effet du paramètre `hibernate.use_minimal_puts`, ce qui force

le rafraîchissement du cache de second niveau pour toutes les données lues en base de données.

- `CacheMode.IGNORE` : la session n'interagit jamais avec le cache, excepté pour invalider des objets mis à jour.

Par exemple, le code suivant :

```
tx = session.beginTransaction();
//mise en cache
Team t = (Team)session.get(Team.class,new Long(1));
tx.commit();
HibernateUtil.closeSession();
session = HibernateUtil.getSession();
session.setCacheMode(CacheMode.IGNORE);
tx = session.beginTransaction();
//récupération du cache
Team t2 = (Team)session.get(Team.class,new Long(1));
tx.commit();
```

évite la lecture depuis le cache et réexécute la requête en base de données pour permettre la récupération de l'objet.

De même, l'exemple suivant :

```
tx = session.beginTransaction();
//mise en cache
session.setCacheMode(CacheMode.GET);
Team t = (Team)session.get(Team.class,new Long(1));
tx.commit();
HibernateUtil.closeSession();
session = HibernateUtil.getSession();
tx = session.beginTransaction();
//récupération du cache
Team t2 = (Team)session.get(Team.class,new Long(1));
tx.commit();
```

n'alimente pas le cache à l'exécution du premier appel à la méthode `session.get()`. Le second appel ne trouve donc pas l'instance demandée en cache et interroge à nouveau la base de données.

Le cache de requête

Nous avons vu que les classes et les collections étaient deux notions différentes du point de vue du cache. Il en va de même des requêtes, qui demandent des traitements spécifiques pour bénéficier du cache mais aussi pour être utilisées conjointement avec le cache des classes et collections.

Analysons le code suivant :

```
tx = session.beginTransaction();
//mise en cache
Team t = (Team)session.get(Team.class,new Long(1));
```

```
tx.commit();
HibernateUtil.closeSession();
session = HibernateUtil.getSession();
tx = session.beginTransaction();
// exécution d'une requête
Team t2 = (Team)session.createQuery("from Team team where team = 1")
  .list()
  .get(0);
// exécution de la même requête
Team t3 = (Team)session.createQuery("from Team team where team = 1")
  .list()
  .get(0);
tx.commit();
```

Ce code produit les traces suivantes :

```
Hibernate: select team0_.TEAM_ID as TEAM1_0_, team0_.NB_LOST as NB2_0_0_,
team0_.TEAM_NAME as TEAM3_0_0_, team0_.COACH_ID as COACH4_0_0_ from TEAM team0_ where
team0_.TEAM_ID=?
Hibernate: select team0_.TEAM_ID as TEAM1_, team0_.NB_LOST as NB2_0_, team0_.TEAM_NAME
as TEAM3_0_, team0_.COACH_ID as COACH4_0_ from TEAM team0_ where (team0_.TEAM_ID=1 )
Hibernate: select team0_.TEAM_ID as TEAM1_, team0_.NB_LOST as NB2_0_, team0_.TEAM_NAME
as TEAM3_0_, team0_.COACH_ID as COACH4_0_ from TEAM team0_ where (team0_.TEAM_ID=1 )
```

La première requête correspond à l'appel de `session.get()`, tandis que les deux dernières correspondent à l'exécution des requêtes HQL. Lorsque l'instance est en cache, l'ensemble des informations est retourné à l'appel de `query.list()`. Ce comportement est attendu.

Afin de tirer profit du cache à partir des requêtes, il faut invoquer la méthode `query.iterate()`, qui exécute la requête pour obtenir les identifiants des instances retournées par la requête. Une fois que l'itération accède à un id particulier, le cache de second niveau est interrogé. Si ce dernier ne contient pas l'instance souhaitée, une requête est exécutée pour récupérer les informations relatives à cette instance.

En conclusion, il n'est pas recommandé d'utiliser la méthode `iterate()`. Dans le cas général où les instances retournées par la requête ne sont pas présentes dans le cache, `iterate()` provoque $n + 1$ requêtes, ce qui impacte considérablement les performances.

Pour mettre en cache une requête, utilisez simplement la méthode `query.setCacheable(true)` :

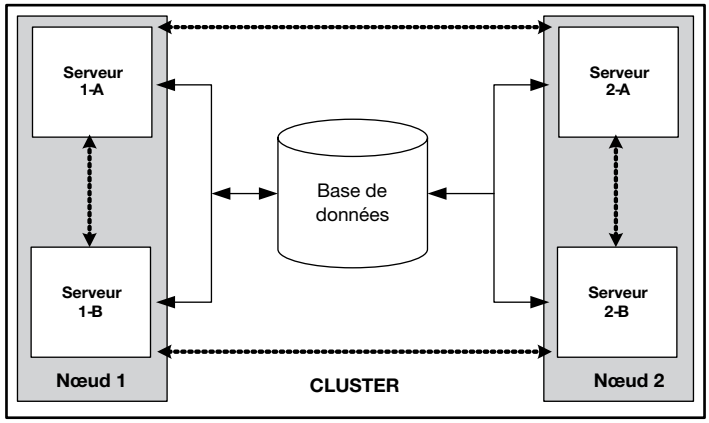
```
Query q1 =  session.createQuery("from Team team where team = :teamId")
  .setParameter("teamId", new Long(1))
  .setCacheable(true);
```

Notez que le cache de requête met en cache non pas l'état de chaque entité du résultat mais les valeurs des identifiants et les résultats de type valeur. C'est pourquoi le cache de requête est généralement utilisé en association avec le cache de second niveau, comme nous venons de le voir.

Principe de réplication de cache avec JBossCache

La figure 9.18 illustre la nécessité d'un cache par réplication comme JBossCache.

Figure 9.18
Configuration en cluster



Cela n'est qu'un exemple simplifié d'une architecture de déploiement. Votre application peut être déployée sur plusieurs machines physiques (nœuds), chacune de ces machines pouvant héberger des serveurs (logiques à l'inverse de physiques) pour tirer profit de fonctionnalités propres aux dernières générations de machines. L'ensemble de ces serveurs peut être client d'une même base de données.

Si vous souhaitez utiliser un cache, celui-ci sera dédié à un seul serveur. La validité des instances contenues dans le cache est primordiale. Une modification d'instance dans le cache du serveur 1-A doit être répliquée aux caches des autres serveurs ou invalidée. Il y a donc nécessité de communication entre tous les acteurs.

Il est important de bien vous assurer que le trafic réseau engendré par un tel environnement ne contrarie pas les apports du cache.

Pour mettre en œuvre JBossCache, vous aurez besoin des fichiers suivants :

- **concurrent-1.3.2.jar**
- **jboss-cache.jar**
- **jboss-common.jar**
- **jboss-jmx.jar**
- **jboss-remoting.jar**
- **jboss-system.jar**
- **jgroups-2.2.7.jar**

Pour vérifier que vos application utilisent bien JBossCache, scrutez les traces et recherchez les lignes suivantes :

```
INFO SettingsFactory:259 - Cache provider:
  org.hibernate.cache.TreeCacheProvider
INFO SettingsFactory:186 - Second-level cache: enabled
```

Malheureusement, du fait des concepts réseau, JBossCache est plus complexe à configurer qu'EHCache. Pour en savoir plus, référez-vous à la documentation du produit, à l'adresse *http://docs.jboss.org/jbcache/TreeCache.html*.

Utiliser un autre cache

Comme pour les pools de connexions, il est possible de brancher n'importe quel système de cache, pourvu que vous fournissiez une implémentation de l'interface `org.hibernate.cache.CacheProvider` :

```
/**
 * Support for pluggable caches.
 * @author Gavin King
 */
public interface CacheProvider {

/**
 * Configure the cache
 *
 * @param regionName the name of the cache region
 * @param properties configuration settings
 * @throws CacheException
 */
public Cache buildCache(String regionName, Properties properties) throws
CacheException;

/**
 * Generate a timestamp
 */
public long nextTimestamp();

/**
 * Callback to perform any necessary initialization of the underlying cache
implementation
 * during SessionFactory construction.
 *
 * @param properties current configuration settings.
 */
public void start(Properties properties) throws CacheException;

/**
 * Callback to perform any necessary cleanup of the underlying cache implementation
 * during SessionFactory.close().
 */
```

```
public void stop();

}
```

Vous pouvez prendre exemple sur `org.hibernate.cache.EHCacheProvider` pour comprendre comment le `CacheProvider` fait le lien entre Hibernate et le système de cache à utiliser.

Visualiser les statistiques

Dans Hibernate 3, la `SessionFactory`, que nous avons présentée au chapitre 2, propose un ensemble complet de statistiques. Pour y accéder, invoquez `sessionFactory.getStatistics()`.

Le code suivant a déjà été utilisé pour montrer comment utiliser le cache de second niveau. Complétons-le afin d'obtenir les statistiques :

```
tx = session.beginTransaction();
//mise en cache
Team t = (Team)session.get(Team.class,new Long(1));
tx.commit();
HibernateUtil.closeSession();
session = HibernateUtil.getSession();
tx = session.beginTransaction();
//récupération du cache
Team t2 = (Team)session.get(Team.class,new Long(1));
tx.commit();
Statistics stats = HibernateUtil.getSessionFactory().getStatistics();
```

Regardez au débogueur de votre IDE préféré à quoi ressemble `stats` *(voir figure 9.19).*

`stats` propose un large éventail de statistiques, notamment, pour notre exemple, les suivants :

- Le premier appel de `get()` interroge tout d'abord le cache de second niveau sans succès (`secondLevelCacheMissCount +1`).
- Il provoque l'exécution d'une requête en base de données.
- À la récupération des données, le cache de second niveau est alimenté (`secondLevelCachePutCount + 1`).
- Le second appel de `get()` interroge à nouveau le cache de second niveau et récupère l'instance demandée sans interroger la base de données (`secondLevelCacheHitCount + 1`).

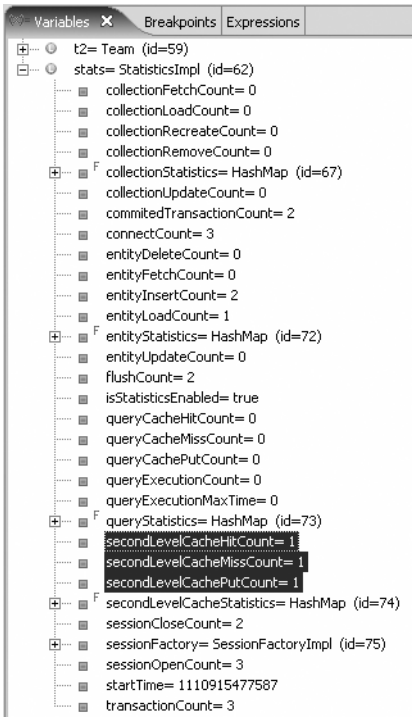
Il s'agit là des statistiques globales relatives au cache de second niveau. Les statistiques globales relatives aux requêtes, aux entités et aux collections sont aussi disponibles, par exemple, *via* les méthodes `stats.getQueryCacheMissCount()`, `stats.getEntityDeleteCount()` ou `stats.getCollectionLoadCount()`. La méthode `stats.getFlushCount()` permet de mesurer le nombre de fois où le flush a été réalisé. Par défaut, la session exécute le

flush automatiquement lorsqu'elle en a besoin. Grâce à cette méthode, vous pouvez contrôler à quelle fréquence et à quelles occasions le flush s'exécute.

Pour obtenir un niveau de détails plus fin sur les aspects du cache de second niveau, vous pouvez invoquer la méthode `getSecondLevelCacheStatistics(String regionName)` sur `stats`. La même finesse de statistiques est fournie pour les collections (`getCollectionStatistics(String role)`), les entités (`getEntityStatistics(String entityName)`) et les requêtes (`getQueryStatistics(String queryString)`).

Pour la liste complète des statistiques disponibles, référez-vous à la javadoc du package `org.hibernate.stat`. Hibernate peut être configuré pour remonter les statistiques *via* JMX.

Figure 9.19
Objet statistique



En résumé

Si l'utilisation d'un pool de connexions n'est pas complexe, il n'en va pas de même pour le cache de second niveau.

En effet, vous avez vu que l'utilisation d'un cache n'était pas la solution magique aux problèmes de performances. Il s'agit d'un élément d'architecture technique à part entière. Un cache configuré à la hâte pourrait affecter les performances, rendre inconsistantes les données, voire rendre instable votre application.

Si vous observez des problèmes de performances, commencez par en analyser la source. Le plus souvent, une optimisation des requêtes sera plus adaptée. Paramétrer finement un cache ne peut réellement se faire qu'en analysant le comportement de votre application une fois que celle-ci est en production. Ce n'est qu'à ce moment que vous êtes à même de traduire le comportement des utilisateurs en nombre d'instances récurrentes, en taux de modification ou encore en nombre de requêtes les plus utilisées.

Conclusion

Ce dernier chapitre vous a montré comment accroître votre productivité en utilisant l'outillage fourni par Hibernate, que ce soit pour gérer vos métadonnées, tester vos requêtes HQL ou générer le schéma de base de données.

Vous êtes aussi à même de brancher un pool de connexions ou même un cache de second niveau, tout en sachant que la mise en place d'un cache est une opération qui demande une étude préalable.

Gardez à l'esprit que l'outillage et les annotations sont en pleine évolution, et consultez régulièrement le site d'Hibernate pour découvrir les nouveautés.