Documentation de référence d'Hibernate

Version: 2.1.8

Table des matières

Préface

WARNING! This is a translated version of the English Hibernate reference documentation. The translated version might not be up to date! However, the differences should only be very minor. Consult the English reference documentation if you are missing information or encounter a translation error. If you like to contribute to a particular translation, contact us on the Hibernate developer mailing list.

Traducteur(s): Anthony Patricio <anthony@hibernate.org>, Emmanuel Bernard <emmanuel@hibernate.org>, Rémy Laroche, Bassem Khadige, Stéphane Vanpoperynghe

Travailler dans les deux univers que sont l'orienté objet et la base de données relationnelle peut être lourd et consommateur en temps dans le monde de l'entreprise d'aujourd'hui. Hibernate est un outil de mapping objet/relationnel pour le monde Java. Le terme mapping objet/relationnel (ORM) décrit la technique consistant à faire le lien entre la représentation objet des données et sa représentation relationnelle basé sur un schéma SQL.

Non seulement, Hibernate s'occupe du transfert des classes Java dans les tables de la base de données (et des types de données Java dans les types de données SQL), mais il permet de requêter les données et propose des moyens de les récupérer. Il peut donc réduire de manière significative le temps de développement qui aurait été dépensé autrement dans une manipulation manuelle des données via SQL et JDBC.

Le but d'Hibernate est de libérer le développeur de 95 pourcent des tâches de programmation liées à la persistence des données communes. Hibernate n'est probablement pas la meilleure solution pour les applications centrées sur les données qui n'utilisent que les procédures stockées pour implémenter la logique métier dans la base de données, il est le plus utile dans les modèles métier orientés objets dont la logique métier est implémentée dans la couche Java dite intermédiaire. Cependant, Hibernate vous aidera à supprimer ou à encapsuler le code SQL spécifique à votre base de données et vous aidera sur la tâche commune qu'est la transformation des données d'une représentation tabulaire à une représentation sous forme de graphe d'objets.

Si vous êtes nouveau dans Hibernate et le mapping Objet/Relationnel voire même en Java, suivez ces quelques étapes :

1. Lisez Chapitre 1, *Exemple simple utilisant Tomcat*, c'est un tutoriel de 30 minutes utilisant Tomcat.

2. Lisez Chapitre 2, *Architecture* pour comprendre les environnements dans lesquels Hibernate peut être utilisé.

3. Regardez le répertoire `eg` de la distribution Hibernate, il contient une application simple et autonome. Copiez votre pilote JDBC dans le répertoire `lib/` et éditez `src/hibernate.properties`, en positionnant correctement les valeurs pour votre base de données. A partir d'une invite de commande dans le répertoire de la distribution, tapez `ant eg` (cela utilise Ant), ou sous Windows tapez `build eg`.

4. Faîtes de cette documentation de référence votre principale source d'information. Pensez à lire *Hibernate in Action* (http://www.manning.com/bauer) si vous avez besoin de plus d'aide avec le design d'applications ou si vous préférez un tutoriel pas à pas. Visitez aussi http://caveatemptor.hibernate.org et téléchargez l'application exemple pour Hibernate in Action.

5. Les questions les plus fréquemment posées (FAQs) trouvent leur réponse sur le site web Hibernate.

6. Des démos, exemples et tutoriaux de tierces personnes sont référencés sur le site web Hibernate.

7. La zone communautaire (Community Area) du site web Hibernate est une bonne source d'information sur les design patterns et sur différentes solutions d'intégration d'Hibernate (Tomcat, JBoss, Spring

Framework, Struts, EJB, etc...).

Si vous avez des questions, utilisez le forum utilisateurs du site web Hibernate. Nous utilisons également l'outil de gestion des incidents JIRA pour tout ce qui est rapports de bogue et demandes d'évolution. Si vous êtes intéressé par le développement d'Hibernate, joignez-vous à la liste de diffusion de développement.

Le développement commercial, le support de production et les formations à Hibernate sont proposés par JBoss Inc (voir http://www.hibernate.org/SupportTraining/). Hibernate est un projet de la suite de produits Open Source Professionels JBoss.

Chapitre 1. Exemple simple utilisant Tomcat

1.1. Vos débuts avec Hibernate

Ce tutoriel détaille la mise en place d'Hibernate 2.1 avec le conteneur de servlet Apache Tomcat sur une application web. Hibernate est prévu pour fonctionner à la fois dans un environnement managé tel que proposé par tous les plus grands serveurs d'applications J2EE, mais aussi dans les applications Java autonomes. Bien que le système de base de données utilisé dans ce toturiel soit PostgreSQL 7.3, le support d'autres bases de données n'est qu'une question de configuration du dialecte SQL d'Hibernate.

Premièrement, nous devons copier toutes les bibliothèques nécessaires à l'installation dans Tomcat. Utilisant un contexte web séparé (`webapps/quickstart`) pour ce tutoriel, nous devons faire attention à la fois au chemin vers des bibliothèques globales (`TOMCAT/common/lib`) et au chemin du classloader contextuel de la webapp dans `webapps/quickstart/WEB-INF/lib` (pour les fichiers JAR) et `webapps/quickstart/WEB-INF/classes`. On se réfèrera aux deux niveaux de classloader que sont le classloader de classpath global et de classpath contextuel de la webapp.

Maintenant, copions les bibliothèques dans les deux classpaths :

1. Copiez le pilote JDBC de la base de données dans le classpath global. C'est nécessaire à l'utilisation du pool de connexions DBCP qui vient avec Tomcat. Hibernate utilise les connexions JDBC pour exécuter les ordres SQL sur la base de données, donc vous devez soit fournir les connexions JDBC poolées, soit configurer Hibernate pour utiliser l'un des pools nativement supportés (C3P0, Proxool). Pour ce tutoriel, copiez la blbliothèque `pg73jdbc3.jar` (pour PostgreSQL 7.3 et le JDK 1.4) dans le classpath global. Si vous voulez utiliser une base de données différente, copiez simplement le pilote JDBC approprié.

2. Ne copiez jamais autre chose dans le classpath global de Tomcat ou vous auriez des problèmes avec divers outils tels que log4j, commons-logging, et d'autres. Utilisez toujours le classpath contextuel de la webapp propre à chaque application, et donc copiez les bibliothèques dans `WEB-INF/lib`, puis copiez vos propres classes ainsi que les fichiers de configuration/de propriété dans `WEB-INF/classes`. Ces deux répertoires sont, par définition de la spécification J2EE, dans le classpath contextuel de la webapp.

3. Hibernate se présente sous la forme d'une blbliothèque JAR. Le fichier `hibernate2.jar` doit être copié dans le classpath contextuel de la webapp avec les autres classes de l'application. Hibernate a besoin de quelques bibliothèques tierces à l'exécution, elles sont embarquées dans la distribution Hibernate et se trouvent dans le répertoire `lib/` ; voir Tableau 1.1, « Bibliothèques tierces nécessaires à Hibernate ». Copiez les bibliothèques tierces requises dans le classpath de contexte.

Tableau 1.1. Bibliothèques tierces nécessaires à Hibernate

Bibliothèque	Description
dom4j (requise)	Hibernate utilise dom4j pour lire la configuration XML et les fichiers XML de métadonnées du mapping.
CGLIB (requise)	Hibernate utilise cette bibliothèque de génération de code pour étendre les classes à l'exécution (en conjonction avec la réflexion Java).
Commons Collections, Commons Logging (requises)	Hibernate utilise diverses bibliothèques du projet Apache Jakarta Commons.
ODMG4 (requise)	Hibernate est compatible avec l'interface de gestion de la persistance

Bibliothèque	Description
	telle que définie par l'ODMG. Elle est nécessaire si vous voulez mapper des collections même si vous n'avez pas l'intention d'utiliser l'API de l'ODMG. Nous ne mappons pas de collections dans ce tutoriel, mais, quoi qu'il arrive c'est une bonne idée de copier ce JAR.
EHCache (requise)	Hibernate peut utiliser diverses implémentations de cache de second niveau. EHCache est l'implémentation par défaut (tant qu'elle n'est pas changée dans le fichier de configuration).
Log4j (optionnelle)	Hibernate utilise l'API Commons Logging, qui peut utiliser log4j comme mécanisme de log sous-jacent. Si la bibliothèque Log4j est disponible dans le classpath, Commons Logging l'utilisera ainsi que son fichier de configuration `log4j.properties` récupéré depuis le classpath. Un exemple de fichier de propriétés pour log4j est embarqué dans la distribution d'Hibernate. Donc, copiez `log4j.jar` et le fichier de configuration (qui se trouve dans `src/`) dans le classpath contextuel de la webapp si vous voulez voir ce que fait Hibernate pour vous.
Nécessaire ou pas ?	Jetez un coup d'oeil à `lib/README.txt` de la distribution d'Hibernate. C'est une liste à jour des bibliothèques tierces distribuées avec Hibernate. Vous y trouverez toutes les bibliothèques listées et si elles sont requises ou optionnelles.

Nous allons maintenant configurer le pool de connexions à la base de données à la fois dans Tomcat mais aussi dans Hibernate. Cela signifie que Tomcat proposera des connexions JDBC poolées (en s'appuyant sur son pool DBCP), et qu'Hibernate demandera ces connexions à travers le JNDI. Tomcat proposant l'accès au pool de connexions via JNDI, nous ajoutons la déclaration de ressource dans le fichier de configuration principal de Tomcat (`TOMCAT/conf/server.xml`) :

```
<Context path="/quickstart" docBase="quickstart">
    <Resource name="jdbc/quickstart" scope="Shareable" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/quickstart">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <!-- paramètres de connexion DBCP à la base de données -->
        <parameter>
            <name>url</name>
            <value>jdbc:postgresql://localhost/quickstart</value>
        </parameter>
        <parameter>
            <name>driverClassName</name><value>org.postgresql.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>quickstart</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value>secret</value>
        </parameter>

        <!-- options du pool de connexion DBCP -->
        <parameter>
            <name>maxWait</name>
            <value>3000</value>
        </parameter>
        <parameter>
```

```
            <name>maxIdle</name>
            <value>100</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>10</value>
        </parameter>
    </ResourceParams>
</Context>
```

Le contexte web que l'on a configuré dans cet exemple se nomme `quickstart`, son répertoire de base étant `TOMCAT/webapp/quickstart`. Pour accéder aux servlets, appeler l'URL `http://localhost:8080/quickstart` à partir de votre navigateur (après avoir bien entendu ajouté le nom de votre servlet et l'avoir lié dans votre fichier `web.xml`). Vous pouvez également commencer à créer une servlet simple qui possède une méthode `process()` vide.

Tomcat utilise le pool de connexions DBCP avec sa configuration et fournit les `Connection`s JDBC poolées à travers l'interface JNDI à l'adresse `java:comp/env/jdbc/quickstart`. Si vous éprouvez des problèmes pour faire fonctionner le pool de connexions, référez-vous à la documentation Tomcat. Si vous avez des messages de type exception du pilote JDBC, commencez par configurer le pool de connexions JDBC sans Hibernate. Des tutoriels sur Tomcat et JDBC sont disponibles sur le Web.

La prochaine étape consiste à configurer Hibernate pour utiliser les connexions du pool attaché au JNDI. Nous allons utiliser le fichier de configuration XML d'Hibernate. L'approche basique utilisant le fichier .properties est équivalente fonctionnellement, mais n'offre pas d'avantage. Nous utiliserons le fichier de configuration XML parce que c'est souvent plus pratique. Le fichier de configuration XML est placé dans le classpath contextuel de la webapp (`WEB-INF/classes`), sous le nom `hibernate.cfg.xml` :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>

        <!-- fichiers de mapping -->
        <mapping resource="Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Le fichier de configuration montre que nous avons stoppé la log des commandes SQL, positionné le dialecte SQL de la base de données utilisée, et fournit le lien où récupérer les connexions JDBC (en déclarant l'adresse JNDI à laquelle est attachée le pool de source de données). Le dialecte est un paramètrage nécessaire du fait que les bases de données diffèrent dans leur interprétation du SQL "standard". Hibernate s'occupe de ces différences et vient avec des dialectes pour toutes les bases de données les plus connues commerciales ou open sources.

Une `SessionFactory` est un concept Hibernate qui représente un et un seul entrepôt de données ; plusieurs bases de données peuvent être utilisées en créant plusieurs fichiers de configuration XML, plusieurs objets `Configuration` et `SessionFactory` dans votre application.

Le dernier élément de `hibernate.cfg.xml` déclare `Cat.hbm.xml` comme fichier de mapping Hibernate pour la

classe `Cat`. Ce fichier contient les métadonnées du lien entre la classe Java (aussi appelé POJO pour Plain Old Java Object) et une table de la base de données (voire plusieurs tables). Nous reviendrons bientôt sur ce fichier. Commençons par écrire la classe java (ou POJO) et déclarons les métadonnées de mapping pour celle-ci.

1.2. La première classe persistante

Hibernate fonctionne au mieux dans un modèle de programmation consistant à utiliser de Bon Vieux Objets Java (Plain Old Java Objects - POJO) pour les classes persistantes (NdT: on parle de POJO en comparaison d'objets de type EJB ou d'objets nécessitants d'hériter d'une quelconque classe de base). Un POJO est souvent un JavaBean dont les propriétés de la classe sont accessibles via des getters et des setters qui encapsulent la représentation interne dans une interface publique :

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

    private void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Hibernate ne restreint pas l'usage des types de propriétés ; tous les types du JDK et les types primitifs (comme `String`, `char` et `Date`) peuvent être mappés, ceci inclus les classes du framework de collection de Java. Vous pouvez les mapper en tant que valeurs, collections de valeurs ou comme associations avec les autres entités. `id` est une propriété spéciale qui représente l'identifiant dans la base de données pour cette classe (appelé aussi clé primaire). Cet identifiant est chaudement recommandé pour les entités comme `Cat` : Hibernate peut utiliser les

identifiants pour son seul fonctionnement interne (non visible de l'application) mais vous perdriez en flexibilité dans l'architecture de votre application.

Les classes persistantes n'ont besoin d'implémenter aucune interface particulière et n'ont pas besoin d'hériter d'une quelconque classe de base. Hibernate n'utilise également aucun mécanisme de manipulation des classes à la construction, tel que la manipulation du byte-code ; il s'appuie uniquement sur le mécanisme de réflexion de Java et sur l'extension des classes à l'exécution (via CGLIB). On peut donc, sans la moindre dépendance entre les classes POJO et Hibernate, les mapper à une table de la base de données.

1.3. Mapper le Chat

Le fichier de mapping `Cat.hbm.xml` contient les métadonnées requises pour le mapping objet/relationnel. Les métadonnées contiennent la déclaration des classes persistantes et le mapping entre les propriétés (les colonnes, les relations de type clé étrangère vers les autres entités) et les tables de la base de données.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- Une chaîne de 32 caractères hexadécimaux est notre
            clé technique. Elle est générée automatiquement par
            Hibernate en utilisant le pattern UUID. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- Un chat possède un nom mais qui ne doit pas être trop
             long. -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

Toute classe persistante doit avoir un identifiant (en fait, uniquement les classes représentant des entités, pas les valeurs dépendant d'objets, qui sont mappées en tant que composant d'une entité). Cette propriété est utilisée pour distinguer les objets persistants : deux chats sont égaux si l'expression `catA.getId().equals(catB.getId())` est vraie, ce concept est appelé *identité de base de données*. Hibernate fournit en standard un certain nombre de générateurs d'identifiants qui couvrent la plupart des scénarii (notamment les générateurs natifs pour les séquences de base de données, les tables d'identifiants hi/lo, et les identifiants assignés par l'application). Nous utilisons le générateur UUID (recommandé uniquement pour les tests dans la mesure où les clés techniques générées par la base de données doivent être privilégiées). et déclarons que la colonne `CAT_ID` de la table `CAT` contient la valeur de l'identifiant généré par Hibernate (en tant que clé primaire de la table).

Toutes les propriétés de `Cat` sont mappées à la même table. La propriété `name` est mappée utilisant une déclaration explicite de la colonne de base de données. C'est particulièrement utile dans le cas où le schéma de la base de données est généré automatiquement (en tant qu'ordre SQL - DDL) par l'outil d'Hibernate

SchemaExport à partir des déclarations du mapping. Toutes les autres propriétés prennent la valeur par défaut donnée par Hibernate ; ce qui, dans la majorité des cas, est ce que l'on souhaite. La table `CAT` dans la base de données sera :

```
 Colonne |         Type          | Modificateurs
---------+-----------------------+---------------
 cat_id  | character(32)         | not null
 name    | character varying(16) | not null
 sex     | character(1)          |
 weight  | real                  |
Indexes : cat_pkey primary key btree (cat_id)
```

Vous devez maintenant créer manuellement cette table dans votre base de données, plus tard, vous pourrez vous référer à Chapitre 15, *Guide des outils* si vous désirez automatiser cette étape avec l'outil SchemaExport. Cet outil crée un fichier de type DDL SQL qui contient la définition de la table, les contraintes de type des colonnes, les contraintes d'unicité et les index.

1.4. Jouer avec les chats

Nous sommes maintenant prêts à utiliser la `Session` Hibernate. C'est l'interface du *gestionnaire de persistance*, on l'utilise pour sauver et récupérer les `Cats` respectivement dans et à partir de la base de données. Mais d'abord, nous devons récupérer une `Session` (l'unité de travail Hibernate) à partir de la `SessionFactory` :

```
SessionFactory sessionFactory =
            new Configuration().configure().buildSessionFactory();
```

Une `SessionFactory` est responsable d'une base de données et n'accepte qu'un seul fichier de configuration XML (`hibernate.cfg.xml`). Vous pouver positionner les autres propriétés (voire même changer le méta-modèle du mapping) en utilisant `Configuration` *avant* de construire la `SessionFactory` (elle est immuable). Comment créer la `SessionFactory` et comment y accéder dans notre application ?

En général, une `SessionFactory` n'est construite qu'une seule fois, c'est-à-dire au démarrage (avec une servlet de type *load-on-startup*). Cela veut donc dire que l'on ne doit pas la garder dans une variable d'instance des servlets, mais plutôt ailleurs. Il faut un support de type *Singleton* pour pouvoir y accéder facilement. L'approche montrée ci-dessous résout les deux problèmes : celui de configuration et celui de la facilité d'accès à `SessionFactory`.

Nous implémentons `HibernateUtil`, une classe utilitaire

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Crée la SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (HibernateException ex) {
            throw new RuntimeException("Problème de configuration : " + ex.getMessage(), ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Ouvre une nouvelle Session, si ce Thread n'en a aucune
```

```
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```

Non seulement cette classe s'occupe de garder `SessionFactory` dans un de ses attributs statiques, mais en plus elle garde la `Session` du thread courant dans une variable de type `ThreadLocal`. Vous devez bien comprendre le concept Java de variable de type tread-local (locale à un thread) avant d'utiliser cette classe utilitaire.

Une `SessionFactory` est threadsafe : beaucoup de threads peuvent y accéder de manière concurrente et demander une `Session`. Une `Session` est un objet non threadsafe qui représente une unité de travail avec la base de données. Les `Sessions` sont ouvertes par la `SessionFactory` et sont fermées quand le travail est terminé :

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

HibernateUtil.closeSession();
```

Dans une `Session`, chaque opération sur la base de données se fait dans une transaction qui isole les opérations de la base de données (c'est également le cas pour les lectures seules). Nous utilisons l'API `Transaction` pour s'abstraire de la stratégie transactionnelle utilisée (dans notre cas, les transactions JDBC). Cela permet d'avoir un code portable et déployable sans le moindre changement dans un environnement transactionnel géré par le conteneur - CMT - (JTA est utilisé dans ce cas). Il est à noter que l'exemple ci-dessus ne gère pas les exceptions.

Notez également que vous pouvez appeler `HibernateUtil.currentSession();` autant de fois que vous voulez, cette méthode vous ramènera toujours la `Session` courante pour ce thread. Vous devez vous assurer que la `Session` est fermée après la fin de votre unité de travail et avant que la réponse HTTP soit envoyée. Cela peut être par exemple dans le code de votre servlet ou dans un filtre de servlet. L'effet de bord intéressant de la seconde solution est l'initialisation tardive : la `Session` est encore ouverte lorsque la vue est construite. Hibernate peut donc charger, lors de votre navigation dans le graphe, les objets qui n'étaient pas initialisés.

Hibernate possède différentes méthodes de récupération des objets à partir de la base de données. La plus flexible est d'utiliser le langage de requêtage d'Hibernate (HQL comme Hibernate Query Language). Ce langage puissant et facile à comprendre est une extension orientée objet du SQL:

```
Transaction tx = session.beginTransaction();

Query query = session.createQuery("select c from Cat as c where c.sex = :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
```

```
    Cat cat = (Cat) it.next();
    out.println("Chat femelle : " + cat.getName() );
}

tx.commit();
```

Hibernate offre également une API orientée objet de *requêtage par critères* qui peut être utilisée pour formuler des requêtes typées. Bien sûr, Hibernate utilise des `PreparedStatements` et les paramètres associés pour toutes ses communications SQL avec la base de données. Vous pouvez également utiliser la fonctionalité de requêtage SQL natif d'Hibernate ou, dans de rares occasions, récupérer un connexion JDBC à partir de la `Session`.

1.5. Conclusion

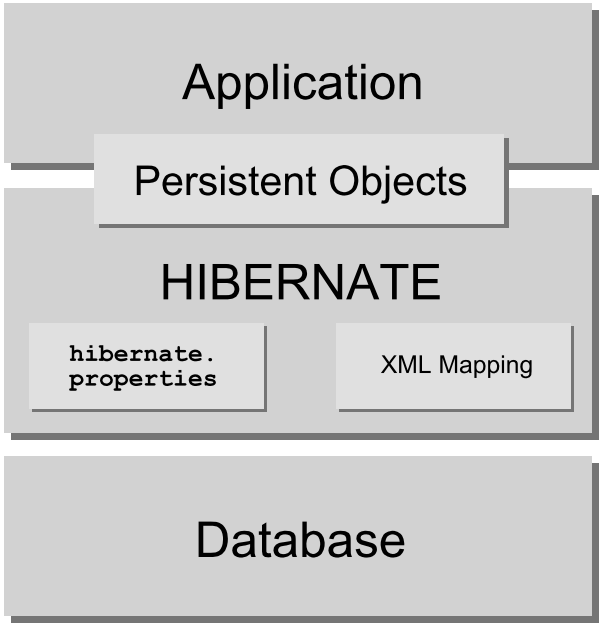
Nous n'avons fait que gratter la surface d'Hibernate dans ce petit tutoriel. Notez que nous n'avons pas inclus de code spécifique aux servlets dans notre exemple. Vous devez créer vous-même une servlet et y insérer le code Hibernate qui convient.

Garder à l'esprit qu'Hibernate, en tant que couche d'accès aux données, est fortement intégré à votre application. En général, toutes les autres couches dépendent du mécanisme de persistance quel qu'il soit. Soyez sûr de comprendre les implications de ce design.

Chapitre 2. Architecture

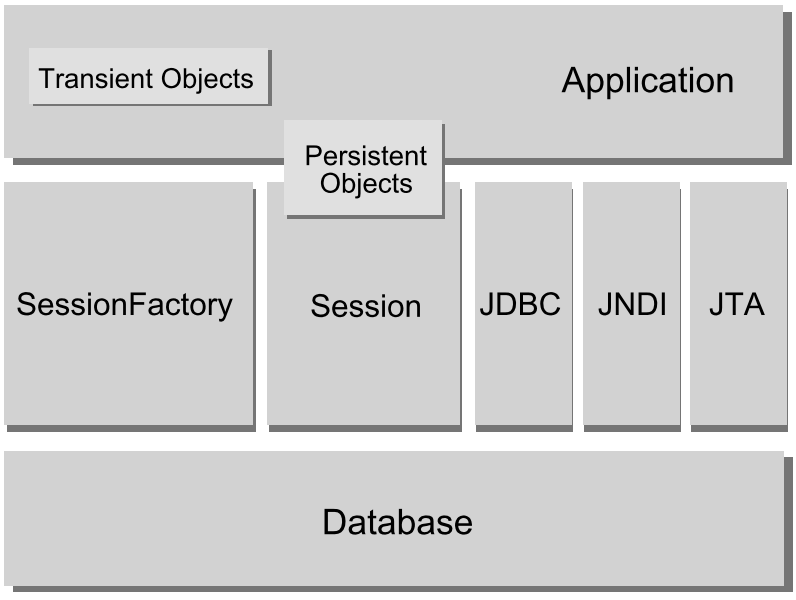
2.1. Généralités

Voici une vue (très) haut niveau de l'architecture d'Hibernate :



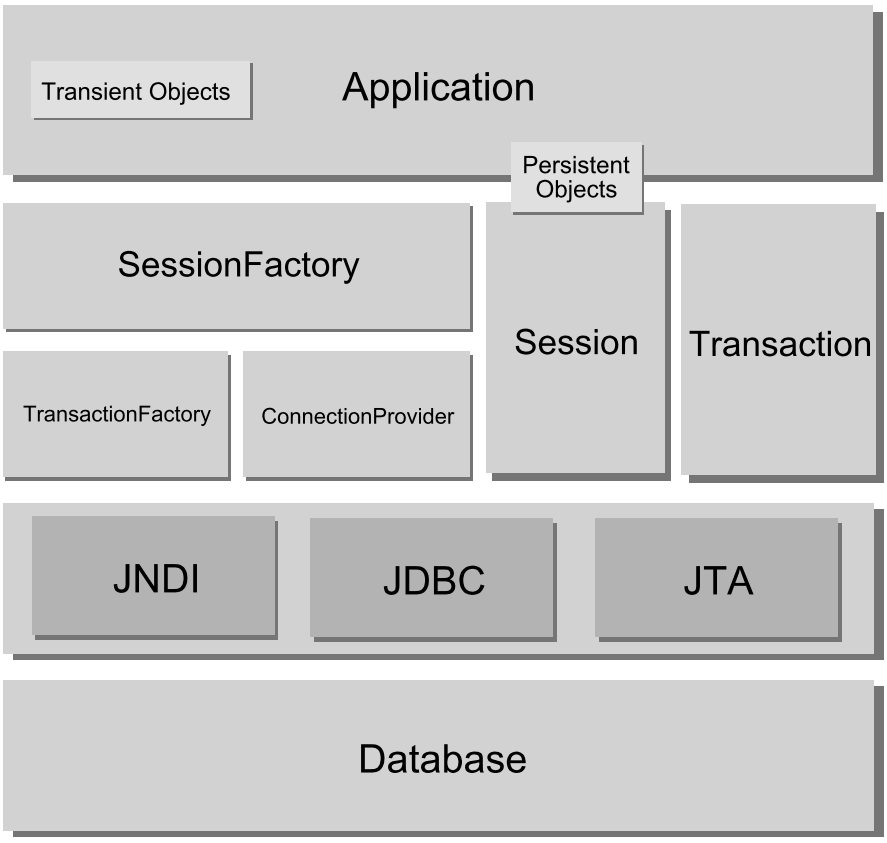
Ce diagramme montre Hibernate utilisant une base de données et des données de configuration pour fournir un service de persistance (et des objets persistants) à l'application.

Nous aimerions décrire une vue plus détaillée de l'architecture. Malheureusement, Hibernate est flexible et supporte différentes approches. Nous allons en montrer les deux extrêmes. L'architecture légère laisse l'application fournir ses propres connexions JDBC et gérer ses propres transactions. Cette approche utilise le minimum des APIs Hibernate.:



L'architecture la plus complète abstrait l'application des APIs JDBC/JTA sous-jacentes et laisse Hibernate

s'occuper des détails.



Voici quelques définitions des objets des diagrammes :

SessionFactory (`net.sf.hibernate.SessionFactory`)
: Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données. Une factory (fabrique) de `Session` et un client de `ConnectionProvider`. Peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au niveau processus ou au niveau cluster.

Session (`net.sf.hibernate.Session`)
: Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC. Factory (fabrique) des objets `Transaction`. Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

Objets et Collections persistants
: Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier. Ceux-ci sont en général les objets de type JavaBean (ou POJOs) ; la seule particularité est qu'ils sont associés avec une (et une seule) `Session`. Dès que la `Session` est fermée, ils seront détachés et libre d'être utilisés par n'importe laquelle des couches de l'application (ie. de et vers la présentation en tant que Data Transfer Objects - DTO : objet de transfert de données).

Objets et collections transients
: Instances de classes persistantes qui ne sont actuellement pas associées à une `Session`. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elle ont pu être instanciées par une `Session` fermée.

Transaction (`net.sf.hibernate.Transaction`)

(Optionnel) Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique. Abstrait l'application des transactions sous-jacentes qu'elles soient JDBC, JTA ou CORBA. Une `Session` peut fournir plusieurs `Transaction`s dans certain cas.

ConnectionProvider (`net.sf.hibernate.connection.ConnectionProvider`)
(Optionnel) Une fabrique de (pool de) connexions JDBC. Abstrait l'application de la `Datasource` ou du `DriverManager` sous-jacent. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

TransactionFactory (`net.sf.hibernate.TransactionFactory`)
(Optionnel) Une fabrique d'instances de `Transaction`. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

Dans une architecture légère, l'application n'utilisera pas les APIs `Transaction`/`TransactionFactory` et/ou n'utilisera pas les APIs `ConnectionProvider` pour utiliser JTA ou JDBC.

2.2. Integration JMX

JMX est le standard J2EE du configuration des composants Java. Hibernate peut être configuré via une MBean standard. Mais dans la mesure où la plupart des serveurs d'application ne supportent pas encore JMX, Hibernate fournit quelques mécanismes de configuration "non-standard".

Merci de vous référer au site web d'Hibernate pour de plus amples détails sur la façon de configurer Hibernate et le faire tourner en tant que composant JMX dans JBoss.

2.3. Support JCA

Hibernate peut aussi être configuré en tant que connecteur JCA. Référez-vous au site web pour de plus amples détails.

Chapitre 3. Configuration de la SessionFactory

Parce qu'Hibernate est conçu pour fonctionner dans différents environnements, il existe beaucoup de paramètres de configuration. Heureusement, la plupart ont des valeurs par défaut appropriées et la distribution d'Hibernate contient un exemple de fichier `hibernate.properties` qui montre les différentes options. Généralement, vous n'avez qu'à placer ce fichier dans votre classpath et à l'adapter.

3.1. Configuration par programmation

Une instance de `net.sf.hibernate.cfg.Configuration` représente un ensemble de mappings des classes Java d'une application vers la base de données SQL. La `Configuration` est utilisée pour construire un objet (immuable) `SessionFactory`. Les mappings sont constitués d'un ensemble de fichiers de mapping XML.

Vous pouvez obtenir une instance de `Configuration` en l'instanciant directement. Voici un exemple de configuration d'une source de données et d'un mapping composé de deux fichiers de configuration XML (qui se trouvent dans le classpath) :

```
Configuration cfg = new Configuration()
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml");
```

Une alternative (parfois meilleure) est de laisser Hibernate charger le fichier de mapping en utilisant `getResourceAsStream()` :

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate va rechercher les fichiers de mappings `/org/hibernate/auction/Item.hbm.xml` et `/org/hibernate/auction/Bid.hbm.xml` dans le classpath. Cette approche élimine les noms de fichiers en dur.

Une `Configuration` permet également plusieurs valeurs optionnelles :

```
Properties props = new Properties();
...
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperties(props);
```

Une `Configuration` est sensée être un objet nécessaire pendant la phase de configuration et être libérée une fois la `SessionFactory` construite.

3.2. Obtenir une SessionFactory

Quand tous les mappings ont été parsés par la `Configuration`, l'application doit obtenir une fabrique d'instances de `Session`. Cette fabrique est supposée être partagée par tous les threads de l'application :

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Cependant, Hibernate permet à votre application d'instancier plus d'une `SessionFactory`. C'est utile si vous utilisez plus d'une base de données.

3.3. Connexion JDBC fournie par l'utilisateur

Une `SessionFactory` peut ouvrir une `Session` en utilisant une connexion JDBC fournie par l'utilisateur. Ce choix de design permet à l'application d'obtenir les connexions JDBC de la façon qu'il lui plait :

```
java.sql.Connection conn = datasource.getConnection();
Session session = sessions.openSession(conn);

// do some data access work
```

L'application doit faire attention à ne pas ouvrir deux `Session`s concurrentes en utilisant la même connexion !

3.4. Connexions JDBC fournie par Hibernate

Alternativement, vous pouvez laisser la `SessionFactory` ouvrir les connexions pour vous. La `SessionFactory` doit recevoir les propriétés de connexions JDBC de l'une des manières suivantes :

1. Passer une instance de `java.util.Properties` à `Configuration.setProperties()`.
2. Placer `hibernate.properties` dans un répertoire racine du classpath
3. Positionner les propriétés `System` en utilisant `java -Dproperty=value`.
4. Inclure des éléments `<property>` dans le fichier `hibernate.cfg.xml` (voir plus loin).

Si vous suivez cette approche, ouvrir une `Session` est aussi simple que :

```
Session session = sessions.openSession(); // ouvre une nouvelle session
// faire quelques accès aux données, une connexion JDBC sera utilisée à la demande
```

Tous les noms et sémantiques des propriétés d'Hibernate sont définies dans la javadoc de la classe `net.sf.hibernate.cfg.Environment`. Nous allons décrire les paramètres les plus importants pour une connexion JDBC.

Hibernate obtiendra des connexions (et les mettra dans un pool) en utilisant `java.sql.DriverManager` si vous positionner les paramètres de la manière suivante :

Tableau 3.1. Propriétés JDBC d'Hibernate

Nom de la propriété	Fonction
`hibernate.connection.driver_class`	*Classe du driver jdbc*
`hibernate.connection.url`	*URL jdbc*
`hibernate.connection.username`	*utilisateur de la base de données*
`hibernate.connection.password`	*mot de passe de la base de données*
`hibernate.connection.pool_size`	*nombre maximum de connexions dans le pool*

L'algorithme natif de pool de connexions d'Hibernate est plutôt rudimentaire. Il a été fait dans le but de vous aider à démarrer et *n'est pas prévu pour un système en production* ou même pour un test de peformance. Utiliser un pool tiers pour de meilleures performances et une meilleure stabilité : remplacer la propriété `hibernate.connection.pool_size` avec les propriétés spécifique au pool de connexions que vous avez choisi.

C3P0 est un pool de connexions JDBC open source distribué avec Hibernate dans le répertoire `lib`. Hibernate utilisera le provider intégré `C3P0ConnectionProvider` pour le pool de connexions si vous positionnez les propriétés `hibernate.c3p0.*`. Il y a également un support intégré pour Apache DBCP et Proxool. Vous devez positionner les propriétés `hibernate.dbcp.*` (propriétés du pool de connexions DBCP) pour activer le `DBCPConnectionProvider`. Le cache des Prepared Statement est activé (fortement recommandé) si `hibernate.dbcp.ps.*` (propriétés du cache de statement de DBCP) sont positionnées. Merci de vous référer à la documentation de apache commons-pool pour l'utilisation et la compréhension de ces propriétés. Vous devez positionner les propriétés `hibernate.proxool.*` si vous voulez utiliser Proxool.

Voici un exemple utilisant C3P0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statement=50
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

Dans le cadre de l'utilisation au sein d'un serveur d'applications, Hibernate peut obtenir les connexions à partir d'une `javax.sql.Datasource` enregistrée dans le JNDI. Positionner les propriétés suivantes :

Tableau 3.2. Propriété d'une Datasource Hibernate

Nom d'une propriété	**fonction**
`hibernate.connection.datasource`	*Nom JNDI de la datasource*
`hibernate.jndi.url`	*URL du fournisseur JNDI* (optionnelle)
`hibernate.jndi.class`	*Classe de l'`InitialContextFactory` du JNDI* (optionnelle)
`hibernate.connection.username`	*utilisateur de la base de données* (optionnelle)
`hibernate.connection.password`	*mot de passe de la base de données* (optionnelle)

voici un exemple utilisant les datasources JNDI fournies par un serveur d'applications :

```
hibernate.connection.datasource = java:/comp/env/jdbc/MyDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = \
    net.sf.hibernate.dialect.PostgreSQLDialect
```

Les connexions JDBC obtenues à partir d'une datasource JNDI participeront automatiquement aux transactions gérées par le conteneur du serveur d'applications.

Des propriétés supplémentaires de connexion peuvent être passées en préfixant le nom de la propriété par "`hibernate.connnection`". Par exemple, vous pouvez spécifier un jeu de caractères en utilisant `hibernate.connnection.charSet`.

Vous pouvez fournir votre propre stratégie d'obtention des connexions JDBC en implémentant l'interface `net.sf.hibernate.connection.ConnectionProvider`. Vous pouvez sélectionner une implémentation

spécifique en positionnant `hibernate.connection.provider_class`.

3.5. Propriétés de configuration optionnelles

Il y a un certain nombre d'autres propriétés qui contrôlent le fonctionnement d'Hibernate à l'exécution. Toutes sont optionnelles et ont comme valeurs par défaut des valeurs "raisonnables" pour un fonctionnement nominal.

Les propriétés de niveau System ne peuvent être positionnées que via la ligne de commande (`java -Dproperty=value`) ou être définies dans `hibernate.properties`. Elle ne peuvent l'être dans une instance de `Properties` passée à la `Configuration`.

Tableau 3.3. Propriétés de configuration d'Hibernate

Nom de la propriété	Fonction
`hibernate.dialect`	Le nom de la classe du `Dialect` Hibernate - active l'utilisation de certaines fonctionalités spécifiques à la plateforme. *ex.* `nom.complet.de.ma.classe.de.Dialect`
`hibernate.default_schema`	Positionne dans le SQL généré un schéma/tablespace par défaut pour les noms de table ne l'ayant pas surchargé. *ex.* `MON_SCHEMA`
`hibernate.session_factory_name`	La `SessionFactory` sera automatiquement liée à ce nom dans le JNDI après sa création. *ex.* `jndi/nom/hierarchique`
`hibernate.use_outer_join`	Active le chargement via les jointures ouvertes. Dépréciée, utiliser `max_fetch_depth`. *ex.* `true` | `false`
`hibernate.max_fetch_depth`	Définit la profondeur maximale d'un arbre de chargement par jointures ouvertes pour les associations à cardinalité unitaire (un-à-un, plusieurs-à-un). Un `0` désactive le chargement par jointure ouverte. *ex.* valeurs recommandées entre `0` et `3`
`hibernate.jdbc.fetch_size`	Une valeur non nulle détermine la taille de chargement des statements JDBC (appelle `Statement.setFetchSize()`).
`hibernate.jdbc.batch_size`	Une valeur non nulle active l'utilisation par Hibernate des mises à jour par batch de JDBC2. *ex.* les valeurs recommandées entre `5` et `30`
`hibernate.jdbc.batch_versioned_data`	Paramétrez cette propriété à `true` si votre pilote JDBC retourne des row counts corrects depuis

Nom de la propriété	Fonction
	`executeBatch()` (il est souvent approprié d'activer cette option). Hibernate utilisera alors le "batched DML" pour versionner automatiquement les données. Par défaut = `false`. *eg.* `true` | `false`
`hibernate.jdbc.use_scrollable_resultset`	Active l'utilisation par Hibernate des resultsets scrollables de JDBC2. Cette propriété est seulement nécessaire lorsque l'on utilise une connexion JDBC fournie par l'utilisateur. Autrement, Hibernate utilise les métadonnées de la connexion. *ex.* `true` | `false`
`hibernate.jdbc.use_streams_for_binary`	Utilise des flux lorsque l'on écrit/lit des types `binary` ou `serializable` vers et à partir de JDBC (propriété de niveau système). *ex.* `true` | `false`
`hibernate.jdbc.use_get_generated_keys`	Active l'utilisation de `PreparedStatement.getGeneratedKeys()` de JDBC3 pour récupérer nativement les clés générées après insertion. Nécessite un pilote JDBC3+, le mettre à false si votre pilote a des problèmes avec les générateurs d'identifiant Hibernate. Par défaut, essaie de déterminer les possibilités du pilote en utilisant les meta données de connexion. *eg.* `true|false`
`hibernate.cglib.use_reflection_optimizer`	Active l'utilisation de CGLIB à la place de la réflexion à l'exécution (Propriété de niveau système, la valeur par défaut étant d'utiliser CGLIB lorsque c'est possible). La réflexion est parfois utile en cas de problème. *ex.* `true` | `false`
`hibernate.jndi.<propertyName>`	Passe la propriété `propertyName` au JNDI `InitialContextFactory`.
`hibernate.connection.isolation`	Positionne le niveau de transaction JDBC. Merci de vous référer à `java.sql.Connection` pour le détail des valeurs mais sachez que toutes les bases de données ne supportent pas tous les niveaux d'isolation. *ex.* `1, 2, 4, 8`
`hibernate.connection.<propertyName>`	Passe la propriété JDBC `propertyName` au `DriverManager.getConnection()`.
`hibernate.connection.provider_class`	Le nom de classe d'un `ConnectionProvider` spécifique.

Nom de la propriété	Fonction
	ex. `nom.de.classe.du.ConnectionProvider`
`hibernate.cache.provider_class`	Le nom de classe d'un `CacheProvider` spécifique. *ex.* `nom.de.classe.du.CacheProvider`
`hibernate.cache.use_minimal_puts`	Optimise le cache de second niveau en minimisant les écritures, au prix de plus de lectures (utile pour les caches en cluster). *ex.* `true|false`
`hibernate.cache.use_query_cache`	Activer le cache de requête, les requêtes individuelles doivent tout de même être déclarées comme mettable en cache. *ex.* `true|false`
`hibernate.cache.query_cache_factory`	Le nom de classe d'une interface `QueryCache` , par défaut = built-in `StandardQueryCache`. *eg.* `nom.de.la.classe.de.QueryCache`
`hibernate.cache.region_prefix`	Un préfixe à utiliser pour le nom des régions du cache de second niveau. *ex.* `prefix`
`hibernate.transaction.factory_class`	Le nom de classe d'une `TransactionFactory` qui sera utilisée par l'API `Transaction` d'Hibernate (la valeur par défaut est `JDBCTransactionFactory`). *ex.* `nom.de.classe.d.une.TransactionFactory`
`jta.UserTransaction`	Le nom JNDI utilisé par la `JTATransactionFactory` pour obtenir la `UserTransaction` JTA du serveur d'applications. *eg.* `jndi/nom/compose`
`hibernate.transaction.manager_lookup_class`	Le nom de la classe du `TransactionManagerLookup` - requis lorsque le cache de niveau JVM est activé dans un environnement JTA. *ex.* `nom.de.classe.du.TransactionManagerLookup`
`hibernate.query.substitutions`	Lien entre les tokens de requêtes Hibernate et les tokens SQL (les tokens peuvent être des fonctions ou des noms littéraux par exemple). *ex.* `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`hibernate.show_sql`	Ecrit les ordres SQL dans la console. *ex.* `true` | `false`

Nom de la propriété	Fonction
`hibernate.hbm2ddl.auto`	Exporte le schéma DDL vers la base de données automatiquement lorsque la `SessionFactory` est créée. La valeur `create-drop` permet de supprimer le schéma de base de données lorsque la `SessionFactory` est fermée explicitement. *ex.* `update` | `create` | `create-drop`

3.5.1. Dialectes SQL

Vous devriez toujours positionner la propriété `hibernate.dialect` à la sous-classe appropriée à votre base de données. Ce n'est pas strictement obligatoire à moins de vouloir utiliser la génération de clé primaire `native` ou par `sequence` ou de vouloir utiliser le mécanisme de lock pessimiste (ex. via `Session.lock()` ou `Query.setLockMode()`). Cependant, si vous spécifiez un dialecte, Hibernate utilisera des valeurs adaptées pour certaines autres propriétés listées ci-dessus, vous évitant l'effort de le faire à la main.

Tableau 3.4. Dialectes SQL d'Hibernate (`hibernate.dialect`)

SGBD	Dialecte
DB2	`net.sf.hibernate.dialect.DB2Dialect`
DB2 AS/400	`net.sf.hibernate.dialect.DB2400Dialect`
DB2 OS390	`net.sf.hibernate.dialect.DB2390Dialect`
PostgreSQL	`net.sf.hibernate.dialect.PostgreSQLDialect`
MySQL	`net.sf.hibernate.dialect.MySQLDialect`
SAP DB	`net.sf.hibernate.dialect.SAPDBDialect`
Oracle (toutes versions)	`net.sf.hibernate.dialect.OracleDialect`
Oracle 9/10g	`net.sf.hibernate.dialect.Oracle9Dialect`
Sybase	`net.sf.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`net.sf.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`net.sf.hibernate.dialect.SQLServerDialect`
SAP DB	`net.sf.hibernate.dialect.SAPDBDialect`
Informix	`net.sf.hibernate.dialect.InformixDialect`
HypersonicSQL	`net.sf.hibernate.dialect.HSQLDialect`
Ingres	`net.sf.hibernate.dialect.IngresDialect`
Progress	`net.sf.hibernate.dialect.ProgressDialect`
Mckoi SQL	`net.sf.hibernate.dialect.MckoiDialect`
Interbase	`net.sf.hibernate.dialect.InterbaseDialect`
Pointbase	`net.sf.hibernate.dialect.PointbaseDialect`

SGBD	Dialecte
FrontBase	`net.sf.hibernate.dialect.FrontbaseDialect`
Firebird	`net.sf.hibernate.dialect.FirebirdDialect`

3.5.2. Chargement par Jointure Ouverte

Si votre base de données supporte les outer joins de type ANSI ou Oracle, *le chargement par jointure ouverte* devrait améliorer les performances en limitant le nombre d'aller-retour avec la base de données (la base de données effectuant donc potentiellement plus de travail). Le chargement par jointure ouverte permet à un graphe connecté d'objets par une relation plusieurs-à-un, un-à-plusieurs ou un-à-un d'être chargé en un seul `SELECT` SQL.

Par défaut, le graphe chargé lorsqu'un objet est demandé, finit aux objets feuilles, aux collections, aux objets avec proxy ou lorsqu'une circularité apparaît.

Le chargement peut être activé ou désactivé (valeur par défaut) pour une *association particulière*, en positionant l'attribut `outer-join` dans le mapping XML.

Le chargement par jointure ouverte peut être désactivé *de manière globale* en positionant la propriété `hibernate.max_fetch_depth` à `0`. Une valeur de `1` ou plus permet les jointures ouvertes pour toutes les associations un-à-un et plusieurs-à-un qui sont, par défaut, positionnées à la valeur de jointure outerte `auto`. Cependant, les associations un-à-plusieurs et les collections ne sont jamais chargées en utilisant une jonture ouverte, à moins de le déclarer de façon explicite pour chaque association. Cette fonctionalité peut être surchargée à l'exécution dans les requêtes Hibernate.

3.5.3. Flux binaires

Oracle limite la taille d'un tableau de `byte` qui peuvent être passées à et vers son pilote JDBC. Si vous souhaitez utiliser des instances larges de type `binary` ou `serializable`, vous devez activer la propriété `hibernate.jdbc.use_streams_for_binary`. *C'est une fonctionalité de niveau JVM uniquement.*

3.5.4. `CacheProvider` spécifique

Vous pouvez intégrer un cache de second niveau de type JVM (ou cluster) en implémentant l'interface `net.sf.hibernate.cache.CacheProvider`. Vous pouvez sélectionner l'implémentation spécifique en positionnant `hibernate.cache.provider_class`.

3.5.5. Configuration de la stratégie transactionnelle

Si vous souhaitez utiliser l'API d'Hibernate `Transaction`, vous devez spécifier une classe factory d'instances de `Transaction` en positionnant la propriété `hibernate.transaction.factory_class`. L'API `Transaction` masque le mécanisme de transaction sous-jacent et permet au code utilisant Hibernate de tourner dans des environnements managés et non-managés sans le moindre changement.

Il existe deux choix standards (fournis) :

`net.sf.hibernate.transaction.JDBCTransactionFactory`
: délègue aux transactions de la base de données (JDBC). Valeur par défaut.

`net.sf.hibernate.transaction.JTATransactionFactory`
: délègue à JTA (si une transaction existant est en cours, la `Session` exécute son travail dans ce contexte ; sinon, une nouvelle transaction est démarrée).

Vous pouvez également définir votre propre stratégie transactionnelle (pour un service de transaction CORBA par exemple).

Si vous voulez utiliser un cache de niveau JVM pour des données muables dans un environnement JTA, vous devez spécifier une stratégie d'obtention du `TransactionManager` JTA. En effet, cet accès n'est pas standardisé par la norme J2EE :

Tableau 3.5. TransactionManagers JTA

Factory de Transaction	Serveur d'application
`net.sf.hibernate.transaction.JBossTransactionManagerLookup`	JBoss
`net.sf.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`net.sf.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere
`net.sf.hibernate.transaction.OrionTransactionManagerLookup`	Orion
`net.sf.hibernate.transaction.ResinTransactionManagerLookup`	Resin
`net.sf.hibernate.transaction.JOTMTransactionManagerLookup`	JOTM
`net.sf.hibernate.transaction.JOnASTransactionManagerLookup`	JOnAS
`net.sf.hibernate.transaction.JRun4TransactionManagerLookup`	JRun4
`net.sf.hibernate.transaction.BESTransactionManagerLookup`	Borland ES

3.5.6. `SessionFactory` associée au JNDI

Une `SessionFactory` Hibernate associée au JNDI peut simplifier l'accès à la fabrique et donc la création de nouvelles `Session`s.

Si vous désirez associer la `SessionFactory` à un nom JNDI, spécifiez un nom (ex. `java:comp/env/hibernate/SessionFactory`) en utilisant la propriété `hibernate.session_factory_name`. Si cette propriété est omise, la `SessionFactory` ne sera pas associée au JNDI (c'est particulièrement pratique dans les environnements ayant une implémentation de JNDI en lecture seule, comme c'est le cas pour Tomcat).

Lorsqu'il associe la `SessionFactory` au JNDI, Hibernate utilisera les valeurs de `hibernate.jndi.url`, `hibernate.jndi.class` pour instancier un contexte d'initialisation. S'ils ne sont pas spécifiés, l'`InitialContext` par défaut sera utilisé.

Si vous décidez d'utiliser JNDI, un EJB ou toute autre classe utilitaire pourra obtenir la `SessionFactory` en faisant un accès au JNDI.

3.5.7. Substitution dans le langage de requêtage

Vous pouvez définir de nouveaux tokens dans les requêtes Hibernate en utilisant la propriété `hibernate.query.substitutions`. Par exemple :

```
hibernate.query.substitutions vrai=1, faux=0
```

remplacerait les tokens `vrai` et `faux` par des entiers dans le SQL généré.

```
hibernate.query.substitutions toLowercase=LOWER
```

permettrait de renommer la fonction SQL `LOWER` en `toLowercase`

3.6. Logguer

Hibernate loggue divers évènements en utilisant Apache commons-logging.

Le service commons-logging délèguera directement à Apache Log4j (si vous incluez `log4j.jar` dans votre classpath) ou le système de log du JDK 1.4 (si vous tournez sous le JDK 1.4 et supérieur). Vous pouvez télécharger Log4j à partir de `http://jakarta.apache.org`. Pour utiliser Log4j, vous devrez placer dans votre classpath un fichier `log4j.properties`. Un exemple de fichier est distribué avec Hibernate dans le répertoire `src/`.

Nous vous recommandons fortement de vous familiariser avec les messages de logs d'Hibernate. Beaucoup de soins a été apporté pour donner le plus de détails possibles sans les rendre illisibles. C'est un outil essentiel en cas de soucis. De même, n'oubliez pas d'activer les logs SQL comme décrit précédemment `hibernate.show_sql`, c'est la première étape pour regarder les problèmes de performance.

3.7. Implémenter une `NamingStrategy`

L'interface `net.sf.hibernate.cfg.NamingStrategy` vous permet de spécifier une "stratégie de nommage" des objets et éléments de la base de données.

Vous pouvez fournir des règles pour automatiquement générer les identifiants de base de données à partir des identifiants Java, ou transformer une colonne ou table "logique" donnée dans le fichier de mapping en une colonne ou table "physique". Cette fonctionnalité aide à réduire la verbosité de documents de mapping, en éliminant le bruit répétitif (les préfixes `TBL_` par exemple). La stratégie par défaut utilisée par Hibernate est minimale.

Vous pouvez définir une stratégie différente en appelant `Configuration.setNamingStrategy()` avant d'ajouter des mappings :

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

`net.sf.hibernate.cfg.ImprovedNamingStrategy` est une stratégie fournie qui peut être utile comme point de départ de quelques applications.

3.8. Fichier de configuration XML

Une approche alternative est de spécifier toute la configuration dans un fichier nommé `hibernate.cfg.xml`. Ce fichier peut être utilisé à la place du fichier `hibernate.properties`, voire même peut servir à surcharger les propriétés si les deux fichiers sont présents.

Le fichier de configuration XML doit par défaut se placer à la racine du CLASSPATH. En voici un exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 2.0//EN"

 "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <!-- une instance de SessionFactory accessible par son nom jndi -->
    <session-factory
        name="java:comp/env/hibernate/SessionFactory">

        <!-- propriétés -->
        <property name="connection.datasource">ma/premiere/datasource</property>
        <property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="use_outer_join">true</property>
        <property name="transaction.factory_class">
            net.sf.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Configurer Hibernate devient donc aussi simple que ceci :

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

Vous pouvez utiliser une fichier de configuration XML de nom différent en utilisant

```
SessionFactory sf = new Configuration()
    .configure("/my/package/catdb.cfg.xml")
    .buildSessionFactory();
```

Chapitre 4. Classes persistantes

Les classes persistantes sont les classes d'une application qui implémentent les entités d'un problème métier (ex. Client et Commande dans une application de commerce électronique). Les classes persistantes ont, comme leur nom l'indique, des instances transiantes mais aussi persistantes c'est-à-dire stockées en base de données.

Hibernate fonctionne de manière optimale lorsque ces classes suivent quelques règles simples, aussi connues comme le modèle de programmation Plain Old Java Object (POJO).

4.1. Un exemple simple de POJO

Toute bonne application Java nécessite une classe persistante représentant les félins.

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifiant
    private String name;
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    void setMate(Cat mate) {
        this.mate = mate;
    }
    public Cat getMate() {
        return mate;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
```

```
    void setColor(Color color) {
        this.color = color;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }
    // addKitten n'est pas nécessaire pour Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }
    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }
}
```

Il y a quatre règles à suivre ici :

4.1.1. Déclarer les accesseurs et modifieurs des attributs persistants

`Cat` déclare des méthodes d'accès pour tous ses attributs persistants. Beaucoup d'autres solutions de mapping Objet/relationnel persistent directement les instances des attributs. Nous pensons qu'il est bien mieux de découpler ce détail d'implémentation du mécanisme de persistence. Hibernate persiste les propriétés suivant le style JavaBeans et reconnait les noms de méthodes de la forme `getFoo`, `isFoo` et `setFoo`.

Les propriétés *n'ont pas* à être déclarées publiques - Hibernate peut persister une propriété avec un paire de getter/setter de visibilité par défault, `protected` ou `private`

4.1.2. Implémenter un constructeur par défaut

`Cat` possède un constructeur par défaut (sans argument) implicite. Toute classe persistante doit avoir un constructeur par défaut (qui peut être non-publique) pour permettre à Hibernate de l'instancier en utilisant `Constructor.newInstance()`.

4.1.3. Fournir une propriété d'indentifiant (optionnel)

`Cat` possède une propriété appelée `id`. Cette propriété conserve la valeur de la colonne de clé primaire de la table d'une base de données. La propriété aurait pu s'appeler complètement autrement, et son type aurait pu être n'importe quel type primitif, n'importe quel "encapsuleur" de type primitif, `java.lang.String` ou `java.util.Date`. (Si votre base de données héritée possède des clés composites, elles peuvent être mappées en utilisant une classe définie par l'utilisateur et possédant les propriétés associées aux types de la clé composite - voir la section concernant les identifiants composites plus bas).

La propriété d'identifiant est optionnelle. Vous pouver l'oublier et laisser Hibernate s'occuper des identifiants de l'objet en interne. Cependant, pour beaucoup d'applications, avoir un identifiant reste un design bon (et très populaire).

De plus, quelques fonctionnalités ne sont disponibles que pour les classes déclarant un identifiant de propriété :

- Mises à jour en cascade (Voir "Cycle de vie des objets")
- `Session.saveOrUpdate()`

Nous recommandons que vous déclariez les propriétés d'identifiant de manière uniforme. Nous recommandons également que vous utilisiez un type nullable (ie. non primitif).

4.1.4. Favoriser les classes non finales (optionnel)

Une fonctionalité clée d'Hibernate, les *proxies*, nécessitent que la classe persistente soit non finale ou qu'elle soit l'implémentation d'une interface qui déclare toutes les méthodes publiques.

Vous pouvez persister, grâce à Hibernate, les classes `final` qui n'implémentent pas d'interface, mais vous ne pourrez pas utiliser les proxies - ce qui limitera vos possibilités d'ajustement des performances.

4.2. Implémenter l'héritage

Une sous-classe doit également suivre la première et la seconde règle. Elle hérite sa propriété d'identifiant de `Cat`.

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. Implémenter `equals()` et `hashCode()`

Vous devez surcharger les méthodes `equals()` et `hashCode()` si vous avez l'intention de "mélanger" des objets de classes persistantes (ex dans un `Set`).

Cette règle ne s'applique que si ces objets sont chargés à partir de deux `Sessions` différentes, dans la mesure où Hibernate ne garantit l'identité de niveau JVM (`a == b` , l'implémentation par défaut d'`equals()` en Java) qu'au sein d'une seule `Session` !

Même si deux objets `a` et `b` représentent la même ligne dans la base de données (ils ont la même valeur de clé primaire comme identifiant), nous ne pouvons garantir qu'ils seront la même instance Java hors du contexte d'une `Session` donnée.

La manière la plus évidente est d'implémenter `equals()`/`hashCode()` en comparant la valeur de l'identifiant des deux objets. Si cette valeur est identique, les deux doivent représenter la même ligne de base de données, ils sont donc égaux (si les deux sont ajoutés à un `Set`, nous n'auront qu'un seul élément dans le `Set`). Malheureusement, nous ne pouvons pas utiliser cette approche. Hibernate n'assignera de valeur d'identifiant qu'aux objets qui sont persistant, une instance nouvellement créée n'aura donc pas de valeur d'identifiant ! Nous recommandons donc d'implémenter `equals()` et `hashCode()` en utilisant *l'égalité par clé métier*.

L'égalité par clé métier signifie que la méthode `equals()` compare uniquement les propriétés qui forment une clé métier, une clé qui identifierait notre instance dans le monde réel (une clé candidate *naturelle*) :

```
public class Cat {
```

```
    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof Cat)) return false;

        final Cat cat = (Cat) other;

        if (!getName().equals(cat.getName())) return false;
        if (!getBirthday().equals(cat.getBirthday())) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getName().hashCode();
        result = 29 * result + getBirthday().hashCode();
        return result;
    }

}
```

Garder à l'esprit que notre clé candidate (dans ce cas, une clé composée du nom et de la date de naissance) n'a à être valide et pertinente que pour une opération de comparaison particulière (peut-être même pour un seul cas d'utilisation). Nous n'avons pas besoin du même critère de stabilité que celui nécessaire à la clé primaire réelle !

4.4. Callbacks de cycle de vie

Une classe persistence peut, de manière facultative, implémenter l'interface `Lifecycle` qui fournit des callbacks permettant aux objets persistants d'effectuer des opérations d'initialisation ou de nettoyage après une sauvegarde ou un chargement et avant une suppression ou une mise à jour.

L'`Interceptor` d'Hibernate offre cependant une alternative moins intrusive.

```
public interface Lifecycle {
        public boolean onSave(Session s) throws CallbackException;   (1)
        public boolean onUpdate(Session s) throws CallbackException; (2)
        public boolean onDelete(Session s) throws CallbackException; (3)
        public void onLoad(Session s, Serializable id);              (4)
}
```

(1) `onSave` - appelé juste avant que l'objet soit sauvé ou inséré
(2) `onUpdate` - appelé juste avant qu'un objet soit mis à jour (quand l'objet est passé à `Session.update()`)
(3) `onDelete` - appelé juste avant que l'objet soit supprimé
(4) `onLoad` - appelé juste après que l'objet soit chargé

`onSave()`, `onDelete()` et `onUpdate()` peuvent être utilisés pour sauver ou supprimer en cascade de objets dépendants. `onLoad()` peut être utilisé pour initialiser des propriétés transiantes de l'objet à partir de son état persistant. Il ne doit pas être utilisé pour charger des objets dépendants parce que l'interface `Session` ne doit pas être appelée au sein de cette méthode. Un autre usage possible de `onLoad()`, `onSave()` et `onUpdate()` est de garder une référence à la `Session` courante pour un usage ultérieur.

Notez que `onUpdate()` n'est pas appelé à chaque fois que l'état persistant d'un objet est mis à jour. Elle n'est appelée que lorsqu'un objet transiant est passé à `Session.update()`.

Si `onSave()`, `onUpdate()` ou `onDelete()` retourne `true`, l'opération n'est pas effectuée et ceci de manière silencieuse. Si une `CallbackException` est levée, l'opération n'est pas effectuée et l'exception est retournée à l'application.

Notez que `onSave()` est appelé après que l'identifiant ait été assigné à l'objet sauf si la génération native de clés est utilisée.

4.5. Callback de validation

Si la classe persistante a besoin de vérifier des invariants avant que son état soit persisté, elle peut implémenter l'interface suivante :

```
public interface Validatable {
        public void validate() throws ValidationFailure;
}
```

L'objet doit lever une `ValidationFailure` si un invariant a été violé. Une instance de `Validatable` ne doit pas changer son état au sein de la méthode `validate()`.

Contrairement aux méthodes de callback de l'interface `Lifecycle`, `validate()` peut être appelé à n'importe quel moment. L'application ne doit pas s'appuyer sur les appels à `validate()` pour des fonctionalités métier.

4.6. Utiliser le marquage XDoclet

Dans le chapitre suivant, nous allons voir comment les mappings Hibernate sont exprimés dans un format XML simple et lisible. Beaucoup d'utilisateurs d'Hibernate préfèrent embarquer les informations de mapping directement dans le code source en utilisant les tags XDoclet `@hibernate.tags`. Nous ne couvrirons pas cette approche dans ce document parce que considérée comme une part de XDoclet. Cepdendant, nous avons inclus l'exemple suivant utilisant la classe `Cat` et le mapping XDoclet.

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifiant
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /**
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="MATE_ID"
     */
    public Cat getMate() {
```

```
        return mate;
    }
    void setMate(Cat mate) {
        this.mate = mate;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  lazy="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten n'est pas nécesaire à Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

Chapitre 5. Mapping O/R basique

5.1. Déclaration de Mapping

Les mappings objet/relationnel sont définis dans un document XML. Le document de mapping est conçu pour être lisible et éditable à la main. Le vocabulaire de mapping est orienté Java, ce qui signifie que les mappings sont construits autour des classes java et non autour des déclarations de tables.

Même si beaucoup d'utilisateurs d'Hibernate choisissent d'écrire les fichiers de mapping à la main, il existe des outils pour les générer, comme XDoclet, Middlegen et AndroMDA.

Enchaînons sur un exemple de mapping:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat" discriminator-value="D">
                        <property name="name" type="string"/>
                </subclass>
        </class>

        <class name="Dog">
                <!-- Le mapping de dog peut être placé ici -->
        </class>

</hibernate-mapping>
```

Nous allons parler du document de mapping. Nous aborderons uniquement les éléments du document utilisés à l'exécution par Hibernate. Ce document contient d'autres attributs et éléments optionnels qui agissent sur le schéma de base de données exporté par l'outil d'export de schéma.(par exemple l'attribut `not-null`.)

5.1.1. Doctype

Tous les mappings XML doivent déclarer le doctype de l'exemple précédent. L'actuelle DTD peut être trouvée à l'URL du dessus, dans le répertoire `hibernate-x.x.x/src/net/sf/hibernate` ou dans `hibernate2.jar`. Hibernate cherchera toujours en priorité la DTD dans le classpath.

5.1.2. hibernate-mapping

Cet élément possède trois attributs optionnels. L'attribut `schema` spécifie à quel schéma appartiennent les tables déclarées par ce mapping. S'il est spécifié, les noms des tables seront qualifiés par le nom de schéma donné. S'il est absent, les noms des tables ne seront pas qualifiées. L'attribut `default-cascade` spécifie quel style de cascade doit être adopté pour les propriétés et collections qui ne spécifient par leur propre attribut `cascade`. L'attribut `auto-import` nous permet d'utiliser, par défaut, des noms de classe non qualifiés dans le langage de requête.

```
<hibernate-mapping
         schema="nomDeSchema"                          (1)
         default-cascade="none|save-update"            (2)
         auto-import="true|false"                      (3)
         package="nom.de.package"                      (4)
 />
```

(1) `schema` (optionnel): Le nom du schéma de base de données.

(2) `default-cascade` (optionnel - par défaut = `none`): Un style de cascade par défaut.

(3) `auto-import` (optionnel - par défaut = `true`): Spécifie si l'on peut utiliser des noms de classes non qualifiés (pour les classes de ce mapping) dans le langage de requête.

(4) `package` (optionnel): Spécifie un préfixe de package à prendre en compte pour les noms de classes non qualifiées dans le mapping courant.

Si vous avez deux classes persistantes avec le même nom (non qualifié), vous devriez utiliser `auto-import="false"`. Hibernate lancera une exception si vous essayez d'assigner deux classes au même nom "importé".

5.1.3. class

Vous pouvez déclarer une classe persistante en utilisant l'élément `class`:

```
<class
        name="NomDeClasse"                             (1)
        table="NomDeTable"                             (2)
        discriminator-value="valeur_de_discriminant"   (3)
        mutable="true|false"                           (4)
        schema="proprietaire"                          (5)
        proxy="InterfaceDeProxy"                       (6)
        dynamic-update="true|false"                    (7)
        dynamic-insert="true|false"                    (8)
        select-before-update="true|false"              (9)
        polymorphism="implicit|explicit"               (10)
        where="condition SQL where quelconque"         (11)
        persister="ClasseDePersistance"                (12)
        batch-size="N"                                 (13)
        optimistic-lock="none|version|dirty|all"       (14)
        lazy="true|false"                              (15)
/>
```

(1) `name` : Le nom de classe entièrement qualifié pour la classe (ou l'interface) persistante.

(2) `table` : Le nom de sa table en base de données.

(3) `discriminator-value` (optionnel - valeur par défaut = nom de la classe) : Une valeur qui distingue les classes filles, utilisé pour le comportement polymorphique. Sont aussi autorisées les valeurs `null` et `not null`.

(4) `mutable` (optionnel, valeur par défaut = `true`) : Spécifie qu'une instance de classe est (ou n'est pas) mutable.

(5) `schema` (optionnel) : Surcharge le nom de schéma défini par l'élément racine `<hibernate-mapping>`.

(6) `proxy` (optionnel) : Spécifie une interface à utiliser pour initialiser tardivement (lazy) les proxies. Vous pouvez spécifier le nom de la classe elle-même.

(7) `dynamic-update` (optionnel, valeur par défaut = `false`): Spécifie si l'ordre SQL `UPDATE` doit être généré à l'exécution et ne contenir que les colonnes dont les valeurs ont changé.

(8) `dynamic-insert` (optionnel, valeur par défaut = `false`): Spécifie si l'ordre SQL `INSERT` doit être généré à l'exécution et ne contenir que les colonnes dont les valeurs ne sont pas null.

(9) `select-before-update` (optionnel, valeur par défaut = `false`): Spécifie qu'Hibernate ne doit *jamais* effectuer un `UPDATE` SQL à moins d'être certain qu'un objet ait réellement été modifié. Dans certains cas (en fait, lorsqu'un objet transiant a été associé à une nouvelle session en utilisant `update()`), cela signifie qu'Hibernate effectuera un `SELECT` SQL supplémentaire pour déterminer si un `UPDATE` est réellement requis.

(10) `polymorphism` (optionnel, par défaut = `implicit`): Détermine si, pour cette classe, une requête polymorphique implicite ou explicite est utilisée.

(11) `where` (optionnel) spécifie une clause SQL `WHERE` à utiliser lorsque l'on récupère des objets de cette classe.

(12) `persister` (optionnel): Spécifie un `ClassPersister` particulier.

(13) `batch-size` (optionnel, par défaut = `1`) spécifie une taille de batch pour remplir les instances de cette classe par identifiant en une seule requête.

(14) `optimistic-lock` (optionnel, par défaut = `version`): Détermine la stratégie de verrou optimiste.

(15) `lazy` (optionnel): Déclarer `lazy="true"` est un raccourci pour spécifier le nom de la classe comme étant l'interface `proxy`.

Il est parfaitement acceptable pour une classe persistante nommée, d'être une interface. Vous devriez alors déclarer les classes implémentant cette interface via l'élément `<subclass>`. Vous pouvez persister n'importe quelle classe interne *statique*. Vous devriez spécifier le nom de classe en utilisant la forme standard : `eg.Foo$Bar`.

Les classes non mutables, `mutable="false"`, ne peuvent être modifiées ou effacées par l'application. Cela permet à Hibernate d'effectuer quelques optimisations de performance mineures.

L'attribut optionnel `proxy` active l'initialisation tardive des instances persistantes de la classe. Hibernate retournera d'abord des proxies CGLIB qui implémentent l'interface définie. Les objets persistants réels seront chargés lorsqu'une méthode du proxy est invoquée. Voir "Proxies pour initialisation tardive" ci dessous.

Le polymorphisme *implicite* signifie que les instances de la classe seront retournées par une requête qui utilise les noms de la classe ou de chacunes de ses superclasses ou encore des interfaces implémentées par cette classe ou ses superclasses. Les instances des classes filles seront retournées par une requête qui utilise le nom de la classe elle même. Le polymorphisme *explicite* signifie que les instances de la classe ne seront retournées que par une requête qui utilise explicitement son nom et que seules les instances des classes filles déclarées dans les éléments `<subclass>` ou `<joined-subclass>` seront retournées. Dans la majorités des cas la valeur par défaut, `polymorphism="implicit"`, est appropriée. Le polymorphisme explicite est utile lorsque deux classes différentes sont mappées à la même table (ceci permet d'écrire une classe "légère" qui ne contient qu'une partie des colonnes de la table - voir la partie design pattern du site communautaire).

L'attribut `persister` vous permet de customiser la stratégie de persistance utilisée pour la classe. Vous pouvez, par exemple, spécifier votre propre classe fille de `net.sf.hibernate.persister.EntityPersister` ou vous pouvez même fournir une nouvelle implémentation de l'interface `net.sf.hibernate.persister.ClassPersister` qui implémente la persistance via, par exemple, des appels à une procédure stockée, la sérialisation dans des fichiers plats ou dans un LDAP. Voir `net.sf.hibernate.test.CustomPersister` pour un exemple simple (de "persistance" dans une `Hashtable`).

Notez que les paramètres `dynamic-update` et `dynamic-insert` ne sont pas hérités par les classes filles et peuvent donc être spécifiés dans les éléments `<subclass>` ou `<joined-subclass>`. Ces paramètres peuvent accroître les performances dans certains cas, mais peuvent aussi être plus lourds dans d'autres cas. A utiliser de manière judicieuse.

L'utilisation de `select-before-update` fera généralement baisser les performances. Il est cependant très

pratique lorsque l'on veut empêcher un trigger de base de données qui se déclenche sur un update d'être appelé inutilement.

Si vous activez `dynamic-update`, vous aurez le choix entre les stratégies de verrou optimiste suivantes:

- `version` vérifie les colonnes version/timestamp
- `all` vérifie toutes les colonnes
- `dirty` vérifie les colonnes modifiées
- `none` n'utilise pas le verrou optimiste

Nous vous recommandons vivement d'utiliser les colonnes version/timestamp pour le verrou optimiste avec Hibernate. C'est la stratégie optimale qui respecte les performances et c'est la seule capable de gérer correctement les modifications faites en dehors de la session (c'est-à-dire : lorsque `Session.update()` est utilisée). Gardez à l'esprit qu'une propriété version ou timestamp ne devrait jamais être nulle, quelle que soit la stratégie d'`unsaved-value`, ou alors une instance sera détectée comme transiante.

5.1.4. id

Les classes mappées *doivent* déclarer la colonne clé primaire de la table. La plupart des classes auront aussi une propriété, respectant la convention JavaBean, contenant l'identifiant unique d'une instance. L'élément `<id>` définit le mapping entre cette propriété et cette colonne clé primaire.

```
<id
        name="nomDePropriete"                   (1)
        type="nomdetype"                        (2)
        column="nom_de_colonne"                 (3)
        unsaved-value="any|none|null|id_value"  (4)
        access="field|property|NomDeClasse">    (5)

        <generator class="generatorClass"/>
</id>
```

(1) `name` (optionnel) : Le nom de la propriété d'identifiant.
(2) `type` (optionnel) : Le nom qui indique le type Hibernate.
(3) `column` (optionnel - par défaut le nom de la propriété) : Le nom de la colonne de la clé primaire.
(4) `unsaved-value` (optionnel - par défaut = `null`) : Une valeur de la propriété d'identifiant qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances transiantes qui ont été sauvegardées ou chargées dans une session précédente.
(5) `access` (optionnel - par défaut = `property`): La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.

Si l'attribut `name` est manquant, on suppose que la classe n'a pas de propriété d'identifiant.

L'attribut `unsaved-value` est important ! Si la propriété d'identifiant de votre classe n'est pas nulle par défaut, vous devriez alors spécifier cet attribut.

Il existe une déclaration alternative : `<composite-id>`. Elle permet d'accéder aux données d'une table ayant une clé composée. Nous vous conseillons fortement de ne l'utiliser que pour ce cas précis.

5.1.4.1. generator

L'élément fils obligatoire `<generator>` définit la classe Java utilisée pour générer l'identifiant unique des instances d'une classe persistante. Si des paramètres sont requis pour configurer ou initialiser l'instance du

générateur, ils seront passés via l'élément <param>.

```
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="net.sf.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

Tous les générateurs implémentent l'interface net.sf.hibernate.id.IdentifierGenerator. C'est une interface très simple ; certaines applications peuvent choisir de fournir leur propre implémentation spécifique. Cependant, Hibernate fournit plusieurs implémentations de manière native. Il y a des diminutifs pour les générateurs natifs :

increment
: génère des identifiants du type long, short ou int qui sont uniques seulement lorsqu'aucun autre process n'insère de données dans la même table. *Ne pas utiliser dans un cluster.*

identity
: supporte les colonnes identity dans DB2, MySQL, MS SQL Server, Sybase et HypersonicSQL. L'identifiant retourné est du type long, short ou int.

sequence
: utilise une séquence dans DB2, PostgreSQL, Oracle, SAP DB, McKoi ou un générateur dans Interbase. L'identifiant retourné est de type long, short ou int

hilo
: utilise l'algorithme hi/lo pour générer de manière performante les identifiants de type long, short ou int, en donnant une table et une colonne (par défaut hibernate_unique_key et next_hi) comme source des valeurs "hi". L'algorithme hi/lo génère des identifiants qui sont uniques pour une base de données donnée. *Ne pas utiliser ce générateur avec des connexions liées à JTA ou gérées par l'utilisateur.*

seqhilo
: utilise l'algorithme hi/lo pour générer les identifiants de type long, short ou int, en donnant le nom d'une séquence de base de données.

uuid.hex
: utilise l'algorithme à 128-bit UUID pour générer les identifiants de type string, uniques sur un réseau donné (l'adresse IP est utilisée). L'UUID est encodée comme une chaîne de 32 chiffres hexadécimaux.

uuid.string
: utilise le même algorithme UUID. L'UUID est encodée comme une chaine de 16 caractères ASCII (n'importe lequel). *Ne pas utiliser avec PostgreSQL.*

native
: choisit identity, sequence ou hilo en fonction des possibilités de la base de données.

assigned
: laisse l'application assigner l'identifiant de l'objet avant l'appel à save().

foreign
: utilise l'identifiant d'un autre objet associé. Généralement utilisé en conjonction d'une association <one-to-one> par clé primaire.

5.1.4.2. Algorithme Hi/Lo

Les générateurs `hilo` et `seqhilo` fournissent deux implémentations alternatives de l'algorithme hi/lo, une approche très répandue pour la génération d'identifiant. La première implémentation nécessite une table "spéciale" pour gérer la prochaine valeur "hi". La seconde utilise une séquence de type Oracle (si supportée).

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Malheureusement, vous ne pouvez utiliser `hilo` lorsque vous fournissez manuellement votre propre `Connection` à Hibernate, ou lorsqu'Hibernate utilise une datasource d'un serveur d'application enrolée dans un contexte JTA. Hibernate doit être capable de récupérer la valeur "hi" dans une nouvelle transaction (séparée de la transaction courante). Une approche classique dans un environnement EJB est d'implémenter l'algorithme hi/lo en utilisant un session bean stateless.

5.1.4.3. UUID Algorithm

L'UUID contient : l'adresse IP, la date de démarrage de la JVM (arrondie au quart de seconde), la date système et une valeur de compteur (unique pour une JVM). Il n'est pas possible d'obtenir l'adresse MAC ou l'adresse mémoire d'un code Java, ceci est donc le mieux que l'on puisse faire sans utiliser JNI.

N'essayez pas d'utiliser `uuid.string` dans PostgreSQL.

5.1.4.4. Colonne Identity et Sequences

Pour les bases de données qui supportent les colonnes identity (DB2, MySQL, Sybase, SQL Server), vous pouvez utiliser la génération de clé `identity`. Pour les bases de données qui supportent les séquences (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB), vous pouvez utiliser la génération de clé de style `sequence`. Ces deux stratégies nécessitent deux requêtes SQL pour insérer un nouvel objet.

```
<id name="id" type="long" column="uid">
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="identity"/>
</id>
```

Pour le développement multi plate formes, la stratégie `native` choisira entre `identity`, `sequence` et `hilo`, en fonction des possiblités de la base de données utilisée.

5.1.4.5. Identifiants assignés

Si vous souhaitez que l'application assigne les identifiants (en opposition à la génération faite par Hibernate), utilisez le générateur `assigned`. Ce générateur spécial utilisera la valeur de l'identifiant déjà assigné à la

propriété d'identifiant de l'objet. Attention lorsque vous utilisez cette possibilité, il faut utiliser des clés avec un sens métier (ce qui est toujours de design discutable).

A cause de leur nature même, les entités qui utilisent ce générateur ne peuvent être sauvées via la méthode saveOrUpdate de la session. Vous devez spécifier vous même si l'objet doit être sauvé ou mis à jour en appelant soit `save()` soit `update()` sur la session.

5.1.5. composite-id

```
<composite-id
        name="nomDePropriete"
        class="NomDeClasse"
        unsaved-value="any|none"
        access="field|property|NomdeClasse">

        <key-property name="nomDePropriete" type="nomdetype" column="nom_de_colonne"/>
        <key-many-to-one name="NomDePropriete" class="NomDeClasse" column="nom_de_colonne"/>
        ......
</composite-id>
```

Pour une table avec clé composée, vous pouvez mapper plusieurs propriétés de la classe comme propriétés identifiantes. L'élément `<composite-id>` accepte des mappings de propriétés via `<key-property>` et des mappings d'éléments fils via `<key-many-to-one>`.

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

Votre classe persistante *doit* surcharger `equals()` et `hashCode()` pour implémenter l'égalité des identifiants composés. Elle doit aussi implémenter `Serializable`.

Malheureusement, cette approche avec identifiant composée signifie qu'un objet persistant est son propre identifiant. Il n'y a pas d'autres "clients" potentiels que l'objet persistant lui même. Vous devez instancier une instance de la classe persistante, renseigner ses propriétés identifiantes avant de pouvoir charger (`load()`) l'état persistant associé à la clé composée. Nous décrirons une méthode plus pratique où la clé composée est implémentée dans une classe distincte dans Section 7.4, « composants en tant qu'identifiants composés ». Les attributs décris ci dessous s'appliquent uniquement à l'approche alternative:

- `name` (optionnel) : Une propriété de type composant qui contient l'identifiant composé (voir section suivante).
- `class` (optionnel - par défaut = le type de la propriété déterminé par réflexion) : La classe composant utilisée comme identifiant composé (voir section suivante).
- `unsaved-value` (optionnel - par défaut = `none`) : Indique qu'une instance transiante doit être considérée comme nouvellement instanciée, si paramétré à `any`.

5.1.6. discriminator

L'élément `<discriminator>` est requis pour la persistance polymorphique dans le cadre de la stratégie de mapping "table par hiérarchie de classe" (table-per-class-hierarchy) et spécifie une colonne discriminatrice de la table. La colonne discriminatrice contient une valeur qui indique à la couche de persistance quelle classe fille doit être instanciée pour un enregistrement particulier. Un ensemble restreint de types peut être utilisé : `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
        column="colonne_du_discriminateur"       (1)
        type="type_du_discriminateur"            (2)
```

```
        force="true|false"                      (3)
        insert="true|false"                     (4)
/>
```

(1) `column` (optionnel - par défaut = `class`) : le nom de la colonne discriminatrice.
(2) `type` (optionnel - par défaut = `string`) : un nom qui indique le type Hibernate
(3) `force` (optionnel - par défaut = `false`) : "force" Hibernate à spécifier les valeurs discriminatrices permises même lorsque toutes les instances de la classe "racine" sont récupérées.
(4) `insert` (optionnel - par défaut = `true`) : positionner le à `false` si votre colonne discriminatrice fait aussi partie d'un identifiant composé mappé.

Les différentes valeurs de la colonne discriminatrice sont spécifiées par l'attribut `discriminator-value` des éléments `<class>` et `<subclass>`.

L'attribut `force` est utile si la table contient des lignes avec d'autres valeurs qui ne sont pas mappées à une classe persistante. Ce qui ne sera généralement pas le cas.

5.1.7. version (optionnel)

L'élément `<version>` est optionnel est indique que la table contient des données versionnées. Ceci est particulièrement utile si vous prévoyez d'utiliser des transations longues (voir plus loin).

```
<version
        column="colonne_de_version"                     (1)
        name="nomDePropriete"                           (2)
        type="nomdetype"                                (3)
        access="field|property|NomDeClasse"             (4)
        unsaved-value="null|negative|undefined"         (5)
/>
```

(1) `column` (optionnel - par défaut = le nom de la propriété) : Le nom de la colonne contenant le numéro de version.
(2) `name` : Le nom de la propriété de classe persistante.
(3) `type` (optionnel - par défaut = `integer`) : Le type du numéro de version.
(4) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.
(5) `unsaved-value` (optionnel - par défaut = `undefined`) : Une valeur de la propriété "version" qui indique qu'une instance est nouvellement instanciée (non sauvegardée), qui la distingue des instances transiantes qui ont été chargées ou sauvegardées dans une session précédente (`undefined` spécifie que la propriété d'identifiant doit être utilisée).

Les numéros de version peuvent être de type `long`, `integer`, `short`, `timestamp` ou `calendar`.

5.1.8. timestamp (optionnel)

L'élément optionnel `<timestamp>` indique que la table contient des données "timestampées". C'est une alternative au versioning. Les timestamps sont par nature une implémentation moins sûre du verrou optimiste. Cependant, l'application peut parfois utiliser les timestamps dans d'autres buts.

```
<timestamp
        column="colonne_de_timestamp"       (1)
        name="nomDePropriete"               (2)
        access="field|property|NomDeClasse" (3)
        unsaved-value="null|undefined"      (4)
/>
```

(1) `column` (optionnel - par défaut = le nom de la propriété) : Le nom de la colonne contenant le timestamp.
(2) `name`: Le nom de la propriété de type Java `Date` ou `Timestamp` dans la classe persistante.
(3) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la propriété.
(4) `unsaved-value` (optionnel - par défaut = `null`) : Une valeur de la propriété "version" qui indique qu'une instance est nouvellement instanciée (non sauvegardée), qui la distingue des instances transiantes qui ont été chargées ou sauvegardées dans une session précédente (`undefined` spécifie que la propriété d'identifiant doit être utilisée).

Notez que `<timestamp>` est équivalent à `<version type="timestamp">`.

5.1.9. property

L'élément `<property>` déclare une propriété persistante de la classe, respectant la convention JavaBean.

```
<property
        name="nomDePropriete"               (1)
        column="nom_de_colonne"             (2)
        type="nomdetype"                    (3)
        update="true|false"                 (4)
        insert="true|false"                 (4)
        formula="expression SQL quelconque" (5)
        access="field|property|NomDeClasse" (6)
/>
```

(1) `name` : Le nom de la propriété, l'initiale étant en minuscule (cf conventions JavaBean).
(2) `column` (optionnel - par défaut = le nom de la propriété) : le nom de la colonne de base de données mappée.
(3) `type` (optionnel) : un nom indiquant le type Hibernate.
(4) `update, insert` (optionnel - par défaut = `true`) : spécifie que les colonnes mappées doivent être incluses dans l'ordre SQL `UPDATE` et/ou `INSERT`. Paramétrer les deux à `false` permet à la propriété d'être "dérivée", sa valeur étant initialisée par une autre propriété qui mappe la(les) même(s) colonne(s), par un trigger ou par une autre application.
(5) `formula` (optionnel) : une expression SQL qui définit une valeur pour une propriété *calculée*. Les propriétés n'ont pas de colonne mappée.
(6) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la propriété.

typename peut être :

1. Le nom d'un type Hibernate basique (ex. `integer, string, character, date, timestamp, float, binary, serializable, object, blob`).
2. Le nom d'une classe Java avec un type basique par défaut (eg. `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. Le nom d'une classe fille de `PersistentEnum` (ex. `eg.Color`).
4. Le nom d'une classe Java serialisable.
5. Le nom d'une classe Java implémentant un type personnalisé (ex. `com.illflow.type.MyCustomType`).

Si vous ne spécifiez pas de type, Hibernate utilisera la réflexion sur la propriété définie pour trouver la bonne correspondance avec le type Hibernate. Hibernate essaiera d'interpréter le nom de la classe retournée par le getter de la propriété en utilisant successivement les règles 2, 3, 4. Cependant, cela ne suffit pas toujours. Dans certains cas, vous aurez toujours besoin d'un attribut `type` (Par exemple, pour distinguer `Hibernate.DATE` de `Hibernate.TIMESTAMP`, ou pour spécifier un type personnalisé).

L'attribut `access` vous permet de contrôler la manière avec laquelle Hibernate accède à la propriété à

l'exécution. Par défaut, Hibernate utilisera les accesseurs de l'attribut. Si vous spécifiez `access="field"`, Hibernate court circuitera les accesseurs et accédera directement à l'attribut, en utilisant la réflexion. Vous pouvez spécifier votre propre stratégie en nommant une classe qui implémente l'interface `net.sf.hibernate.property.PropertyAccessor`.

5.1.10. many-to-one

Une association simple vers une autre classe persistante est déclarable en utilisant un élément `many-to-one`. Le modèle relationnel est une association many-to-one (Il s'agit au sens propre de la référence à un objet).

```
<many-to-one
        name="nomDePropriete"                                (1)
        column="nom_de_colonne"                              (2)
        class="NomDeClasse"                                  (3)
        cascade="all|none|save-update|delete"                (4)
        outer-join="true|false|auto"                         (5)
        update="true|false"                                  (6)
        insert="true|false"                                  (6)
        property-ref="nomDeProprieteDUneClasseAssociee"      (7)
        access="field|property|NomDeClasse"                  (8)
        unique="true|false"                                  (9)
/>
```

(1) `name` : Le nom de la propriété.

(2) `column` (optionnel) : Le nom de la colonne.

(3) `class` (optionnel - par défaut = au type de la propriété déterminé par réflexion) : Le nom de la classe associée.

(4) `cascade` (optional) : Spécifie quelles opérations doivent être effectuées en cascade de l'objet parent vers l'objet associé.

(5) `outer-join` (optionnel - par défaut = `auto`) : active le chargement par outer-join lorsque `hibernate.use_outer_join` est activé.

(6) `update, insert` (optionnel - par défaut = `true`) : spécifie que les colonnes mappées doivent être incluses dans l'ordre SQL `UPDATE` et/ou `INSERT`. Paramétrer les deux à `false` permet à la propriété d'être "dérivée", sa valeur étant initialisée par une autre propriété qui mappe la(les) même(s) colonne(s), par un trigger ou par une autre application.

(7) `property-ref` : (optionnel) Le nom de la propriété de la classe associée qui est liée à cette clé étrangère. Si non spécifiée, la clé primaire de la classe associée est utilisée.

(8) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.

(9) `unique` (optionnel) : Active la génération DDL d'une contrainte unique pour la colonne clé-étrangère.

L'attribut `cascade` autorise les valeurs suivantes : `all`, `save-update`, `delete`, `none`. Fixer une valeur différente de `none` propagera certaines opérations à l'objet (fils) associé. Voir "Cycle de vie de l'objet" ci dessous.

L'attribut `outer-join` accepte trois valeurs différentes :

- `auto` (par défaut) : Charge l'association en utilisant une jointure ouverte si la classe associée n'a pas de proxy.
- `true` : Charge toujours l'association en utilisant une jointure ouverte.
- `false` : Ne charge jamais l'association en utilisant une jointure ouverte.

Une déclaration typique de `many-to-one` est aussi simple que

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

L'attribut `property-ref` ne devrait être utilisé que pour mapper des données d'un système hérité (lecagy

system) où une clé étrangère fait référence à une autre clé unique de la table associée. Ce genre de modèle relationnel peut être qualifié de... laid. Par example, supposez que la classe `Product` a un numéro de série unique, qui n'est pas la clé primaire (L'attribut `unique` contrôle la génération DDL d'Hibernate avec l'outil SchemaExport).

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

Voici le mapping que `OrderItem` pourrait utiliser:

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

Cela n'est clairement pas encouragé.

5.1.11. one-to-one

Une association one-to-one vers une autre classe persistante est déclarée en utilisant un élément `one-to-one`.

```
<one-to-one
        name="nomDePropriete"                               (1)
        class="NomDeClasse"                                 (2)
        cascade="all|none|save-update|delete"               (3)
        constrained="true|false"                            (4)
        outer-join="true|false|auto"                        (5)
        property-ref="nomDeProprieteDUneClasseAssociee"     (6)
        access="field|property|NomDeClasse"                 (7)
/>
```

(1) `name` : Le nom de la propriété.

(2) `class` (optionnel - par défaut = le type de la propriété déterminée par réflexion) : le nom de la classe associée.

(3) `cascade` (optionnel) : spécifie quelles opérations doivent être réalisées en cascade de l'objet parent vers l'objet associé.

(4) `constrained` : (optionnel) spécifie qu'une contrainte sur la clé primaire de la table mappée fait référence à la table de la classe associée. Cette option affecte l'ordre dans lequel `save()` et `delete()` sont effectués en cascade (elle est aussi utilisée dans l'outil schema export).

(5) `outer-join` (optionnel - par défaut = `auto`) : Active le chargement par jointure ouverte de l'association lorsque `hibernate.use_outer_join` est activé.

(6) `property-ref`: (optionnel) : Le nom de la propriété de la classe associée qui est liée à cette clé étrangère. Si non spécifié, la clé primaire de la classe associée est utilisée.

(7) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.

Il y a deux types d'association one-to-one:

- les associations sur clé primaire
- les associations sur clé étrangère unique

Les associations sur clé primaire ne nécessitent pas de colonne supplémentaire dans la table ; si deux enregistrements sont liés par l'association alors les deux enregistrements partagent la même valeur de clé primaire. Ainsi, si vous souhaitez que deux objets soient liés par association sur clé primaire, vous devez vous assurer qu'ils aient la même valeur d'identifiant !

Pour une association par clé primaire, ajoutez les mappings suivant à `Employee` et `Person`, respectivement:

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Assurez vous que les clés primaires des lignes associées dans les tables PERSON et EMPLOYEE sont égales. Nous utilisons, dans ce cas, une stratégie de génération d'identifiant Hibernate spéciale, appelée `foreign`:

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

Une nouvelle instance de `Person` voit alors son identifiant assigné à la même valeur de clé primaire que l'instance d'`Employee` référencée par la propriété `employee` de cette `Person`.

Par ailleurs, une clé étrangère avec une contrainte d'unicité, d'`Employee` vers `Person`, peut etre déclarée comme :

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

Et cette association peut être bidirectionnelle en ajoutant ceci dans le mapping de `Person`:

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. component, dynamic-component

L'élément `<component>` mappe des propriétés d'un objet fils à des colonnes de la table de la classe parent. Les composants peuvent eux aussi déclarer leurs propres propriétés, composants ou collections. Voir "Components" plus tard.

```
<component
        name="nomDePropriete"              (1)
        class="NomDeClasse"                (2)
        insert="true|false"                (3)
        upate="true|false"                 (4)
        access="field|property|NomDeCLasse">(5)

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

(1) `name`: Le nom de la propriété.

(2) `class` (optionnel - par défaut = le type de la propriété déterminé par réflexion) : Le nom de la classe du composant (fils).

(3) `insert` : Est ce que la colonne mappée apparait dans l'`INSERT` SQL?

(4) `update` : Est ce que la colonne mappée apparait dans l'`UPDATE` SQL?

(5) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.

Les tags `<property>` fils mappent les propriétés de la classe fille aux colonnes de la table.

L'élément `<component>` accepte un sous élément `<parent>` qui mappe une propriété du composant comme

référence vers l'entité contenante.

L'élément `<dynamic-component>` accepte qu'une `Map` soit mappée comme un composant, où les noms des propriétés font référence aux clés de la map.

5.1.13. subclass

Enfin, les requêtes polymorphiques nécessitent une déclaration explicite de chaque classe héritée de la classe racine. Pour la stratégie de mapping (recommandée) table par hiérarchie de classes (table-per-class-hierarchy), la déclaration `<subclass>` est utilisée.

```
<subclass
        name="NomDeClasse"                              (1)
        discriminator-value="valeur_de_discriminant"  (2)
        proxy="InterfaceDeProxy"                        (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <property .... />
        .....
</subclass>
```

(1) `name` : Le nom complet de la classe fille.
(2) `discriminator-value` (optionnel - par défaut le nom de la classe) : Une valeur qui permet de distinguer individuellement les classes filles.
(3) `proxy` (optionnel) : Spécifie une classe ou une interface à utiliser pour le chargement tardif par proxies.
(4) `lazy` (optionnel) : Paramétrer `lazy="true"` est équivalent à définir la classe elle-même comme étant son interface de `proxy`.

Chaque classe fille peut déclarer ses propres propriétés persistantes et classes filles. Les propriétés `<version>` et `<id>` sont supposées être héritées de la classe racine. Chaque classe fille dans la hiérarchie doit définir une `discriminator-value` unique. Si aucune n'est spécifiée, le nom complet de la classe java est utilisé.

5.1.14. joined-subclass

D'autre part, une classe fille qui est persistée dans sa propre table (stratégie de mapping table par sous-classe - table-per-subclass) est déclarée en utilisant un élément `<joined-subclass>`.

```
<joined-subclass
        name="NomDeClasse"                  (1)
        proxy="InterfaceDeProxy"            (2)
        lazy="true|false"                   (3)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

        <property .... />
        .....
</subclass>
```

(1) `name` : Le nom complet de la classe fille.
(2) `proxy` (optionnel) : Spécifie une classe ou interface à utiliser pour le chargement tardif par proxy.
(3) `lazy` (optionnel) : Fixer la valeur à `lazy="true"` est équivalent à spécifier le nom de la classe comme étant l'interface de `proxy`.

Il n'y a pas de colonne discriminante pour cette stratégie de mapping. Chaque classe fille doit, cependant,

déclarer une colonne de table contenant l'identifiant de l'objet en utilisant l'élément `<key>`. Le mapping écrit en début de chapitre serait réécrit comme:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                        <key column="CAT"/>
                        <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- le mapping de Dog pourrait être ici -->
        </class>

</hibernate-mapping>
```

5.1.15. map, set, list, bag

Les collections sont décrites plus loin.

5.1.16. import

Supposez que votre application possède deux classes persistantes avec le même nom et que vous ne souhaitiez pas spécifier le nom qualifié (package) dans les requêtes Hibernate. Les classes peuvent être importées explicitement, plutôt que de compter sur `auto-import="true"`. Vous pouvez même importer les classes qui ne sont pas explicitement mappées.

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="NomDeClasse"           (1)
        rename="Alias"                (2)
/>
```

(1) `class` : Le nom complet de n'importe quelle classe.

(2) `rename` (optionnel - par défaut = le nom de la classe sans son package) : Un nom pouvant servir dans le langage de requête.

5.2. Types Hibernate

5.2.1. Entités et valeurs

Pour comprendre le comportement des différents objets (au sens java), dans le contexte d'un service de persistance, nous devons les séparer en deux groupes :

Une *entité* existe indépendamment de n'importe quel objet contenant une référence à l'entité. Ceci est contradictoire avec le modèle java habituel où un objet non référencé est un candidat pour le garbage collector. Les entités peuvent être explicitement sauvées et effacées (à l'exception que la sauvegarde et l'effacement peuvent être fait en cascade d'un objet parent vers ses enfants). C'est différent du modèle ODMG de persistance par atteignabilité - et correspond plus généralement à la façon d'utiliser les objets dans les grands systèmes. Les entités supportent les références partagées et circulaires. Elles peuvent aussi être versionnées.

Un état persistant d'une entité est constitué de références vers d'autres entités et instances de types *valeur*. Les valeurs sont des types primitifs, des collections, des composants et certains objets immuables. Contrairement aux entités, les valeurs (spécialement les collections et les composants) sont persistées et supprimées par atteignabilité. Puisque les objets de type valeur (et primitifs) sont persistés et effacés avec les entités qui les contiennent, ils ne peuvent pas être versionnés indépendamment. Les valeurs n'ont pas d'identifiant indépendant, elles ne peuvent donc pas être partagées entre deux entités ou collections.

Tous les types Hibernate, à l'exception des collections, supportent la sémantique null.

Jusqu'à présent, nous avons utilisé le terme "classes persistantes" pour faire référence aux entités. Nous allons continuer de le faire. Cependant, dans l'absolu, toutes les classes persistantes définies par un utilisateur, et ayant un état persistant, ne sont pas des entités. Un *composant* est une classe définie par l'utilisateur avec la sémantique d'une valeur.

5.2.2. Les types de valeurs basiques

Les *types basiques* peuvent etre grossièrement séparés en

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Les types effectuant le mapping entre des types primitifs Java (ou leur classes d'encapsulation) et les types de colonnes SQL appropriés (spécifique au vendeur). `boolean, yes_no` et `true_false` sont des encodages possibles pour les `boolean` Java ou `java.lang.Boolean`, sa classe encapsulante.

`string`
: Un type effectuant le mapping entre `java.lang.String` et `VARCHAR` (ou `VARCHAR2` pour Oracle).

`date, time, timestamp`
: Des types effectuant le mapping entre `java.util.Date` (et ses classes filles) et les types SQL `DATE`, `TIME` et `TIMESTAMP` (ou équivalent).

`calendar, calendar_date`
: Des types effectuant le mapping entre `java.util.Calendar` et les types SQL `TIMESTAMP` et `DATE` (ou équivalent).

`big_decimal`
: Un type effectuant le mapping entr `java.math.BigDecimal` et `NUMERIC` (ou `NUMBER` pour Oracle).

`locale, timezone, currency`
: Des types effetuant le mapping entre d'une part `java.util.Locale`, `java.util.TimeZone` et `java.util.Currency` et d'autre part `VARCHAR` (ou `VARCHAR2` pour Oracle). Les instances de `Locale` et `Currency` sont mappées à leurs codes ISO. Les instances de `TimeZone` sont mappées à leur `ID`.

`class`
: Un type effectuant le mapping entre `java.lang.Class` et `VARCHAR` (ou `VARCHAR2` pour Oracle). Une `Class` est mappée à son nom entièrement qualifié.

`binary`
: Mappe un tableau de byte vers un type binaire SQL approprié.

`text`
: Mappe de longues chaînes Java vers un type SQL `CLOB` ou `TEXT`.

`serializable`
: Mappe les types java sérialisables vers un type binaire SQL approprié. Vous pouvez aussi indiquer le type Hibernate `serializable` avec le nom de la classe java sérialisable ou d'une interface qui ne fait ni référence à un type basique ni n'implémente `PersistentEnum`.

`clob, blob`
: Mappe les types de classes JDBC `java.sql.Clob` et `java.sql.Blob`. Ces types peuvent être inopportun pour certaines applications, dans la mesure où les objets blob et clob ne peuvent être réutilisés en dehors d'une transaction (de plus, le support des drivers est plutôt inégal et imparfait).

Les identifiants des entités et collections peuvent être de tout type basique excepté `binary`, `blob` and `clob` (Les identifiants composés sont aussi admis, voir plus loin).

Les types de valeurs basiques ont des contantes `Type` correspondant dans `net.sf.hibernate.Hibernate`. Par exemple, `Hibernate.STRING` représente le type `string`.

5.2.3. Type persistant d'enumération

Un type *enum* est un concept java classique où une classe contient un (petit) nombre constant d'instances immuables. Vous pouvez créer un type enum en implémentant `net.sf.hibernate.PersistentEnum`, définissant les opérations `toInt()` et `fromInt()` :

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;
            case 1: return GINGER;
            case 2: return BLACK;
            default: throw new RuntimeException("Unknown color code");
        }
    }
}
```

Le nom du type Hibernate est simplement le nom de la classe énumérée, dans le cas présent `eg.Color`.

5.2.4. Types de valeurs personnalisés

Il est relativement facile pour les développeurs de créer leurs propres types de valeurs. Par exemple, vous voulez persister des propriétés du type `java.lang.BigInteger` vers des colonnes `VARCHAR`. Hibernate ne fournit pas nativement un type pour cela. Les types personnalisés ne sont pas restreints à mapper une propriété (ou un élément de collection) à une simple colonne de table. Vous pouvez, par exemple, avoir une propriété Java `getName()`/`setName()` de type `java.lang.String` qui est persistée dans les colonnes `FIRST_NAME`, `INITIAL`, `SURNAME`.

Pour implémenter un type personnalisé, implémentez soit `net.sf.hibernate.UserType`, ou `net.sf.hibernate.CompositeUserType` et déclarer, dans les propriétés l'utilisant, le nom complet (avec package) de la classe dans l'élement type. Voir `net.sf.hibernate.test.DoubleStringType` pour comprendre ce qu'il est possible de faire.

```
<property name="twoStrings" type="net.sf.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

Notez l'utilisation des tags `<column>` pour mapper vers plusieurs colonnes.

Hibernate fournit un large éventail de types natifs et supporte les composants, vous ne devrez avoir besoin d'un type personnalisé que dans de rares cas. Néanmoins, il est bon d'utiliser les types personnalisés pour des classes (non-entité) qui reviennent souvent dans votre application. Par exemple une classe `MonetoryAmount` est un bon candidat pour un `CompositeUserType`, puisqu'elle pourrait facilement être mappée comme composant. Un des arguments en faveur de ce choix est l'abstraction. Avec les types personnalisés vos documents de mappings n'auraient pas à être modifiés lors de possibles modifications sur la définition des valeurs monétaires.

5.2.5. Type de mappings "Any"

Il y a un dernier type de mapping de propriété. L'élément `<any>` définit une association polymorphique vers des classes de plusieurs tables. Ce type de mapping demande toujours plus d'une colonne. La première colonne contient le type de l'entité associée. Les colonnes restantes contiennent l'identifiant. Il est impossible de spécifier une contrainte de clé étrangère pour ce type d'association, il ne faut donc pas retenir cette option comme un moyen usuel de mapper des associations polymorphiques. Vous ne devez utiliser ceci que dans des cas très spécifiques (audit logs, données de session utilisateur, etc).

```
<any name="anyEntity" id-type="long" meta-type="eg.custom.Class2TablenameType">
    <column name="table_name"/>
    <column name="id"/>
</any>
```

L'attribut `meta-type` laisse l'application spécifier un type personnalisé qui mappe les valeurs des colonnes de base de données à des classes persistances qui ont comme type de propriété d'identifiant le type spécifié par `id-type`. Si le meta-type retourne une instance de `java.lang.Class`, rien d'autre n'est requis. Mais si meta-id fait référence à un type basique comme `string` ou `character`, vous devez spécifier explicitement le mapping entre les valeurs et les classes.

```
<any name="anyEntity" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="nomDepropriete"                   (1)
        id-type="nomdutypedidentifiant"         (2)
        meta-type="nomdumetatype"               (3)
        cascade="none|all|save-update"          (4)
        access="field|property|NomDeClasse"     (5)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name` : le nom de la propriété.

(2) `id-type` : le type de l'identifiant.

(3) `meta-type` (optionnel - par défaut = `class`) : un type qui mappe `java.lang.Class` à une seule colonne, ou un type admis pour un mapping de discrimination.

(4) `cascade` (optionnel - par défaut = `none`) : le style de cascade.

(5) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.

L'ancien type `object` qui avait un rôle similaire dans Hibernate 1.2 est toujours supporté, mais est désormais semi-déprécié.

5.3. identificateur SQL mis entre guillemets

Vous pouvez forcer Hibernate à placer, dans le SQL généré, les noms des tables et des colonnes entre guillemets en incluant la table ou le nom de colonne entre guillemets simples dans le document de configuration. Hibernate utilisera la syntaxe appropriée dans le SQL généré en fonction du `Dialect` (généralement des guillemets doubles, mais des crochets pour SQL Server et des guillemets inversés pour MySQL).

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.4. Fichiers de mapping modulaires

Il est possible de définir les mappings `subclass` et `joined-subclass` dans des documents de mappings séparés, directement en dessous de `hibernate-mapping`. Ceci vous permet d'étendre une hiérarchie de classes en ajoutant simplement un fichier de mapping. Vous devez spécifier l'attribut `extends` du mapping de la classe fille, nommant une classe mère déjà définie. L'utilisation de cette option rend l'ordre des documents de mapping important !

```
<hibernate-mapping>
        <subclass name="eg.subclass.DomesticCat" extends="eg.Cat" discriminator-value="D">
             <property name="name" type="string"/>
        </subclass>
</hibernate-mapping>
```

Chapitre 6. Mapping des Collections

6.1. Collections persistantes

Cette section ne contient pas beaucoup d'exemples Java. Nous supposons que vous savez déjà utiliser le framework de collections Java. Il n'y a donc pas grand chose de plus à savoir - avec quelques définitions, vous pouvez utiliser les collections Java de la même manière que vous l'avez toujours fait.

Hibernate peut persister des instances de `java.util.Map`, `java.util.Set`, `java.util.SortedMap`, `java.util.SortedSet`, `java.util.List`, et tous les tableaux d'entités et valeurs persistantes. Les propriétés de `java.util.Collection` ou `java.util.List` peuvent aussi être persistées avec la sémantique de sac (bag).

A savoir: les collections persistantes ne conservent pas de sémantique supplémentaire introduite par les implémentations de l'interface Collection (ex: l'ordre d'itération d'une `LinkedHashSet`). Les collections persistantes agissent respectivement comme `HashMap`, `HashSet`, `TreeMap`, `TreeSet` et `ArrayList` Par ailleurs, le type java de la propriété contenant la collection doit être du type de l'interface (ex: `Map`, `Set` ou `List` ; jamais `HashMap`, `TreeSet` ou `ArrayList`). Cette restriction existe parce qu'Hibernate remplace dasn votre dos vos instances de `Map`, `Set` et `List` par des instances de ses propres implémentations de `Map`, `Set` ou `List` (A ce titre, faîtes attention à l'utilisation de == sur vos collections).

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.save(cat);
kittens = cat.getKittens(); //Okay, la collection kittens est un Set
(HashSet) cat.getKittens(); //Erreur !
```

Les collections obéissent aux règles auxquelles sont soumises les types valeurs : pas de références partagées, les collections sont créées ou effacées en même temps que l'entité contenante. A cause de la nature du modèle relationnel, elles ne supportent pas la sémantique nulle; Hibernate ne distingue pas une collection nulle d'une collection vide.

Les collections sont automatiquement persistées lorsqu'elles sont référencées par un objet persistant et automatiquement effacées lorsqu'elles sont déréférencées. Si une collection est passée d'un objet persistant à un autre, ses éléments devrait être déplacés d'une table vers une autre. Vous ne devriez pas vous soucier beaucoup de cela. Vous n'avez qu'à utiliser les collections Hibernate de la même façon que les collections Java ordinaires, mais vous devez être certains de comprendre les définitions des associations bidirectionnelles (discutées plus tard) avant de les utiliser.

Les instances de collections se différencient en base de données par une clé étrangère vers l'entité contenante. Cette clé étrangère est appelée *clé de collection*. La clé de collection est mappée par l'élément `<key>`.

Les collections peuvent contenir d'autres types que ceux d'Hibernate, y compris tous les types de base, les types entités et les composants. Ceci est une définition importante : un objet dans une collection peut être traité soit avec la sémantique d'un "passage par valeur" (elle dépendra alors du propriétaire de la collection) soit être une référence à une autre entité ayant son propre cycle de vie. Les collections ne peuvent contenir d'autres collections. Le type contenu est appelé *type d'élément de collection*. Les éléments de collection sont mappés grâce à `<element>`, `<composite-element>`, `<one-to-many>`, `<many-to-many>` ou `<many-to-any>`. Les deux premiers mappent des éléments avec la sémantique de valeur, les trois autres sont utilisés pour mapper des associations avec des entités.

Toutes les collections, à l'exception de `Set` et Bag ont une colonne *index* - une colonne qui mappe vers l'index d'un tableau, d'une `List` ou une clé de `Map`. L'index de `Map` peut être de n'importe quel type de base, type entité ou même type composite (il ne peut être une collection). L'index d'un tableau ou d'une list est toujours de type `integer`. Les index sont mappés en utilisant `<index>`, `<index-many-to-many>`, `<composite-index>` ou `<index-many-to-any>`.

Il existe beaucoup de mappings différents pour les collections, couvrant plusieurs modèles relationnels. Nous vous conseillons d'essayer l'outil de génération de schéma pour assimiler comment ces déclarations se traduisent en base de données.

6.2. Mapper une Collection

Les collections sont mappées par les éléments `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` et `<primitive-array>`. `<map>` est représentatif :

```
<map
    name="nomDePropriete"                                          (1)
    table="nom_de_table"                                           (2)
    schema="nom_de_schema"                                         (3)
    lazy="true|false"                                              (4)
    inverse="true|false"                                           (5)
    cascade="all|none|save-update|delete|all-delete-orphan"        (6)
    sort="unsorted|natural|ClassDeComparateur"                     (7)
    order-by="nom_de_colonne asc|desc"                             (8)
    where="clause SQL where quelconque"                            (9)
    outer-join="true|false|auto"                                   (10)
    batch-size="N"                                                 (11)
    access="field|property|NomDeClasse"                            (12)
>

    <key .... />
    <index .... />
    <element .... />
</map>
```

(1) `name` : le nom de la prorpiété contenant la collection

(2) `table` (optionnel - par défaut = nom de la propriété) : le nom de la table de la collection (non utilisé pour les associations one-to-many)

(3) `schema` (optionnel) : le nom du schéma pour surcharger le schéma déclaré dans l'élément racine

(4) `lazy` (optionnel - par défaut = `false`) : active l'initialisation tardive (non utilisé pour les tableaux)

(5) `inverse` (optionnel - par défaut = `false`) : définit cette collection comme l'extrêmité "inverse" de l'association bidirectionnelle.

(6) `cascade` (optionnel - par défaut = `none`) : active les opérations de cascade vers les entités filles

(7) `sort` (optionnel) : spécifie une collection triée via un ordre de tri `naturel`, ou via une classe comparateur donnée (implémentant Comparator)

(8) `order-by` (optionnel, seulement à partir du JDK1.4) : spécifie une colonne de table (ou des colonnes) qui définit l'ordre d'itération de `Map`, `Set` ou Bag, avec en option `asc` ou `desc`

(9) `where` (optionnel) : spécifie une condition SQL arbitraire `WHERE` à utiliser au chargement ou à la suppression d'une collection (utile si la collection ne doit contenir qu'un sous ensemble des données disponibles)

(10) `outer-join` (optionnel) : spécifie que la collection doit être chargée en utilisant une jointure ouverte, lorsque c'est possible. Seule une collection (par `SELECT` SQL) pour être chargée avec une jointure ouverte.

(11) `batch-size` (optionnel, par défaut = `1`) : une taille de batch (batch size) utilisée pour charger plusieurs instances de cette collection en initialisation tardive.

(12) `access` (optionnel - par défaut = `property`) : La stratégie qu'Hibernate doit utiliser pour accéder à la valeur de la propriété.

Le mapping d'une `List` ou d'un tableau nécessite une colonne à part pour contenir l'index du tableau ou de la list (le `i` dans `foo[i]`). Si votre modèle relationnel n'a pas de colonne index, (par exemple si vous travaillez avec une base de données sur laquelle vous n'avez pas la main), utilisez alors un `Set` non ordonné. Cela semble aller à l'encontre de beaucoup de personnes qui pensent qu'une `List` est un moyen pratique d'accéder à une collection désordonnée. Les collections Hibernate obéissent strictement aux sémantiques des interfaces des collections `Set`, `List` et `Map`. Les éléments de `List` ne se réarrangent pas spontanément !

D'un autre côté, les personnes qui veulent utiliser une `List` pour émuler le comportement d'un *bag* (sac) ont une raison légitime. Un bag (sac) est une collection non triée, non ordonnée qui peut contenir le même élément plusieurs fois. Le framework de collections Java ne dispose pas d'une interface `Bag`, ainsi vous devez l'émuler avec une `List`. Hibernate vous permet de mapper des propriétés de type `List` ou `Collection` avec l'élément `<bag>`. Notez que la définition de bag ne fait pas partie du contrat `Collection` et qu'elle est même en conflit avec certains aspects de la définition du contrat d'une `List` (vous pouvez, cependant, trier un bag (sac) de manière aritraire, nous en discuterons plus tard).

Note : Les bags Hibernate volumineux mappé avec `inverse="false"` ne sont pas efficaces et doivent être évités ; Hibernate ne peut créer, effacer ou mettre à jour individuellement les enregistrements, puisqu'il n'y a pas de clé pouvant servir à identifier un enregistrement particulier.

6.3. Collections de valeurs et associations Plusieurs-vers-Plusieurs

Une table de collection est requise pour toute collection de valeurs et toute collection de références vers d'autres entités mappées avec une association plusieurs-vers-plusieurs (la définition naturelle d'une collection Java). La table a besoin de d'une(de) clé(s) étrangère(s), d'une(de) colonne(s) élément et si possible d'une(de) colonne(s) index.

La clé étrangère d'une table de collection vers la table de l'entité propriétaire est déclarée en utilisant l'élément `<key>`.

```
<key column="nom_de_colonne"/>
```

(1) `column` (requis) : Le nom de la colonne clé étrangère

Pour les collections indexées comme les lists et les maps, nous avons besoin d'un élément `<index>`. Pour les lists, cette colonne contient des entiers numérotés à partir de zéro. Soyez certains que votre index commence bien par zéro (surtout si vous travaillez avec une base de données existante). Pour les maps, la colonne peut contenir des valeurs de chacun des types Hibernate.

```
<index
        column="nom_de_colonne"             (1)
        type="nomdetype"                    (2)
/>
```

(1) `column` (requis) : Le nom de la colonne contenant les valeurs de l'index de la collection.
(2) `type` (optionnel, par défaut = `integer`) : Le type de l'index de la collection.

Alternativement, une map peut être indexée par des objets de type entité. Nous utilisons alors l'élément `<index-many-to-many>`.

```
<index-many-to-many
        column="nom_de_colonne"             (1)
        class="NomDeClasse"                 (2)
/>
```

(1) `column` (requis): Le nom de la colonne contenant la clé étrangère vers l'entité index de la collection.
(2) `class` (requis) : La classe entité utilisée comme index de collection.

Pour une collection de valeurs, nous utilisons l'element `<element>`.

```
<element
        column="nom_de_colonne"            (1)
        type="nomdetype"                   (2)
/>
```

(1) `column` (requis) : Le nom de la colonne contenant les valeurs des éléments de la collection.
(2) `type` (requis) : Le type d'un élément de la colleciton.

Une collection d'entités avec sa propre table correspond à la notion relationnelle d'une *association plusieurs-vers-plusieurs*. Une association plusieurs vers plusieurs est le mapping le plus naturel pour une collection Java mais n'est généralement pas le meilleur modèle relationnel.

```
<many-to-many
        column="nom_de_colonne"                         (1)
        class="NomDeClasse"                             (2)
        outer-join="true|false|auto"                    (3)
    />
```

(1) `column` (requis) : Le nom de la colonne contenant la clé étrangère de l'entité
(2) `class` (requis) : Le nom de la classe associée.
(3) `outer-join` (optionnel - par défaut = `auto`) : active le chargement par jointure ouverte pour cette association lorsque `hibernate.use_outer_join` est activé.

Quelques exemple, d'abord un set de String :

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

Un bag contenant des integers (avec un ordre d'itération déterminé par l'attribut `order-by`) :

```
<bag name="sizes" table="SIZES" order-by="SIZE ASC">
    <key column="OWNER"/>
    <element column="SIZE" type="integer"/>
</bag>
```

Un tableau d'entités - dans ce cas une association many to many (notez que les entités ont un cycle de vie, `cascade="all"`):

```
<array name="foos" table="BAR_FOOS" cascade="all">
    <key column="BAR_ID"/>
    <index column="I"/>
    <many-to-many column="FOO_ID" class="org.hibernate.Foo"/>
</array>
```

Une map d'index de String vers des Date:

```
<map name="holidays" table="holidays" schema="dbo" order-by="hol_name asc">
    <key column="id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Une List de composants (décrits dans le prochain chapitre):

```
<list name="carComponents" table="car_components">
    <key column="car_id"/>
    <index column="posn"/>
    <composite-element class="org.hibernate.car.CarComponent">
            <property name="price" type="float"/>
            <property name="type" type="org.hibernate.car.ComponentType"/>
            <property name="serialNumber" column="serial_no" type="string"/>
    </composite-element>
</list>
```

6.4. Associations Un-vers-Plusieurs

Une *association un vers plusieurs* lie les tables de deux classes *directement*, sans table de collection intermédiaire (Ceci implémente un modèle relationnel *un-vers-plusieurs*). Ce modèle relationnel perd quelques unes des sémantiques des collections Java:

- Il ne peut y avoir de valeur nulle contenue dans map, set ou list
- Une instance de la classe entité contenue ne peut appartenir à plus d'une instance de la collection
- Une instance de la classe entité contenue ne peut apparaitre dans plus d'une valeur de l'index de la collection

Une association de `Foo` vers `Bar` nécessite l'ajout d'une colonne clé et si possible d'une colonne index vers la table de la classe entité contenue, `Bar`. Ces colonnes sont mappées en utilisant les éléments `<key>` et `<index>` décrits précédemment.

Le tag `<one-to-many>` indique une assocation un vers plusieurs.

```
<one-to-many class="NomDeClasse"/>
```

(1) `class` (requis) : Le nom de la classe associée.

Exemple :

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="org.hibernate.Bar"/>
</set>
```

Notez que l'élément `<one-to-many>` n'a pas besoin de déclarer de colonne. Il n'est pas non plus nécessaire de déclarer un nom de table ou quoique ce soit.

Note importante : Si la colonne `<key>` d'une association `<one-to-many>` est déclarée `NOT NULL`, Hibernate peut provoquer des violations de contraintes lorsqu'il créé ou met à jour l'association. Pour éviter ce problème, *vous devez utiliser une association bidirectionnelle* avec l'extrémité plusieurs (set ou bag) marquéz comme `inverse="true"`. Voir la discussion sur les associations bidirectionnelles plus tard.

6.5. Initialisation tardive

Les collections (autres que les tableaux) peuvent être initialisée de manière tardives, ce qui signifie qu'elles ne chargent leur état de la base de données que lorsque l'application a besoin d'y accéder. L'initialisation intervient de manière transparente pour l'utilisateur, l'application n'a donc pas à se soucier de cela (en fait, l'initialisation transparente est la principale raison pour laquelle Hibernate a besoin de ses propres implémentations de collection). Ainsi, si l'application essaie quelque chose comme :

```
s = sessions.openSession();
User u = (User) s.find("from User u where u.name=?", userName, Hibernate.STRING).get(0);
Map permissions = u.getPermissions();
s.connection().commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Erreur !
```

Il arrivera une mauvaise surprise. Dans la mesure où les collections "permissions" n'ont pas été initialisées avant que la `Session` soit commitée, la collection ne sera jamais capable de charger son état. Pour corriger le cas précédent, il faut déplacer la ligne qui lit la collection juste avant le commit (Il existe d'autres moyens avancés de résourdre ce problème).

Une autre façon de faire est d'utilisez une collection initialisée immédiatement. Puisque l'initialisation tardive peut mener à des bogues comme le précédent, l'initialisation immédiate est le comportement par défaut. Cependant, il est préférable d'utiliser l'initialisation tardive pour la plupart des collections, spécialement pour les collections d'entités (pour des raisons de performances).

Les exceptions qui arrivent lors d'une initialisation tardive sont encapsulées dans une `LazyInitializationException`.

Déclarer une collection comme tardive en utilisant l'attribut optionnel `lazy` :

```
<set name="names" table="NAMES" lazy="true">
    <key column="group_id"/>
    <element column="NAME" type="string"/>
</set>
```

Dans certaines architectures applicatives, particulièrement quand le code qui accède aux données et celui qui les utilise ne se trouvent pas dans la même couche, on peut avoir un problème pour garantir que la session est ouverte pour l'initialisation de la collection. Il y a deux moyens clzssiques de résoudre ce problème :

- Dans une application web, un filtre de servlet peut être utilisé pour ne fermer la `Session` qu'à la fin de la requête de l'utilisateur, une fois que la vue a été rendue. Bien entendu, cela nécessite de mettre en place une gestion rigoureuse des exceptions de l'infrastructure applicative. Il est vital que la `Session` soit fermée et la transaction achevée avant le retour vers l'utilisateur, même si une exception survient pendant le rendement de la vue. Le filtre de servlet doit pouvoir accéder à la `Session` pour cette approche. Nous recommandons d'utiliser une variable `ThreadLocal` pour garder la `Session` courante (voir chapitre 1, pour un exemple d'implémentation).Section 1.4, « Jouer avec les chats »).

- Dans une application avec une couche métier séparée, la logique métier doit "préparer" toutes les collections qui seront requises par la couche web avant d'effectuer le retour. Cela signifie que la couce métier doit charger toutes les données nacessaires au cas d'utilisation qui nous occupe et les retourner à la couche de présentation/web. Généralement, l'application invoque `Hibernate.initialize()` pour chaque collection qui sera requise par l'étage web (cet appel doit être effectué avant la fermeture de la session) ou charg la collection via une requête en utilisant une clause `FETCH`.

- Vous pouvez aussi attacher un objet précédemment chargé à une nouvelle `Session` en utilisant `update()` ou `lock()` avant d'accéder aux collections non initialisées (ou autres proxys). Hibernate ne peut le faire automatiquement, cela introduirait une sémantique de transaction !

Vous pouvez utiliser la méthode `filter()` de l'API Session d'Hibernate pour avoir la taille de la collection sans l'initialiser :

```
( (Integer) s.filter( collection, "select count(*)" ).get(0) ).intValue()
```

`filter()` ou `createFilter()` sont aussi utilisés pour récupérer de manière efficace un sous ensemble d'une collection sans avoir à l'initialiser entièrement.

6.6. Collections triées

Hibernate supporte les collections qui implémentent `java.util.SortedMap` et `java.util.SortedSet`. Vous devez spécifier un comparateur dans le fichier de mapping :

```
<set name="aliases" table="person_aliases" sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

Les valeurs de l'attribut `sort` sont `unsorted`, `natural` et le nom d'une classe implémentant `java.util.Comparator`.

Les collections triées se comportent comme `java.util.TreeSet` ou `java.util.TreeMap`.

Si vous souhaitez que la base de données trie elle même les éléments d'une collection, utilisez l'attribut `order-by` des mappings de `set`, `bag` ou `map`. Cette solution n'est disponible qu'à partir du JDK 1.4 ou plus (elle est implémentée via les `LinkedHashSet` ou `LinkedHashMap`). Ceci effectue un tri dans la requête SQL, et non en mémoire dans la JVM.

```
<set name="aliases" table="person_aliases" order-by="name asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

Notez que la valeur de l'attribut `order-by` est un tri SQL et non HQL !

Les associations peuvent aussi être triées à l'exécution par des critères arbitraires en utilisant `filter()`.

```
sortedUsers = s.filter( group.getUsers(), "order by this.name" );
```

6.7. `Utiliser un <idbag>`

Si, comme nous, vous êtes complêtement d'accord sur le fait que les clés composites sont une mauvaise idée et que les entités devraient avoir des identifiants synthétiques (clés techniques), alors vous devez trouver étrange que les associations plusieurs vers plusieurs et les collections de valeurs que nous avons montrées jusqu'à présent soient toutes mappées dans des tables possédant des clés composites ! En fait, ce point est discutable ; une table d'association pure ne semble pas tirer bénéfice d'une clé technique (bien qu'une collection de valeurs composées le pourrait). Néanmoins, Hibernate propose une fonctionnalité (un peu expérimentale) qui vous permet de mapper des associations many to many et des collections de valeurs vers une table ayant une clé

technique.

L'élément `<idbag>` vous permet de mapper une `List` (ou `Collection`) avec les caractéristiques d'un bag.

```
<idbag name="lovers" table="LOVERS" lazy="true">
    <collection-id column="ID" type="long">
        <generator class="hilo"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="eg.Person" outer-join="true"/>
</idbag>
```

Comme vous pouvez le voir, un `<idbag>` possède un générateur d'id synthétique, tout comme une classe entité ! Une clé technique différente est assignée à chaque enregistrement de la collection. Hibernate ne fournit cependant pas de mécanisme pour trouver la valeur de la clé technique d'un enregistrement particulier.

Notez que la performance de mise à jour pour un `<idbag>` est *nettement* meilleure que pour un `<bag>` ! Hibernate peut localiser les enregistrements individuellement et les mettre à jour ou les effacer individuellement, comme dans une list, une map ou un set.

Dans l'implémentation courante, la génération d'identifiant `identity` n'est pas supportée pour les identifiants de collection `<idbag>`.

6.8. Associations Bidirectionnelles

Une *association bidirectionnelle* permet de naviguer à partir des deux extrémités de l'association. Les deux types d'association bidirectionnelles supportées sont :

un-vers-plusieurs
: un set ou un bag d'un côté, un simple entité de l'autre

plusieurs-vers-plusieurs
: un set ou un bag de chaque côté

Notez qu'Hibernate ne supporte pas les associations bidirectionnelles avec une collection indexée (list, map or array), vous devez utiliser un mapping set ou bag.

Vous pouvez spécifier une association plusieurs-vers-plusieurs bidirectionnelle, en mappant simplement deux associations plusieurs-vers-plusieurs à la même table d'association de la base de données et en déclarant une extrémité *inverse* (celle de votre choix). Voici un exemple d'association bidirectionnelle d'une classe vers *elle-même* (chaque categorie peut avoir plusieurs items et chaque item peut être dans plusieurs categories):

```
<class name="org.hibernate.auction.Category">
    <id name="id" column="ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM" lazy="true">
        <key column="CATEGORY_ID"/>
        <many-to-many class="org.hibernate.auction.Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="org.hibernate.auction.Item">
    <id name="id" column="ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true" lazy="true">
        <key column="ITEM_ID"/>
```

```
        <many-to-many class="org.hibernate.auction.Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

Les changement effectués uniquement sur l'extrêmité inverse ne sont *pas* persistés. Ceci signifie qu'Hibernate possède deux représentations en mémoire pour chaque association bidirectionnelle, un lien de A vers B et l'autre de B vers A. Ceci est plus facile à comprendre si vous penser au modèle objet Java et comment l'on créé une relation plusieurs-vers-plusieurs en Java:

```
category.getItems().add(item);          // La catégorie connait désormais la relation
item.getCategories().add(category);     // L'Item connait désormais la relation

session.update(item);                     // Aucun effet, rien n'est persisté !
session.update(category);                 // La relation est persistée
```

Le côté non-inverse est utilisé pour sauvegarder la réprésentation mémoire de la relation en base de données. Nous aurions un INSERT/UPDATE inutile et provoquerions probalement une violation de contrainte de clé étrangère si les deux côtés déclenchaient la mise à jour ! Ceci est également vrai pour les associations un-vers-plusieurs bidirectionnelles.

Vous pouvez mapper une association un-vers-plusieurs bidirectionnelle en mappant une association un-vers-plusieurs vers la(les) même(s) colonne(s) de table que sa relation inverse plusieurs-vers-une et en déclarant l'extrêmité plisieurs avec `inverse="true"`.

```
<class name="eg.Parent">
    <id name="id" column="id"/>
    ....
    <set name="children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="eg.Child"/>
    </set>
</class>

<class name="eg.Child">
    <id name="id" column="id"/>
    ....
    <many-to-one name="parent" class="eg.Parent" column="parent_id"/>
</class>
```

Mapper un côté d'une association avec `inverse="true"` n'impacte pas les opérations de cascade, ce sont deux concepts différents !

6.9. Associations ternaires

Il y a deux approches pour mapper une association ternaire. La première est d'utiliser des éléments composites (voir ci-dessous). La seconde est d'utiliser une `Map` ayant une association comme index :

```
<map name="contracts" lazy="true">
    <key column="employer_id"/>
    <index-many-to-many column="employee_id" class="Employee"/>
    <one-to-many column="contract_id" class="Contract"/>
</map>
```

```
<map name="connections" lazy="true">
    <key column="node1_id"/>
    <index-many-to-many column="node2_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

6.10. Associations hétérogènes

Les éléments `<many-to-any>` et `<index-many-to-any>` fournissent de vraies associations hétérogènes. Ces éléments de mapping fonctionnnent comme l'élément `<any>` - et ne devraient être utilisés que très rarement.

6.11. Exemples de collection

Les sections précédentes sont un peu confuses. Regardons un exemple, cette classe :

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}
```

possède une collection d'instances de `eg.Child`. Si chacun des child (fils) possède au plus un parent, le mapping le plus naturel est une association un-vers-plisieurs :

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Ceci mappe les définitions suivantes :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

Si le parent est *requis*, utilisez une association un-vers-plusieurs bidirectionnelle :

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
```

```
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="eg.Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

Notez la contrainte `NOT NULL` :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

D'un autre côté, si le child (fils) peut avoir plusieurs parents, une association plusieurs-vers-plusieurs est appropriée :

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="eg.Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Définitions des tables :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

Chapitre 7. Mappings des composants

La notion de *composant* est réutilisée dans différents contextes et pour différents buts dans Hibernate.

7.1. Objets dépendants

Une composant est un objet contenu et qui est persisté comme un type de valeur, pas comme une entité. Le terme "composant" fait référence à la notion de composition en programmation Orientée Objet (pas aux composants architecturaux). Par exemple, vous pourriez modéliser une personne de cette façon :

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

`Name` peut être persisté en tant que composant de `Person`. Notez que `Name` définit des méthodes getter/setter pour ses propriétés persistantes, mais n'a ni besoin de déclarer d'interface particulière, ni besoin d'une propriété

d'identifiant.

Notre mapping hibernate ressemblera à :

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- attribut class optionnel -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

La table personne contient les colonnes `pid`, `birthday`, `initial`, `first` et `last`.

Comme tous les types de valeur, les composants ne supportent par les références partagées. La sémantique de la valeur nulle d'un composant est *intrinsèque*. Lorsque l'on recharge l'objet contenu, Hibernate considèrera que si toutes les colonnes du composant sont nulles, alors le composant dans son ensemble est nul. Ce comportement devrait être appriprié dans la plupart des cas.

Les propriétés d'un composant peuvent être de n'importe quel type Hibernate (collections, association plusieurs-vers-un, autres composants, etc). Les composants dans des composants ne devraient *pas* être considérés comme exotiques. Hibernate est fait pour supporter un modèle objet très fin.

L'élément `<component>` accepte un sous-élément `<parent>` qui mappe une propriété de la classe du composant vers une référence à l'entité contenant l'élément.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name">
        <parent name="namedPerson"/> <!-- reference vers Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

7.2. Collections d'objets dependants

Les collections d'objets dépendants sont supportées (ex un tableau de type `Name`). Déclarez votre collection de composants en remplaçant la balise `<element>` par une balise `<composite-element>`.

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- attribut class requis -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

Note : si vous utilisez un `Set` d'éléments composés, il est très important d'implémenter `equals()` et `hashCode()` correctement.

Les éléments composés peuvent eux-mêmes contenir des composants mais pas de collection. Si votre composant contient d'autres composants, utiliser la balise `<nested-composite-element>`. C'est un cas plutôt exotique - une collection de composants qui eux-mêmes ont des composants. Face à cette situation vous devriez vous demander si une association un-à-plusieurs n'est pas plus adaptée. Essayez de remodeler l'élément composé en une entité - mais notez que bien que le modèle Java restera identique, le modèle relationnel et la sémantique de persistance étant légèrement différents.

Notez qu'un mapping d'éléments composites ne supporte pas les propriétés nullables lorsque vous utilisez un `<set>`. Hibernate doit utiliser chaque valeur de colonnes pour identifier un enregistrement lorsqu'il supprime les objets (il n'y a pas de colonne séparée faisant office de clé primaire dans la table des éléments composites), et ce n'est pas possible avec des valeurs nulles. Vou devez donc soit vous limiter à des propriétés non-nulles, soit choisir `<list>`, `<map>`, `<bag>` ou `<idbag>` lors de vos mappings d'éléments composites.

Un cas particulier d'élément composite est un élément composite contenant un élément `<many-to-one>`. Un tel mapping vous permet de mapper les colonnes supplémentaires d'une table d'association plusieurs-vers-plusieurs et de les rendre accessibles dans la classe de l'élément composite. L'exemple suivant est une association plusieurs-vers-plusieurs entre une `Order` (commande) et un `Item` (article) où `purchaseDate`, `price` et `quantity` sont des propriétés de l'association :

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- l'attribut classe est optionnel -->
        </composite-element>
    </set>
</class>
```

De la même façon les associations ternaires (ou quaternaires, etc...) sont possibles :

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

Les éléments composites peuvent faire partie des requêtes en utilisant la même syntaxe que celle utilisée pour les associations entre entités.

7.3. Composants pour les indexes de Map

L'élément `<composite-index>` vous permet de mapper une classe composant en tant que clé d'une `Map`. Vérifier que vous avez bien surchargé `hashCode()` et `equals()` dans la classe composant.

7.4. composants en tant qu'identifiants composés

Vous pouvez utiliser un composant comme identifiant d'une classe d'entité. Votre composant doit satisfaire

certains critères :

- Il doit implémenter `java.io.Serializable`.
- Il doit réimplémenter `equals()` et `hashCode()` de manière consistante avec la notion d'égalité d'une clé composite dans la base de données.

Vous ne pouvez pas utiliser d'`IdentifierGenerator` pour générer de clés composées. L'application doit au contraire assigner ses propres identifiants

Dans la mesure ou un identifiant composé doit être assigné avant de pouvoir sauver l'objet, on ne peut pas utiliser la propriété `unsaved-value` de l'identifiant pour distinguer les instances nouvelles des instances sauvées dans une prédédente session.

Vous pouvez à la place implémenter `Interceptor.isUnsaved()` si vous souhaitez tout de même utiliser `saveOrUpdate()` ou la sauvegarde / mise à jour en cascade. Vous pouvez également positionner l'attribut `unsaved-value` sur l'élément `<version>` (ou `<timestamp>`) pour spécifier une valeur qui indique une nouvelle instance transiante. Dans ce cas, la version de l'entité est utilisée à la place de l'identifiant (assigné), et vous n'avez pas à implémenter vous-même `Interceptor.isUnsaved()`.

Utilisez l'élément `<composite-id>` (mêmes attributs et éléments que `<component>`) au lieu de `<id>` pour la déclaration d'une classe identifiant composé :

```
<class name="eg.Foo" table"FOOS">
    <composite-id name="compId" class="eg.FooCompositeID">
        <key-property name="string"/>
        <key-property name="short"/>
        <key-property name="date" column="date_" type="date"/>
    </composite-id>
    <property name="name"/>
    ....
</class>
```

En conséquence, chaque clé étrangère vers la table `FOOS` est aussi composée. Vous devez déclarez ceci dans les mappings de et vers les autres classes. Une association vers `Foo` serait déclarée comme :

```
<many-to-one name="foo" class="eg.Foo">
<!-- l'attribut class est optionnel, comme d'habitude -->
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-one>
```

Le nouvel élément `<column>` est aussi utilisé par les types personnalisés à multiple colonnes. C'est une alternative à l'attribut `column`. Une collection avec des éléments de type `Foo` utiliserait :

```
<set name="foos">
    <key column="owner_id"/>
    <many-to-many class="eg.Foo">
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </many-to-many>
</set>
```

Comme d'habitude, `<one-to-many>`, ne déclare pas de colonne.

Si `Foo` contient lui même des collections, elles auront aussi besoin d'une clé étrangère composée.

```
<class name="eg.Foo">
    ....
```

```
    ....
    <set name="dates" lazy="true">
        <key>   <!-- une collection hérite du type de clé composite -->
            <column name="foo_string"/>
            <column name="foo_short"/>
            <column name="foo_date"/>
        </key>
        <element column="foo_date" type="date"/>
    </set>
</class>
```

7.5. Composants dynamiques

Vous pouvez même mapper une propriété de type `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO"/>
    <property name="bar" column="BAR"/>
    <many-to-one name="baz" class="eg.Baz" column="BAZ"/>
</dynamic-component>
```

La définition d'un mapping de `<dynamic-component>` est identique à `<component>`. L'avantage de ce type de mapping est la possibilité de déterminer les propriétés réelles du bean au moment du déploiement, en éditant simplement le document de mapping (la manipulation à l'éxécution du document de mapping est aussi possible, via un parseur DOM).

Chapitre 8. Mapping de l'héritage de classe

8.1. Les trois stratégies

Hibernate supporte les trois stratégies d'héritage de base.

- une table par hiérarchie de classe (table per class hierarchy)
- une table par classe fille (table per subclass)
- une table par classe concrête (table per concrete class, avec limitations)

Il est même possible d'utiliser différentes stratégies de mapping pour différentes branches d'une même hiérarchie d'héritage, mais les mêmes limitations, que celle rencontrées dans la stratégie une table par classe concrète, s'appliquent. Hibernate ne supporte pas le mélange des mappings `<subclass>` et `<joined-subclass>` dans un même élément `<class>`.

Supposons que nous ayons une interface `Payment`, implémentée par `CreditCardPayment`, `CashPayment`, `ChequePayment`. La stratégie une table par hiérarchie serait :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Une seule table est requise. Une grande limitation de cette stratégie est que les colonnes déclarées par les classes filles ne peuvent avoir de contrainte `NOT NULL`.

La stratégie une table par classe fille serait :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
```

```
    </joined-subclass>
</class>
```

Quatre tables sont requises. Les trois tables des classes filles ont une clé primaire associée à la table classe mère (le modèle relationnel est une association un-vers-un).

Notez que l'implémentation Hibernate de la stratégie un table par classe fille ne nécessite pas de colonne discriminante dans la table classe mère. D'autres implémentations de mappers Objet/Relationnel utilisent une autre implémentation de la stratégie une table par classe fille qui nécessite une colonne de type discriminant dans la table de la classe mère. L'approche prise par Hibernate est plus difficile à implémenter mais plus correcte d'une point de vue relationnel.

Pour chacune de ces deux stratégies de mapping, une association polymorphique vers `Payment` est mappée en utilisant `<many-to-one>`.

```
<many-to-one name="payment"
    column="PAYMENT"
    class="Payment"/>
```

La stratégie une table par classe concrète est très différente.

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

Trois tables sont requises. Notez que l'interface `Payment` n'est jamais explicitement nommée. A la place, nous utilisons le *polymorphisme implicite* d'Hibernate. Notez aussi que les propriétés de `Payment` sont mappées dans chacune des classes filles.

Dans ce cas, une association polymorphique vers `Payment` est mappée en utilisant `<any>`.

```
<any name="payment"
        meta-type="class"
        id-type="long">
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

Il serait plus judicieux de définir un `UserType` comme `meta-type`, pour gérer le mapping entre une chaîne de caractère discriminante et les classes filles de `Payment`.

```
<any name="payment"
```

```
        meta-type="PaymentMetaType"
        id-type="long">
    <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH or CHEQUE -->
    <column name="PAYMENT_ID"/>
</any>
```

Il y a une autre chose à savoir sur ce mapping. Dans la mesure où chaque classe fille est mappée dans leur propre élément `<class>` (et puisque `Payment` n'est qu'une interface), chacune des classes filles peut facilement faire partie d'une autre stratégie d'héritage que cela soit une table par hiérarchie ou une table par classe fille ! (et vous pouvez toujours utiliser des requêtes polymorphiques vers l'interface) `Payment`).

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

Encore une fois, nous ne mentionnons pas explicitement `Payment`. Si nous exécutons une requête sur l'interface `Payment` - par exemple, `from Payment` - Hibernate retournera automatiquement les instances de `CreditCardPayment` (et ses classes filles puisqu'elles implémentent aussi `Payment`), `CashPayment` et `ChequePayment` mais pas les instances de `NonelectronicTransaction`.

8.2. Limitations

Hibernate suppose qu'une association mappe exactement une colonne clé étrangère. Plusieurs associations par clé étrangère sont tolérées (vous pouvez avoir besoin de spécifier `inverse="true"` ou `insert="false" update="false"`), mais il n'est pas possible de mapper chaque association vers plusieurs clés étrangères. Ceci signifie que :

- quand une association est modifiée, c'est toujours la même clé étrangère qui est mise à jour
- quand une association est chargée de manière tardive, une seule requête à la base de données est utilisée
- quand une association est chargée immédiatement, elle peut l'être en utilisant une simple jointure ouverte

Ceci implique que les associations polymorphiques un-vers-plusieurs vers des classes mappées en utilisant la stratégie une table par classe concrète *ne sont pas supportées* (charger ces associations nécessiterait de

multiples requêtes ou jointures ouverte).

Le tableau montre les limitations des mappings une table par classe concrète, et du polymorphique implicite, avec Hibernate.

Tableau 8.1. Caractéristiques des stratégies d'héritages

Stratégie d'héritage	Plusieurs-vers-un polymorphiques	Un-vers-un polymorphiques	Un-vers-plusieurs polymorphiques	Plusieurs-vers-plusieurs polymorphiques	`s.get()` polymorphiques	Requêtes polymorphiques	Jointures polymorphiques
une table par hiérarchie de classe	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
une table par classe fille	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
une table par classe concrète (polymorphisme implicite)	`<any>`	*non supportés*	*non supportés*	`<many-to-any>`	*utiliser un requête*	`from Payment p`	*non supportées*

Chapitre 9. Manipuler les données persistantes

9.1. Création d'un objet persistant

Un objet (une instance entité) est *transiant* ou *persistant* pour une `Session` donnée. Les objets nouvellement instanciés sont bien sûr transiants. La session offre les services de sauvegarde (de persistence) des instances transiantes :

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

`save()` avec un seul argument, génère et assigne un identifiant unique à `fritz`. La même méthode avec deux arguments essaie de persister `pk` en utilisant l'identifiant donné. Généralement, nous vous déconseillons l'utilisation de la version à deux arguments puisqu'elle pourrait être utilisée pour créer des clés primaires avec une signification métier. Elle est plus efficace, dans certaines situations, comme l'utilisation d'Hibernate pour la persistance d'un Entity Bean BMP.

Les objets associés peuvent être persistés dans l'ordre que vous voulez du moment que vous n'avez pas de contrainte `NOT NULL` sur une clé étrangère. Il n'y a aucun risque de violation de contrainte de clé étrangère. Cependant, vous pourriez violer une contrainte `NOT NULL` si vous invoquiez `save()` sur des objets dans le mauvais ordre.

9.2. Chargement d'un objet

La méthode `load()` offerte par la `Session` vous permet de récupérer une instance persistante si vous connaissez son identifiant. Une des versions prend comme argument un objet class et charge l'état dans un objet nouvellement instancié. La seconde version permet d'alimenter une instance dans laquelle l'état sera chargé. La version qui prend comme argument une instance est particulèrement utile si vous pensez utiliser Hibernate avec des Entity Bean BMP, elle est fournie dans ce but. Vous découvrirez d'autres cas où l'utiliser (Pooling d'instance maison, etc.)

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// il est nécessaire de transformer les identifiants primitifs
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new Long(pkId) );
```

```
Cat cat = new DomesticCat();
// charge l'état de pk dans cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Il est à noter que `load()` lèvera une exception irréversible s'il ne trouve pas d'enregistrement correspondant en base données. Si la classe est mappée avec un proxy, `load()` retourne un objet qui est un proxy non initialisé et n'interrogera la base de données qu'à la première invocation d'une méthode de l'objet. Ce comportement est très utile si vous voulez créer une association vers un objet sans réellement le charger depuis la base de données.

Si vous n'êtes par certain que l'enregistrement correspondant existe, vous devriez utiliser la méthode `get()`, qui interroge immédiatement la base de données et retourne null s'il n'y a aucun enregistrement correspondant.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

Vous pouvez aussi charger un objet en utilisant un ordre SQL de type `SELECT ... FOR UPDATE`. Référez vous à la section suivante pour une présentation des `LockMode`s d'Hibernate.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Notez que les instances associées ou collections contenues ne sont *pas* selectionnées en utilisant "`FOR UPDATE`".

Il est possible de recharger un objet et toutes ses collections à n'importe quel moment en utilisant la méthode `refresh()`. Ceci est utile quand les triggers d'une base de données sont utilisés pour initialiser certaines propriétés de l'objet.

```
sess.save(cat);
sess.flush(); //force l'ordre SQL INSERT
sess.refresh(cat); //recharge l'état (après exécution des triggers)
```

9.3. Requêtage

Si vous ne connaissez pas le(s) identifiant(s) de l'objet (ou des objets) que vous recherchez, utlisez la méthode `find()` offerte par la `Session`. Hibernate s'appuie sur un langage d'interrogation, orienté objet, simple mais puissant.

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);

List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate " +
    "where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate.bithdate is null" );

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
```

```
    izi,
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birthday is null"
);
```

Le second argument de `find()` est un objet ou un tableau d'objets. Le troisième argument est un type Hibernate ou un tableau de types Hibernate. Ces types passés en argument sont utilisés pour lier les objets passés en argument au `?` de la requête (ce qui correspond aux IN parameters d'un `PreparedStatement` JDBC). Comme en JDBC, il est préférable d'utiliser ce mécanisme de liaison (binding) plutôt que la manipulation de chaîne de caractères.

La classe `Hibernate` définit un certain nombre de méthodes statiques et de constantes, proposant l'accès à la plupart des types utilisés, comme les instances de `net.sf.hibernate.type.Type`.

Si vous pensez que votre requête retournera un très grand nombre d'objets, mais que vous n'avez pas l'intention de tous les utliser, vous pourriez améliorer les performances en utilisant la méthode `iterate()`, qui retourne un `java.util.Iterator`. L'itérateur chargera les objets à la demande en utilisant les identifiants retounés par la requête SQL initiale (ce qui fait un total de n+1 selects).

```
// itération sur les ids
Iterator iter = sess.iterate("from eg.Qux q order by q.likeliness");
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // récupération de l'objet
    // condition non définissable dans la requête
    if ( qux.calculateComplicatedAlgorithm() ) {
        // effacez l'instance en cours
        iter.remove();
        // n'est plus nécessaire pour faire le reste du process
        break;
    }
}
```

Malheureusement, `java.util.Iterator` ne déclare aucune exception, donc les exceptions SQL ou Hibernate qui seront soulevées seront transformées en `LazyInitializationException` (une classe fille de `RuntimeException`).

La méthode `iterate()` est également plus performante si vous prévoyez que beaucoup d'objets soient déjà chargés et donc disponibles via la session, ou si le résultat de la requête retourne très souvent les mêmes objets (quand les données ne sont pas en cache et ne sont pas dupliqués dans le résultat, `find()` est presque toujours plus rapide). Voici un exemple de requête qui devrait être appelée via la méthode `iterate()` :

```
Iterator iter = sess.iterate(
    "select customer, product " +
    "from Customer customer, " +
    "Product product " +
    "join customer.purchases purchase " +
    "where product = purchase.product"
);
```

Invoquer la requête précédente avec `find()` retournerait un `ResultSet` JDBC très volumineux et contenant plusieurs fois les mêmes données.

Les requêtes Hibernate retournent parfois des tuples d'objets, dans ce cas chaque tuple est retourné sous forme de tableau (d'objets) :

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = (Foo) tuple[0]; Bar bar = (Bar) tuple[1];
    ....
}
```

9.3.1. Requêtes scalaires

Les requêtes peuvent spécifier une propriété d'une classe dans la clause `select`. Elles peuvent même appeler les fonctions SQL d'aggrégation. Ces propriétés ou aggrégations sont considérées comme des résultats "scalaires".

```
Iterator results = sess.iterate(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color"
);
while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

```
Iterator iter = sess.iterate(
    "select cat.type, cat.birthdate, cat.name from DomesticCat cat"
);
```

```
List list = sess.find(
    "select cat, cat.mate.name from DomesticCat cat"
);
```

9.3.2. L'interface de requêtage Query

Si vous avez besoin de définir des limites sur le résultat d'une requête (nombre maximum d'enregistrements et / ou l'indice du premier résultat que vous souhaitez récupérer), utilisez une instance de `net.sf.hibernate.Query` :

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Vous pouvez même définir une requête nommée dans le document de mapping. N'oubliez pas qu'il faut utiliser une section `CDATA` si votre requête contient des caractères qui pourraient être interprétés comme un marqueur XML.

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

L'interface d'interrogation supporte l'utilisation de paramètres nommés. Les paramètres nommés sont des variables de la forme `:name` que l'on peut retrouver dans la requête. `Query` dispose de méthodes pour lier des valeurs à ces paramètres nommés ou aux paramètres `?` du style JDBC. *Contrairement à JDBC, l'indice des paramètres Hibernate démarre de zéro.* Les avantages des paramètres nommés sont :

- les paramètres nommés sont indépendants de l'ordre dans lequel ils apparaissent dans la requête
- ils peuvent être présents plusieurs fois dans une même requête
- ils sont auto-documentés (par leur nom)

```
//paramètre nommé (préféré)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//paramètre positionné
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//paramètre nommé liste
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

9.3.3. Iteration scrollable

Si votre driver JDBC supporte les `ResultSets` scrollables, l'interface `Query` peut être utilisée pour obtenir des `ScrollableResults` qui permettent une navigation plus flexible sur les résultats.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // cherche le premier 'name' de chaque page pour une liste de 'cats' triée par 'name'
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Retourne la première page de 'cats'
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
```

Le comportement de `scroll()` est similaire à celui d'`iterate()`, à la différence près que les objets peuvent être initialisés de manière sélective avec `get(int)`, au lieu d'une initialisation complète d'une ligne de resultset.

9.3.4. Filtrer les collections

Un filtre (*filter*) de collection est un type spécial de requête qui peut être appliqué à une collection ou un tableau

persistant. La requête peut faire référence à `this`, ce qui signifie "l'élément de la collection courante".

```
Collection blackKittens = session.filter(
    pk.getKittens(), "where this.color = ?", Color.BLACK, Hibernate.enum(Color.class)
);
```

La collection retournée est considérée comme un bag.

Remarquez que les filtres n'ont pas besoin de clause `from` (bien qu'ils puissent en avoir une si nécessaire). Les filtres ne sont pas limités à retourner des éléments de la collection qu'ils filtrent.

```
Collection blackKittenMates = session.filter(
    pk.getKittens(), "select this.mate where this.color = eg.Color.BLACK"
);
```

9.3.5. Les requêtes par critères

HQL est extrêmement puissant mais certaines personnnes préfèreront construire leurs requêtes dynamiquement, en utilisant une API orientée objet, plutôt qu'une chaîne de caractères dans leur code JAVA. Pour ces personnes, Hibernate fournit `Criteria` : une API d'interrogation intuitive.

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq("color", eg.Color.BLACK) );
crit.setMaxResults(10);
List cats = crit.list();
```

Si vous n'êtes pas à l'aise avec les syntaxes type SQL, c'est peut être la manière la plus simple de commencer avec Hibernate. Cette API est aussi plus extensible que le HQL. Les applications peuvent s'appuyer sur leur propre implémentation de l'interface `Criterion`.

9.3.6. Requêtes en SQL natif

Vous pouvez construire votre requête en SQL, en utilisant `createSQLQuery()`. Il est nécessaire de placer vos alias SQL entre accolades.

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT AS {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT AS {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

Les requêtes SQL peuvent contenir des paramètres nommés et positionnés, comme dans les requêtes Hibernate.

9.4. Mise à jour des objets

9.4.1. Mise à jour dans la même session

Les instances transactionnelles persistantes (objets chargés, sauvegardés, créés ou résultats d'une recherche par la `Session`) peuvent être manipulées par l'application. Toute modification sur un état persistant sera sauvegardée (persistée) quand la `Session` sera *flushée* (ceci sera décrit plus tard dans ce chapitre). Le moyen le plus simple de modifier l'état d'un objet est donc de le charger (`load()`), et de le manipuler pendant que la `Session` est ouverte :

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // les modifications de 'cat' sont automatiquement détectées et sauvegardées
```

Il arrive que cette approche ne convienne pas puisqu'elle nécessite une même session pour exécuter les deux ordres SQL `SELECT` (pour charger l'objet) et `UPDATE` (pour sauvegarder son état mis à jour) dans la même session. Hibernate propose une méthode alternative.

9.4.2. Mise à jour d'objets détachés

Certaines applications ont besoin de récupérer un objet dans une transaction, de le passer ensuite à la couche de présentation pour modification, et enfin de le sauvegarder dans une nouvelle transaction (les applications suivant cette approche se trouvent dans un contexte d'accès aux données hautement concurrent, elles utilisent généralement des données versionnées pour assurer l'isolation des transactions). Cette approche nécessite un modèle de développement légèrement différent de celui décrit dans la section précedente. Hibernate supporte ce modèle en proposant la méthode `Session.update()`.

```
// dans la première session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// dans une couche supérieure de l'application
cat.setMate(potentialMate);

// plus tard, dans une nouvelle session
secondSession.update(cat);  // mise à jour de 'cat'
secondSession.update(mate); // mise à jour de  'mate'
```

Si `Cat` avec l'identifiant `catId` avait déja été chargé par `secondSession` au moment où l'application essaie de le mettre à jour, une exception aurait été soulevée.

L'application devrait unitairement mettre à jour (`update()`) les instances transiantes accessibles depuis l'instance transiante donnée si et *seulement* si elle souhaite que leur état soit aussi mis à jour (A l'exception des objets engagés dans un cycle de vie dont nous parlerons plus tard).

Les utilisateurs d'Hibernate ont émis le souhait de pouvoir soit sauvegarder une instance transiante en générant un nouvel identifiant, soit mettre à jour son état en utilisant son identifiant courant. La méthode `saveOrUpdate()` implémente cette fonctionnalité.

Hibernate distingue les "nouvelles" (non sauvegardées) instances, des instances existantes (sauvegardées ou chargées dans une session précédente) grâce à la valeur de leur propriété d'identifiant (ou de version, ou de timestamp). L'attribut `unsaved-value` du mapping `<id>` (ou `<version>`, ou `<timestamp>`) spécifie quelle valeur doit être interprétée comme représentant une "nouvelle" instance.

```
<id name="id" type="long" column="uid" unsaved-value="null">
    <generator class="hilo"/>
</id>
```

Les valeurs permises pour `unsaved-value` sont :

- `any` - toujours sauvegarder (save)
- `none` - toujours mettre à jour (update)
- `null` - sauvegarder (save) quand l'identifiant est nul (valeur par défaut)
- une valeur valide pour l'identifiant - sauvegarder (save) quand l'identifiant est nul ou égal à cette valeur
- `undefined` - par défaut pour `version` ou `timestamp`, le contrôle sur l'identifiant est alors utilisé

```
// dans la première session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// dans une couche supérieure de l'application
Cat mate = new Cat();
cat.setMate(mate);

// plus tard, dans une nouvelle session
secondSession.saveOrUpdate(cat);   // mise à jour de l'état existant (cat a un id non null)
secondSession.saveOrUpdate(mate);  // sauvegarde d'une nouvelle instance (mate a un id null)
```

L'utilisation et la sémantique de `saveOrUpdate()` semble confuse pour les nouveaux utilisateurs. Tout d'abord, tant que vous n'essayez pas d'utiliser une instance d'une session dans une autre session, il est inutile d'utiliser `update()` ou `saveOrUpdate()`. De pan entiers d'applications n'utiliseront aucune de ces deux méthodes.

Généralement `update()` ou `saveOrUpdate()` sont utilisés dans les scénarii suivant:

- l'application charge un objet dans une première session
- l'objet est passé à la couche UI
- l'objet subit quelques modificatons
- l'objet redescend vers la couche métier
- l'application persiste ces modifications en appelant `update()` dans une seconde session

`saveOrUpdate()` réalise ce qui suit :

- si l'objet est déjà persistant dans la session en cours, ne fait rien
- si l'objet n'a pas d'identifiant, elle le `save()`
- si l'identifiant de l'objet corresponds au critère défini par `unsaved-value`, elle le `save()`
- si l'objet est versionné (`version` ou `timestamp`), alors la version est vérifiée en priorité sur l'identifiant, sauf si `unsaved-value="undefined"` (valeur par défaut) est utilisé pour la version
- si un autre objet associé à la session a le même identifiant, une exception est soulevée

9.4.3. Réassocier des objets détachés

La méthode `lock()` permet à l'application de réassocier un objet non modifié avec une nouvelle session.

```
//simple réassociation :
sess.lock(fritz, LockMode.NONE);
//vérifie la version, puis ré associe :
sess.lock(izi, LockMode.READ);
//vérifie la version en utilisant SELECT ... FOR UPDATE, puis réassocie :
sess.lock(pk, LockMode.UPGRADE);
```

9.5. Suppression d'objets persistants

`Session.delete()` supprimera l'état d'un objet de la base de données. Evidemment, votre application peut toujours contenir une référence à cet objet. La meilleure façon de l'apréhender est donc de se dire que `delete()` transforme une instance persistante en instance transiante.

```
sess.delete(cat);
```

Vous pouvez aussi effacer plusieurs objets en passant une requête Hibernante à la méthode `delete()`.

Vous pouvez supprimer les objets dans l'ordre que vous souhaitez, sans risque de violer une contrainte de clé étrangère. Cependant, vous pourriez violer une contrainte `NOT NULL` si vous invoquiez `delete()` sur des objets dans le mauvais ordre.

9.6. Flush

La `Session` exécute parfois les ordres SQL nécessaires pour synchroniser l'état de la connexion JDBC avec l'état des objets contenus en mémoire. Ce processus, *flush*, se déclenche :

- à certaines invocations de `find()` ou d'`iterate()`
- à l'appel de `net.sf.hibernate.Transaction.commit()`
- à l'appel de `Session.flush()`

Les ordres SQL sont exécutés dans cet ordre :

1. toutes les insertions d'entités, dans le même ordre que celui utilisé pour la sauvegarde (`Session.save()`) des objets correpsondants
2. toutes les mises à jour d'entités
3. toutes les suppressions de collection
4. toutes les suppressions, insertions, mises à jour d'éléments de collection
5. toutes les insertions de collection
6. toutes les suppressions d'entités, dans le même ordre que celui utilisé pour la suppression (`Session.delete()`) des objets correpsondants

(Une exception existe pour les objets utilisant les générations d'ID `native` puisqu'ils sont insérés quand ils sont sauvegardés).

A moins d'appeler explicitement `flush()`, il n'y a aucune garantie sur le moment *où* la `Session` exécute les appels JDBC, seul l'*ordre* dans lequel ils sont appelés est garanti. Cependant, Hibernate garantit que les méthodes `Session.find(..)` ne retourneront jamais de données périmées ; ni de données érronées.

Il est possible de changer les comportements par défaut pour que le flush s'exécute moins fréquement. La classe `FlushMode` définit trois modeds différents. Ceci est utile pour des transactions en lecture seule où il peut être utlisé pour accroître (très) légèrement les performances.

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); //autorise les requêtes à retourner des données corrompues
Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);
// exécute quelques requêtes....
sess.find("from Cat as cat left outer join cat.kittens kitten");
//les modifications de 'izi' ne sont sont pas flushées!
...
tx.commit(); //flush s'exécute
```

9.7. Terminer une Session

La fin d'une session implique quatre phases :

- flush de la session
- commit de la transaction

- fermeture de la session
- traitement des exceptions

9.7.1. Flusher la Session

Si vous utilisez l'API `Transaction`, vous n'avez pas à vous soucier de cette étape. Elle sera automatiquement réalisée à l'appel du commit de la transaction. Autrement, vous devez invoquer `Session.flush()` pour vous assurer que les changements sont synchronisés avec la base de données.

9.7.2. Commit de la transaction de la base de données

Si vous utilisez l'API `Transaction` d'Hibernate, cela donne :

```
tx.commit(); // flush la Session et commit la transaction
```

Si vous gérez vous-même les transactions JDBC, vous devez manuellement appeler la méthode `commit()` de la connexion JDBC.

```
sess.flush();
sess.connection().commit();  // pas nécessaire pour une datasource JTA
```

Si vous décidez de ne *pas* committer vos modifications :

```
tx.rollback();  // rollback la transaction
```

ou :

```
// pas nécessaire pour une datasource JTA mais important dans le cas contraire
sess.connection().rollback();
```

Si vous faites un rollback d'une transaction vous devriez immédiatement la fermer et arrêter d'utiliser la session courante, ceci pour assurer l'intégrité de l'état interne d'Hibernate.

9.7.3. Fermeture de la Session

Un appel de `Session.close()` marque la fin d'une session. La conséquence principale de `close()` est que la connexion JDBC est relachée par la session.

```
tx.commit();
sess.close();
```

```
sess.flush();
sess.connection().commit();  // pas nécessaire pour une datasource JTA
sess.close();
```

Si vous gérez vous même votre connexion, `close()` retourne une référence à cette connexion, vous pouvez ainsi la fermer manuellement ou la rendre au pool. Si ce n'est pas le cas `close()` rend la connexion au pool.

9.8. Traitement des exceptions

Hibernate, au cours de son utilisation, peut lever des exceptions, généralement des `hibernateExceptions`. La cause éventuelle de l'exception qui peut être récupérée en utilisant `getCause()`.

Si la `Session` lève une exception, vous devrez immédiatement effectuer un rollback de la transaction, appeler `session.close()` et ne plus utiliser l'instance courante de la `Session`. Certaines méthodes de `Session` ne laisseront *pas* la session dans un état consistant. Cela signifie que toutes les exceptions levées par Hibernate doivent être considérées comme fatales, vous pourriez donc envisager de convertir l'exception non runtime `HibernateException` en `RuntimeException` (la solution la plus simple est de changer la clause `extends` dans `HibernateException.java` et de recompiler le tout). Notez que le fait que l'exception ne soit pas runtime est dû à une erreur des premiers ages d'Hibernate et sera corrigée dans la prochaine version majeure.

Hibernate essaiera de convertir les `SQLException`s levées lors des interactions avec la base de données en des sous-classes de `JDBCException`. La `SQLException` sous-jacente est accessible en appelant `JDBCException.getCause()`. Hibernate convertit la `SQLException` en une sous-classe appropriée de `JDBCException` en s'appuyant sur le `SQLExceptionConverter` attaché à la `SessionFactory`. Par défaut, le `SQLExceptionConverter` utilisé est celui défini par le dialecte ; il est cependant possible d'attacher une implémentation spécifique (voir la javadoc de `SQLExceptionConverterFactory` pour plus de détails). Les sous-types standards de `JDBCException` sont :

- `JDBCConnectionException` - indique une erreur lors de la communication JDBC sous-jacente.
- `SQLGrammarException` - indique une erreur de grammaire ou de syntaxe du SQL envoyé.
- `ConstraintViolationException` - indique une forme de violation de contrainte d'intégrité.
- `LockAcquisitionException` - indique une erreur lors de l'acquisition d'un niveau de verrou requis pour exécuter l'opération demandée.
- `GenericJDBCException` - indique une exception générique dont la cause ne tombe pas dans les catégories précédentes.

Comme toujours, toutes les exceptions sont considérées comme fatales à la `Session` et à la transaction courante. Le fait qu'Hibernate sache maintenant mieux distinguer les différents types de `SQLException` n'implique en aucune manière que les exceptions soient récupérables du point de vue de la `Session`. La hiérarchie typée des exceptions permet à l'application de mieux réagir en catégorisant la cause de l'exception plus simplement si besoin.

Il est recommandé d'effectuer le traitement des exceptions comme suit :

```
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();
    // faire qqch
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    sess.close();
}
```

Ou, si vous gérez les transactions JDBC manuellement :

```
Session sess = factory.openSession();
try {
    // faire qqch
    ...
    sess.flush();
    sess.connection().commit();
}
catch (Exception e) {
    sess.connection().rollback();
    throw e;
```

```
}
finally {
    sess.close();
}
```

Ou, si vous utilisez une datasource couplée avec JTA :

```
UserTransaction ut = .... ;
Session sess = factory.openSession();
try {
    // faire qqch
    ...
    sess.flush();
}
catch (Exception e) {
    ut.setRollbackOnly();
    throw e;
}
finally {
    sess.close();
}
```

N'oubliez pas qu'un serveur d'applications (dans un environnement géré par JTA) ne rollback les transactions automatiquement que pour les exceptions `java.lang.RuntimeExceptions`. Si une exception applicative est levée (à savoir une exception non runtime `HibernateException`), vous devez appeler la méthode `setRollbackOnly()` d'`EJBContext` vous-même ou, comme montré dans l'exemple précédent, l'encapsuler dans une `RuntimeException` (par exemple `EJBException` pour un rollback automatique.

9.9. Cycles de vie et graphes d'objets

Pour sauvegarder ou mettre à jour tous les objets contenus dans un graphe d'objets associés, vous pouvez soit

- invoquer individuellement `save()`, `saveOrUpdate()` ou `update()` sur chaque objet OU
- lier des objets en utilisant `cascade="all"` ou `cascade="save-update"`.

De même, pour supprimer tous les objets d'un graphe, vous pouvez soit

- invoquer individuellement `delete()` sur chaque objet OU
- lier des objets en utilisant `cascade="all"`, `cascade="all-delete-orphan"` or `cascade="delete"`.

Recommandation :

- Si la durée de vie de l'objet fils est liée à celle de l'objet père, faîtes en un *objet lié au cycle de vie* en spécifiant `cascade="all"`.
- Autrement, invoquez explicitement `save()` et `delete()` dans le code de l'application. Si vous souhaitez vraiment éviter de taper du code supplémentaire, utilisez `cascade="save-update"` et appeler explicitement `delete()`.

Mapper une assocation (plusieurs-vers-une, ou une collection) avec `cascade="all"` définit l'association comme une relation de type *parent/fils* où la sauvegarde/mise à jour/suppression du parent engendre la sauvegarde/mise à jour/suppression du ou des fils. Par ailleurs, la simple référence à un fils depuis un parent persistant engendrera une sauvegarde/mise à jour de l'enfant. La métaphore est cependant incomplète. Un fils qui n'est plus référencé par son père n'est *pas* automatiquement supprimé, sauf dans le cas d'une association `<one-to-many>` mappée avec `cascade="all-delete-orphan"`. Les définitions précises des opérations en cascade sont les suivantes :

- Si un parent est sauvegardé, tous ces fils sont passés à `saveOrUpdate()`

- Si un parent est passé à `update()` ou à `saveOrUpdate()`, tous ces fils sont passés à `saveOrUpdate()`
- Si un fils transiant devient référencé par un parent persistant, il est passé à `saveOrUpdate()`
- Si le parent est supprimé, tous ces enfants sont passés à `delete()`
- Si un enfant transiant est déréférencé par un parent persistant, *rien ne se passe* (l'application devra explicitement supprimer l'enfant si nécessaire) sauf si `cascade="all-delete-orphan"` est positionné, dans ce cas le fils "orphelin" est supprimé

Hibernate n'implémente pas complètement la persistence par atteignabilité, ce qui aurait pour conséquence (inefficace) la garbage collection des objets persistants. Cependant, en raison de la demande, Hibernate supporte la notion de persistance d'entités lorsqu'elles sont référencées par un autre objet persistant. Les associations définies avec `cascade="save-update"` ont ce comportement. Si vous souhaitez utliser cette approche dans toute votre application, il est plus facile de spécifier l'attribut `default-cascade` de l'élément `<hibernate-mapping>`.

9.10. Intercepteurs

L'interface `Interceptor` fournit des "callbacks" de la session vers l'application permettant à l'application de consulter et / ou manipuler des propriétés d'un objet persistant avant qu'il soit sauvegardé, mis à jour, supprimé ou chargé. Une utilisation possible de cette fonctionnalité est de tracer l'accès à l'information. Par exemple, l'`Interceptor` qui suit va automatiquement positionner le `createTimestamp` quand un `Auditable` est créé et mettre à jour la propriété `lastUpdateTimestamp` quand un `Auditable` est modifié.

```
package net.sf.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import net.sf.hibernate.Interceptor;
import net.sf.hibernate.type.Type;

public class AuditInterceptor implements Interceptor, Serializable {

    private int updates;
    private int creates;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // ne rien faire
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }
```

```
    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void postFlush(Iterator entities) {
        System.out.println("Creations: " + creates + ", Updates: " + updates);
    }

    public void preFlush(Iterator entities) {
        updates=0;
        creates=0;
    }

    ......
    ......

}
```

L'intercepteur doit être spécifié quand la session est créée.

```
Session session = sf.openSession( new AuditInterceptor() );
```

Vous pouvez aussi activer un intercepteur pour toutes les sessions d'une SessionFactory, en utilisant la `Configuration` :

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

9.11. API d'accès aux métadonnées

Hibernate a besoin d'un meta-modèle très riche de toutes les entités et types de valeurs. Parfois, ce modèle est très utile à l'application elle même. Par exemple, l'application peut utiliser les métadonnées d'Hibernate pour implémenter un algorithme "intelligent" de copie qui comprend quels objets doivent être copiés (valeurs de types muables) et quels objets ne peuvent l'être (valeurs de types imuables, et éventuellement les entités associées).

Hibernate expose les métadonnées au travers des interfaces `ClassMetadata` et `CollectionMetadata` et la hiérarchie de `Type`. Les instances des interfaces de métadonnées peuvent être obtenues depuis la `SessionFactory`.

```
Cat fritz = ......;
```

```
Long id = (Long) catMeta.getIdentifier(fritz);
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);
Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();
// retourne une Map de toutes les propriétés qui ne sont pas des collections ou des associations
// TODO: what about components?
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Chapitre 10. Transactions et accès concurrents

Hibernate n'est pas une base de données en lui même. C'est un outil léger de mapping objet relationnel. La gestion des transactions est déléguée à la connexion à base de données sous-jacente. Si la connexion est enregistrée dans JTA, les opérations effectuées pas la `Session` sont des parties atomiques de la transaction JTA. On peut voir Hibernate comme une fine surcouche de JDBC qui lui ajouterait les sémantiques objet.

10.1. Configurations, Sessions et Fabriques (Factories)

Une `SessionFactory` est un objet threadsafe, couteux à créer, prévu pour être partagé par tous les threads de l'application. Une `Session` est un objet non threadsafe, non coûteux qui ne doit être utilisé qu'une fois, pour un process métier donné, puis détruit. Par exemple, lorsque vous utilisez Hibernate dans une application à base de servlets, les servlets peuvent obtenir une `SessionFactory` en utilisant

```
SessionFactory sf = (SessionFactory)getServletContext().getAttribute("my.session.factory");
```

Chaque appel de service créé une nouvelle `Session`, la `flush()`, `commit()` sa connexion, la `close()` et finalement la libère (La `SessionFactory` peut aussi être référencée dans le JNDI ou dans une variable statique *Singleton*).

Dans un bean session sans état, une approche similaire peut être utilisée. Le bean obtiendra une `SessionFactory` dans `setSessionContext()`. Ensuite, chaque méthode métier créera une `Session`, appelera `flush()` puis `close()`. Ben sûr, l'application n'a pas à appeler `commit()` sur la connexion.(Laissez cela à JTA, la connexion à la base de données participe automatiquement aux transactions gérées par le container).

Nous utilisons l'API `Transaction` d'Hibernate comme décrit précédemment. Un simple `commit()` de la `Transaction` Hibernate "flush" l'état et committe chaque connexion à la base de données associée (en gérant de manière particulière les transactions JTA).

Assurez vous de bien comprendre le sens de `flush()`. L'opération Flush() permet de synchroniser la source de données persistante avec les modifications en mémoire mais *pas* l'inverse. Notez que pour toutes les connexions/transactions JDBC utilisées par Hibernate, le niveau d'isolation de transaction pour ces connexions s'applique à toutes les opérations effectuées par Hibernate !

Les sections suivantes traitent d'approches alternatives qui utilisent le versioning pour garantir l'atomicité de la transaction. Elles sont considérées comme des techniques "avancées", et donc à utiliser en sachant ce que l'on fait.

10.2. Threads et connections

Vous devez respecter les règles suivantes lorsque vous créez des Sessions Hibernate :

- Ne jamais créer plus d'une instance concurrente de `Session` ou `Transaction` par connexion à la base de données.
- Soyez extrêmement rigoureux lorsque vous créez plus d'une `Session` par base de données par transaction. La `Session` traçant elle-même les modifications faites sur les objets chargés, une autre `Session` pourrait voir des données corrompues.
- La `Session` n'est *pas* threadsafe ! Deux thread concurrents ne doivent jamais accéder à la même `Session` . Généralement, la `Session` doit être considérée comme une unité de travail unitaire !

10.3. Comprendre l'identité d'un objet

L'application peut accéder de manière concurrente à la même entité persistente via deux unités de travail différentes. Cependant, une instance de classe persistante n'est jamais partagée par deux instances `Session`. Il y a donc deux notions d'identité différentes.

Identité dans la base de données

```
foo.getId().equals( bar.getId() )
```

Identité dans la JVM

```
foo==bar
```

Pour les objets rattachés à une `Session` *donnée*, les deux notions sont identiques. Cependant, puisque l'application peut accéder de manière concurrente au "même" (identité persistante - dans la base de données) objet métier par deux sessions différentes, les deux instances seront en fait "différentes" (identité dans JVM).

Cette approche laisse la gestion de la concurrence à Hibernate et à la base de données. L'application n'aura jamais besoin de synchroniser un objet métier tant qu'elle s'en tient à un thread par `Session` ou à l'identité d'un objet (dans une `Session`, l'application peut utiliser sans risque `==` pour comparer deux objets).

10.4. Gestion de la concurrence par contrôle optimiste

Beaucoup de traitements métiers nécessitent une série d'interactions avec l'utilisateur entrecoupées d'accès à la base de données. Dans les applications web et les applications d'entreprise, il n'est pas acceptable qu'une transaction de base de données se déroule le temps de plusieurs interactions avec l'utilisateur.

La couche applicative prend dont en partie la responsabilité de maintenir l'isolation des traitements métier. C'est pourquoi, nous appelons ce processus une *transaction applicative*. Une transaction applicative pourra s'étendre sur plusieurs transactions à la base de données. Elle sera atomique si seule la dernière des transactions à la base de données enregistre les données mises à jour, les autres ne faisant que des accès en lecture.

La seule statégie remplissant les critères de concurrence et scalabitité élevées est le contrôle optimiste de la concurrence en appliquant des versions aux données : on utilisera par la suite le néologisme versionnage. Hibernate fournit trois approches pour écrire des applications basées sur la concurrence optimiste.

10.4.1. Session longue avec versionnage automatique

Une seule instance de `Session` et ses instances persistantes sont utilisées pour toute la transaction d'application.

La `Session` utilise le vérouillage optimiste pour s'assurer que plusieurs transactions à la base de données ne soient vues par l'application que comme une seule transaction logique (transaction applicative). La `Session` est déconnectée de sa connexion JDBC lorsqu'elle est en attente d'interaction avec l'utilisateur. Cette approche est la plus efficace en terme d'accès à la base de données. L'application n'a pas à ce soucier de la vérification de version ou du réattachement d'instaces détachées.

```
// foo est une instance chargée plus tôt par la Session
session.reconnect();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

L'objet `foo` sait par quelle `Session` il a été chargé. Dès que la `Session` obtient une connexion JDBC, un commit sera fait sur les modifications apportées à l'objet.

Ce pattern est problématique si la `Session` est trop volumineuse pour être stockées pendant le temps de réflexion de l'utilisateur, par exemple il est souhaitable qu'une `HttpSession` reste aussi petite que possible. Comme la `Session` est aussi le premier niveau de cache et contient tous les objets chargés, il n'est probablement possible de n'utiliser cette stratégie que pour des cycles contenant peu de requêtes/réponses. C'est, en fait, recommandé puisque la `Session` risquerait très vite de contenir des données obsolètes.

10.4.2. Plusieurs sessions avec versionnage automatique

Chaque interaction avec la base de données se fait dans une nouvelle `Session`. Cependant, les mêmes instances persistantes sont réutilisées pour chaque interaction à la base de données. L'application manipule l'état des instances détachées, chargées à l'initialement par une autre `Session`, puis "réassociées" en utilisant `Session.update()` ou `Session.saveOrUpdate()`.

```
// foo est une instance chargée plus tôt par une autre Session
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
session.flush();
session.connection().commit();
session.close();
```

Vous pouvez aussi appeler `lock()` au lieu de `update()` et utiliser `LockMode.READ` (effectuant un contrôle de version en court circuitant tous les caches) si vous êtes sûrs que l'objet n'a pas été modifié.

10.4.3. Contrôle de version de manière applicative

Chaque interaction avec la base de données se fait dans une nouvelle `Session` qui recharge toutes les instances persistantes depuis la base de données avant de les manipuler. Cette approche force l'application à assurer son propre contrôle de version pour garantir l'isolation de la transaction d'application (bien sur, Hibernate continuera de mettre à jour les numéros de version pour vous). Cette approche est la moins performante en terme d'accès à la base de données. Elle est ressemble plus à celle utilisée par les EJBs entités.

```
// foo est une instance chargée plus tôt par une autre Session
session = factory.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

Evidemment, si vous vous trouvez dans un environnement avec peu de concurrence sur les données et que vous n'avez pas besoin de contrôle de version, vous pouvez utiliser cette méthode en retirant simplement le contrôle de version.

10.5. Déconnexion de Session

La première approche décrite ci dessus consiste à maintenir une seule `Session` pour tout un process métier qui englobe plusieurs interactions avec l'utilisateur (par exemple, une servlet peut stocker une `Session` dans l'`HttpSession` de l'utilisateur). Pour des raisons de performance, il est préférable

1. d'effectuer un commit de la `Transaction` (ou de la connexion JDBC) puis
2. déconnecter la `Session` de la connexion JDBC

avant d'attendre l'activité de l'utilisateur. La méthode `Session.disconnect()` déconnectera la session de la connexion JDBC et la retournera au pool (à moins que vous ne fournissiez la connexion).

`Session.reconnect()` obtient une nouvelle connexion (ou vous devez en fournir une) et redémarre la session. Après reconnexion, pour forcer le contrôle de version sur les données que vous ne modifiez pas, vous pouvez appeler `Session.lock()` sur les objets susceptibles d'avoir été modifiés par une autre transaction. Vous n'avez pas besoin de vérouiller (lock) les données que vous *êtes en train* de modifier.

Voici un exemple :

```
SessionFactory sessions;
List fooList;
Bar bar;
....
Session s = sessions.openSession();

Transaction tx = null;
try {
    tx = s.beginTransaction();

    fooList = s.find(
            "select foo from eg.Foo foo where foo.Date = current date"
        //utilisation de la fonction date de DB2
    );
    bar = (Bar) s.save(Bar.class);

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    s.close();
    throw e;
}
s.disconnect();
```

Puis :

```
s.reconnect();

try {
    tx = s.beginTransaction();

    bar.setFooTable( new HashMap() );
    Iterator iter = fooList.iterator();
    while ( iter.hasNext() ) {
        Foo foo = (Foo) iter.next();
        s.lock(foo, LockMode.READ);    //vérifie que foo n'est pas obsolète
        bar.getFooTable().put( foo.getName(), foo );
    }

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    s.close();
}
```

Vous pouvez voir que la relation entre les `Transactions` et les `Sessions` est de type plusieurs-vers-une. Une `Session` représente une conversation entre l'application et la base de données. La `Transaction` divise cette

conversation en plusieurs unités atomiques de travail au niveau de la base de données.

10.6. Vérouillage pessimiste

Il n'est pas prévu que les utilisateurs passent beaucoup de temps à se soucier des stratégies de verrou. Il est généralement suffisant de spécifier le niveau d'isolation pour les connexions JDBC puis de laisser la base de données faire le travail. Cependant, les utilisateurs avancés veulent parfois obtenir des verrous pessimistes exclusifs, ou réobtenir les verrous au début d'une nouvelle transaction.

Hibernate utilisera toujours les mécanismes de vérouillage de la base de données, il ne vérouillera jamais les objets en mémoire !

La classe `LockMode` définit les niveaux de verrou qui peuvent être obtenus par Hibernate. Un verrou est obtenu pas les mécanismes suivant ;

- `LockMode.WRITE` est obtenu automatiquement lorsqu'Hibernate insère ou modifie un enregistrement.
- `LockMode.UPGRADE` peut être obtenu à la demande explicite de l'utilsateur en utilisant la syntaxe `SELECT ... FOR UPDATE` sur les bases de données qui la supportent.
- `LockMode.UPGRADE_NOWAIT` peut être obtenu à la demande explicite de l'utilsateur grâce à la syntaxte `SELECT ... FOR UPDATE NOWAIT` sous Oracle.
- `LockMode.READ` est obtenu automatiquement lorsqu'Hibernate consulte des données avec des niveaux d'isolation de type lectures reproductibles (repeatable read) ou de type sérialisable (serializable). Peut être réobtenu à la demande explicite de l'utilsateur
- `LockMode.NONE` représente l'absence de verrou. Tous les objets basculent à ce verrou à la fin d'une `Transaction`. Les objets associés à la session via l'appel de `update()` ou `saveOrUpdate()` démarrent aussi sur ce mode de verrou.

La "demande explicite de l'utilsateur" se traduit par les moyens suivants :

- un appel de `Session.load()`, spécifiant un mode de verrou (`LockMode`).
- un appel de `Session.lock()`.
- un appel de `Query.setLockMode()`.

Si `Session.load()` est appelée avec `UPGRADE` ou `UPGRADE_NOWAIT`, et que l'obet demandé n'a pas encore été chargé par la session, l'objet sera chargé en utilisant `SELECT ... FOR UPDATE`. Si `load()` est appelé et que l'objet a déja été chargé avec un mode moins restrictif, Hibernate appelle `lock()` pour cet objet.

`Session.lock()` effectue un contrôle de version si le mode de verrou spécifié est `READ`, `UPGRADE` ou `UPGRADE_NOWAIT` (Dans le cas de `UPGRADE` ou `UPGRADE_NOWAIT`, `SELECT ... FOR UPDATE` est utilisé).

Si la base de données ne supporte pas le mode de verrou demandé, Hibernate utilisera un mode approchant approprié (au lieu de lancer une exception). Ce qui garantit la portabilité des applications

Chapitre 11. HQL: Langage de requêtage d'Hibernate

Hibernate fourni un langage d'interrogation extrêmement puissant qui ressemble (et c'est voulu) au SQL. Mais ne soyez pas distraits par la syntaxe ; HQL est totalement orienté objet, comprenant des notions d'héritage, de polymorphisme et d'association.

11.1. Sensibilité à la casse

Les requêtes sont insensibles à la casse, à l'exception des noms des classes Java et des propriétés. Ainsi, `SeLeCT` est identique à `sELEct` et à `SELECT` mais `net.sf.hibernate.eg.FOO` n'est pas identique `net.sf.hibernate.eg.Foo` et `foo.barSet` n'est pas identique à `foo.BARSET`.

Ce guide utilise les mots clés HQL en minuscule. Certains utilisateurs trouvent les requêtes écrites avec les mots clés en majuscule plus lisibles, mais nous trouvons cette convention pénible lorsqu'elle est lue dans du code Java.

11.2. La clause from

La requête Hibernate la plus simple est de la forme :

```
from eg.Cat
```

qui retourne simplement toutes les instances de la classe `eg.Cat`.

La plupart du temps, vous devrez assigner un *alias* puisque vous voudrez faire référence à `Cat` dans d'autres parties de la requête.

```
from eg.Cat as cat
```

Cette requête assigne l'alias `cat` à l'instance `Cat`, nous pouvons donc utiliser cet alias ailleurs dans la requête. Le mot clé `as` est optionnel ; nous aurions pu écrire

```
from eg.Cat cat
```

Plusieurs classes peuvent apparaître, ce qui conduira à un produit cartésien (encore appelé jointures croisées).

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

C'est une bonne pratique que de nommer les alias dans les requêtes en utilisant l'initiale en miniscule, ce qui a le mérite d'être en phase avec les standards de nommage Java pour les variables locales (`domesticCat`).

11.3. Associations et jointures

On peut aussi assigner des alias à des entités associées, ou même aux éléments d'une collection de valeurs, en utilisant un `join` (jointure).

```
from eg.Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

from eg.Cat as cat left join cat.mate.kittens as kittens

from Formula form full join form.parameter param
```

Les types de jointures supportées sont celles de ANSI SQL

- `inner join` (jointure fermée)
- `left outer join` (jointure ouverte par la gauche)
- `right outer join` (jointure ouverte par la droite)
- `full join` (jointure ouverte totalement - généralement inutile)

Les constructions des jointures `inner join`, `left outer join` et `right outer join` peuvent être abbrégées.

```
from eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Par ailleurs, une jointure "fetchée" (rapportée) permet d'initialiser les associations ou collections de valeurs en même temps que leur objet parent, le tout n'utilisant qu'un seul Select. Ceci est particulièrement utile dans le cas des collections. Ce système permet de surcharger les déclarations "lazy" et "outer-join" des fichiers de mapping pour les associations et collections.

```
from eg.Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Une jointure "fetchée" (rapportée) n'a généralement pas besoin de se voir assigner un alias puisque les objets associés n'ont pas à être utilisés dans les autres clauses. Notez aussi que les objets associés ne sont pas retournés directement dans le résultat de la requête mais l'on peut y accéder via l'objet parent.

Notez que, dans l'implémentation courante, seule une seule collection peut être "fetchée" par requête (une autre stratégie ne serait pas performante). Notez aussi que le mot-clé `fetch` ne peut pas être utilisé lorsque l'on appelle `scroll()` ou `iterate()`. Notez enfin que `full join fetch` et `right join fetch` ne sont pas utiles en général.

11.4. La clause select

La clause `select` sélectionne les objets et propriétés qui doivent être retournés dans le résultat de la requête. Soit :

```
select mate
from eg.Cat as cat
    inner join cat.mate as mate
```

La requête recherchera les `mate`s liés aux `Cat`s. Vous pouvez explimer la requête d'une manière plus compacte :

```
select cat.mate from eg.Cat cat
```

Vous pouvez même sélectionner les éléments d'une collection en utilisant la fonction `elements`. La requête suivante retourne tous les kittens de chaque cat.

```
select elements(cat.kittens) from eg.Cat cat
```

Les requêtes peuvent retourner des propriétés de n'importe quel type, même celles de type composant (component).

```
select cat.name from eg.DomesticCat cat
where cat.name like 'fri%'

select cust.name.firstName from Customer as cust
```

Les requêtes peuvent retourner plusieurs objets et/ou propriétés sous la forme d'un tableau du type `Object[]`

```
select mother, offspr, mate.name
from eg.DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

ou sous la forme d'un objet Java typé

```
select new Family(mother, mate, offspr)
from eg.DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

à condition que la classe `Family` possède le constructeur approprié.

11.5. Fonctions d'aggrégation

Les requêtes HQL peuvent aussi retourner le résultat de fonctions d'aggrégation sur les propriétés :

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from eg.Cat cat
```

Les collections peuvent aussi apparaître à l'intérieur des fonctions d'aggrégation dans la clause `select`

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

Les fonctions supportées sont

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

Les mots clé `distinct` et `all` peuvent être utilisés et ont la même signification qu'en SQL.

```
select distinct cat.name from eg.Cat cat

select count(distinct cat.name), count(cat) from eg.Cat cat
```

11.6. Requêtes polymorphiques

Un requête comme :

```
from eg.Cat as cat
```

retourne non seuleument les instances de `Cat`, mais aussi celles des sous classes comme `DomesticCat`. Les

requêtes Hibernate peuvent nommer n'importe quelle classe ou interface Java dans la clause `from`. La requête retournera les instances de toutes les classes persistantes qui étendent cette classe ou implémente cette interface. La requête suivante retournera tous les objets persistants :

```
from java.lang.Object o
```

L'interface `Named` peut être implémentée par plusieurs classes persistantes:

```
from eg.Named n, eg.Named m where n.name = m.name
```

Notez que ces deux dernières requêtes nécessitent plus d'un `SELECT` SQL. Ce qui signifie que la clause `order by` ne trie pas correctement la totalité des résultats (cela signifie aussi que vous ne pouvez exécuter ces requêtes en appelant `Query.scroll()`).

11.7. La clause where

La clause `where` vous permet de réduire la liste des instances retournées.

```
from eg.Cat as cat where cat.name='Fritz'
```

retourne les instances de `Cat` dont name est égale à 'Fritz'.

```
select foo
from eg.Foo foo, eg.Bar bar
where foo.startDate = bar.date
```

retournera les instances de `Foo` pour lesquelles il existe une instance de `bar` avec la propriété `date` est égale à la propriété `startDate` de `Foo`. Les expressions utilisant la navigation rendent la clause `where` extrêmement puissante. Soit :

```
from eg.Cat cat where cat.mate.name is not null
```

Cette requête se traduit en SQL par une jointure interne à une table. Si vous souhaitez écrire quelque chose comme :

```
from eg.Foo foo
where foo.bar.baz.customer.address.city is not null
```

vous finiriez avec une requête qui nécessiterait quatre jointures en SQL.

L'opérateur = peut être utilisé pour comparer aussi bien des propriétés que des instances :

```
from eg.Cat cat, eg.Cat rival where cat.mate = rival.mate

select cat, mate
from eg.Cat cat, eg.Cat mate
where cat.mate = mate
```

La propriété spéciale (en minuscule) `id` peut être utilisée pour faire référence à l'identifiant d'un objet (vous pouvez aussi utiliser le nom de cette propriété).

```
from eg.Cat as cat where cat.id = 123

from eg.Cat as cat where cat.mate.id = 69
```

La seconde requête est particulièrement efficace. Aucune jointure n'est nécessaire !

Les propriétés d'un identifiant composé peuvent aussi être utilisées. Supposez que `Person` ait un identifiant composé de `country` et `medicareNumber`.

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456

from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Une fois de plus, la seconde requête ne nécessite pas de jointure.

De même, la propriété spéciale `class` interroge la valeur discriminante d'une instance dans le cas d'une persistance polymorphique. Le nom d'une classe Java incorporée dans la clause where sera traduite par sa valeur discriminante.

```
from eg.Cat cat where cat.class = eg.DomesticCat
```

Vous pouvez aussi spécifier les propriétés des composants ou types utilisateurs composés (components, composite user types etc). N'essayez jamais d'utiliser un expression de navigation qui se terminerait par une propriété de type composant (qui est différent d'une propriété d'un composant). Par exemple, si `store.owner` est une entité avec un composant `address`

```
store.owner.address.city    // correct
store.owner.address         // erreur!
```

Un type "any" possède les propriétés spéciales `id` et `class`, qui nous permettent d'exprimer une jointure de la manière suivante (où `AuditLog.item` est une propriété mappée avec `<any>`).

```
from eg.AuditLog log, eg.Payment payment
where log.item.class = 'eg.Payment' and log.item.id = payment.id
```

Dans la requête précédente, notez que `log.item.class` et `payment.class` feraient référence à des valeurs de colonnes de la base de données complètement différentes.

11.8. Expressions

Les expressions permises dans la clause `where` incluent la plupart des choses que vous pouvez utiliser en SQL :

- opérateurs mathématiques `+, -, *, /`
- opérateur de comparaison binaire `=, >=, <=, <>, !=, like`
- opérateurs logiques `and, or, not`
- concatenation de chaîne de caractères ||
- fonctions SQL scalaires comme `upper()` et `lower()`
- Parenthèses `( )` indiquant un regroupement
- `in`, `between`, `is null`
- paramètres JDBC IN `?`
- paramètres nommées `:name`, `:start_date`, `:x1`
- littéral SQL `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- Constantes Java `public static final` `eg.Color.TABBY`

`in` et `between` peuvent être utilisés comme suit :

```
from eg.DomesticCat cat where cat.name between 'A' and 'B'

from eg.DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

et la forme négative peut être écrite

```
from eg.DomesticCat cat where cat.name not between 'A' and 'B'

from eg.DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

De même, `is null` et `is not null` peuvent être utilisés pour tester les valeurs nulle.

Les Booléens peuvent être facilement utilisés en déclarant les substitutions de requêtes dans la configuration Hibernate :

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

Ce qui remplacera les mots clés `true` et `false` par `1` et `0` dans la traduction SQL du HQL suivant:

```
from eg.Cat cat where cat.alive = true
```

Vous pouvez tester la taille d'une collection par la propriété spéciale `size`, ou la fonction spéciale `size()`.

```
from eg.Cat cat where cat.kittens.size > 0

from eg.Cat cat where size(cat.kittens) > 0
```

Pour les collections indexées, vous pouvez faire référence aux indices minimum et maximum en utilisant `minIndex` and `maxIndex`. De manière similaire, vous pouvez faire référence aux éléments minimum et maximum d'une collection de type basiques en utilisant `minElement` et `maxElement`.

```
from Calendar cal where cal.holidays.maxElement > current date
```

Ceci existe aussi sous forme de fonctions (qui, contrairement à l'écriture précédente, n'est pas sensible à la casse):

```
from Order order where maxindex(order.items) > 100

from Order order where minelement(order.items) > 10000
```

Les fonctions SQL `any, some, all, exists, in` supportent que leur soient passées l'élément, l'index d'une collection (fonctions `elements` et `indices`) ou le résultat d'une sous requête (voir ci dessous).

```
select mother from eg.Cat as mother, eg.Cat as kit
where kit in elements(foo.kittens)

select p from eg.NameList list, eg.Person p
where p.name = some elements(list.names)

from eg.Cat cat where exists elements(cat.kittens)

from eg.Player p where 3 > all elements(p.scores)

from eg.Show show where 'fizard' in indices(show.acts)
```

Notez que l'écriture de - `size`, `elements`, `indices`, `minIndex`, `maxIndex`, `minElement`, `maxElement` - ont un

usage restreint :

- dans la clause `where` : uniquement pour les bases de données qui supportent les requêtes imbriquées (sous requêtes)
- dans la clause `select` : uniquement `elements` et `indices` ont un sens

Les éléments de collections indexées (arrays, lists, maps) peuvent être référencés via index (dans une clause where seuleuement) :

```
from Order order where order.items[0].id = 1234

select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar

select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11

select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

L'expression entre `[]` peut même être une expression arithmétique.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL propose aussi une fonction `index()` interne, pour les éléments d'une association one-to-many ou d'une collections de valeurs.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Les fonctions SQL scalaires supportées par la base de données utilisée peuvent être utilisées

```
from eg.DomesticCat cat where upper(cat.name) like 'FRI%'
```

Si vous n'êtes pas encore convaincu par tout cela, imaginez la taille et l'illisibilité qui caractériseraient la transformation SQL de la requête HQL suivante :

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

Un indice : cela donnerait quelque chose comme

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
```

```
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

11.9. La clause order by

La liste retounée par la requête peut être triée par n'importe quelle propriété de la classe ou du composant retourné :

```
from eg.DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

Le mot optionnel `asc` ou `desc` indique respectivement si le tri doit être croissant ou décroissant.

11.10. La clause group by

Si la requête retourne des valeurs aggrégées, celles ci peuvent être groupées par propriété ou composant :

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color

select foo.id, avg( elements(foo.names) ), max( indices(foo.names) )
from eg.Foo foo
group by foo.id
```

Note: vous pouvez aussi utiliser l'écriture `elements` et `indices` dans une clause select, même pour des bases de données qui ne supportent pas les sous requêtes.

Une clause `having` est aussi permise.

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Les fonctions SQL et les fonctions d'aggrégations sont permises dans les clauses `having` et `order by`, si elles sont supportées par la base de données (ce que ne fait pas MySQL par exemple).

```
select cat
from eg.Cat cat
    join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Notez que ni la clause `group by` ni la clause `order by` ne peuvent contenir d'expressions arithmétiques.

11.11. Sous requêtes

Pour les bases de données le supportant, Hibernate supporte les sous requêtes dans les requêtes. Une sous requête doit être entre parenthèses (souvent pour un appel à une fonction d'agrégation SQL) Même les sous requêtes corrélées (celles qui font référence à un alias de la requête principale) sont supportées.

```
from eg.Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from eg.DomesticCat cat
)

from eg.DomesticCat as cat
where cat.name = some (
    select name.nickName from eg.Name as name
)

from eg.Cat as cat
where not exists (
    from eg.Cat as mate where mate.mate = cat
)

from eg.DomesticCat as cat
where cat.name not in (
    select name.nickName from eg.Name as name
)
```

11.12. Exemples HQL

Les requêtes Hibernate peuvent être relativement puissantes et complexes. En fait, la puissance du langage de requêtage est l'un des avantages principaux d'Hibernate. Voici quelques exemples très similaires aux requêtes que nous avons utilisées lors d'un récent projet. Notez que la plupart des requêtes que vous écrirez seront plus simples que les exemples suivantes !

La requête suivante retourne l'id de commande (order), le nombre d'articles (items) et la valeur totale de la commande (order) pour toutes les commandes non payées d'un client (customer) particulier pour un total minimum donné, le tout trié par la valeur totale. La requête SQL générée sur les tables `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` et `PRICE` est composée de quatre jointures interne ainsi que d'une sous requête non corrélée.

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

Quel monstre ! En principe, nous ne sommes pas très fan des sous requêtes, la requête ressemblait donc plutôt à cela :

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
```

```
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

La requête suivante compte le nombre de paiements (payments) pour chaque status, en excluant les paiements dans le status `AWAITING_APPROVAL` où le changement de status le plus récent à été fait par l'utilisateur courant. En SQL, cette requête effectue deux jointures internes et des sous requêtes corrélées sur les tables `PAYMENT`, `PAYMENT_STATUS` et `PAYMENT_STATUS_CHANGE`.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

Si nous avions mappé la collection `statusChanges` comme une list, au lieu d'un set, la requête aurait été plus facile à écrire.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

La requête qui suit utilise la fonction de MS SQL `isNull()` pour retourner tous les comptes (accounts) et paiements (payments) impayés pour l'organisation à laquelle l'uilisateur (user) courant appartient. Elle est traduite en SQL par trois jointures internes, une jointure externe ainsi qu'une sous requête sur les tables `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` et `ORG_USER`.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

Pour d'autres base de données, nous aurions dû faire sans la sous requête (corrélée)

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

11.13. Trucs & Astuces

Vous pouvez compter le nombre de résultats d'une requête sans les retourner :

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

Pour trier les résultats par la taille d'une collection, utilisez :

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

Si votre base de données supporte les sous requêtes, vous pouvez placer des conditions sur la taille de la sélection dans la clause where de votre requête :

```
from User usr where size(usr.messages) >= 1
```

Si votre base de données ne supporte pas les sous requêtes, utilisez :

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

Cette solution ne peut pas retourner un `User` avec zéro message à cause de la jointure interne, la forme suivante peut donc être utile :

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

Les propriétés d'un JavaBean peuvent être injectées dans les paramètres nommés d'un requête :

```
Query q = s.createQuery("from foo in class Foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean possède getName() and getSize()
List foos = q.list();
```

Les collections sont paginables via l'utilisation de l'interface `Query` avec un filtre :

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Les éléments d'une collection peuvent être triés ou groupés en utilisant un filtre de requête :

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

Vous pouvez récupérer la taille d'une collection sans l'initialiser :

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

Chapitre 12. Requêtes par critères

Intuitive et extensible, l'API d'interrogation par critère est désormais offerte par Hibernate. Pour le moment, cette API est moins puissante que l'HQL et toutes les possibilités qu'il offre. En particulier, criteria ne supporte pas la projection ou l'aggrégation.

12.1. Créer une instance de `Criteria`

L'interface `net.sf.hibernate.Criteria` représente une requête sur une classe persistente donnée. La `Session` fournit les instances de `Criteria`.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

12.2. Restriction du résultat

Un criterion (critère de recherche) est une instance de l'interface `net.sf.hibernate.expression.Criterion`. La classe `net.sf.hibernate.expression.Expression` définit des méthodes pour obtenir des types de `Criterion` pré définis.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.between("weight", minWeight, maxWeight) )
    .list();
```

Les expressions peuvent être goupées de manière logique.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(0) ),
        Expression.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Expression.disjunction()
        .add( Expression.isNull("age") )
            .add( Expression.eq("age", new Integer(0) ) )
            .add( Expression.eq("age", new Integer(1) ) )
            .add( Expression.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Il y a plusieurs types de criterion pré définis (sous classes de `Expression`), mais l'une d'entre elle particulièrement utile vous permet de spécifier directement du SQL.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.sql("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

La zone `{alias}` sera remplacée par l'alias de colonne de l'entité que l'on souhaite intérroger.

12.3. Trier les résultats

Vous pouvez trier les résultats en utilisant `net.sf.hibernate.expression.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

12.4. Associations

Vous pouvez facilement spécifier des contraintes sur des entités liées, par des associations en utilisant `createCriteria()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .createCriteria("kittens")
        .add( Expression.like("name", "F%")
    .list();
```

Notez que la seconde `createCriteria()` retourne une nouvelle instance de `Criteria`, qui se rapporte aux éléments de la collection `kittens`.

La forme alternative suivante est utile dans certains cas.

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Expression.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` ne créé par de nouvelle instance de `Criteria`.)

Notez que les collections kittens contenues dans les instances de `Cat` retournées par les deux précédentes requêtes ne sont *pas* pré-filtrées par les critères ! Si vous souhaitez récupérer uniquement les kittens qui correspondent à la criteria, vous devez utiliser `returnMaps()`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Expression.eq("name", "F%") )
    .returnMaps()
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

12.5. Peuplement d'associations de manière dynamique

Vous pouvez spéficier au runtime le peuplement d'une association en utilisant `setFetchMode()` (c'est-à-dire le chargement de celle-ci). Cela permet de surcharger les valeurs "lazy" et "outer-join" du mapping.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

Cette requête recherchera `mate` et `kittens` via les jointures externes.

12.6. Requête par l'exemple

La classe `net.sf.hibernate.expression.Example` vous permet de construire un critère suivant une instance d'objet donnée.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Les propriétés de type version, identifiant et association sont ignorées. Par défaut, les valeurs null sont exclues.

Vous pouvez ajuster la stratégie d'utilisation de valeurs de l'`Exemple`.

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclure les valeurs zéro
    .excludeProperty("color")  //exclure la propriété nommée "color"
    .ignoreCase()              //ne respecte pas la casse sur les chaines de caractères
    .enableLike();             //utilise like pour les comparaisons de string
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

Vous pouvez utiliser les "exemples" pour des critères sur les objets associés.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

Chapitre 13. Requêtes en sql natif

Vous pouvez aussi écrire vos requêtes dans le dialecte SQL natif de votre base de données. Ceci est utile si vous souhaitez utiliser les fonctionnalités spécifiques de votre base de données comme le mot clé CONNECT d'Oracle. Cette fonctionnalité offre par ailleurs un moyen de migration plus propre et doux d'une application basée sur SQL/JDBC vers une application Hibernate.

13.1. Créer une requête basée sur SQL

Les requêtes de type SQL sont invocables via l'interface `Query` comme les requêtes HQL. La seule différence est l'utilisation de `Session.createSQLQuery()`.

```
Query sqlQuery = sess.createSQLQuery("select {cat.*} from cats {cat}", "cat", Cat.class);
sqlQuery.setMaxResults(50);
List cats = sqlQuery.list();
```

Les trois arguments nécessaires à `createSQLQuery()` sont :

- la chaîne de caractères représentant la requête SQL
- un alias de table
- la classe persistante retournée par la requête

L'alias est utilisé dans la requête pour référencer les propriétés de la classe mappée (`Cat` dans notre exemple). Vous pouvez récupérer plusieurs objets par ligne en passant en argument un tableau de string (d'alias) ainsi que le tableau de `Class` correspondantes.

13.2. Alias et références de propriétés

La notation `{cat.*}` utilisée précédemment signifie "toutes les propriétés". Vous pouvez lister de manière explicite les propriétés, mais vous devez laisser Hibernate gérer les alias SQL des colonnes pour chaque propriété. La forme/nom de ces alias de colonnes est l'alias de leur table postfixé par le nom de la propriété (cat.id). Dans l'exemple suivant, nous récupérons `Cats` depuis une table (`cat_log`) différente de celle déclarée dans nos métadonnées de mapping. Notez que nous pouvons utiliser l'alias de la propriété dans la clause where.

```
String sql = "select cat.originalId as {cat.id}, "
    + "  cat.mateid as {cat.mate}, cat.sex as {cat.sex}, "
    + "  cat.weight*10 as {cat.weight}, cat.name as {cat.name}"
    + "     from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql, "cat", Cat.class)
    .setLong("catId", catId)
    .list();
```

A savoir : si vous listez chaque propriété de manière explicite, vous devez inclure toutes les propriétés de la classe et de ses *sous-classes* !

13.3. Requêtes SQL nommées

Des requêtes SQL nommées peuvent être définies dans le document de mapping et être appellées de la même

manière que les requêtes HQL nommées.

```
List people = sess.getNamedQuery("mySqlQuery")
    .setMaxResults(50)
    .list();
```

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person"/>
    SELECT {person}.NAME AS {person.name},
           {person}.AGE AS {person.age},
           {person}.SEX AS {person.sex}
    FROM PERSON {person} WHERE {person}.NAME LIKE 'Hiber%'
</sql-query>
```

Chapitre 14. Améliorer les performances

14.1. Comprendre les performances des Collections

Nous avons déjà passé du temps à discuter des collections. Dans cette section, nous allons traiter du comportement des collections à l'exécution.

14.1.1. Classification

Hibernate définit trois types de collections :

- les collections de valeurs
- les associations un-vers-plusieurs
- les associations plusieurs-vers-plusieurs

Cette classification distingue les différentes relations entre les tables et les clés étrangères mais ne nous apprend rien de ce que nous devons savoir sur le modèle relationnel. Pour comprendre parfaitement la structure relationnelle et les caractéristiques des performances, nous devons considérer la structure de la clé primaire qui est utilisée par Hibernate pour mettre à jour ou supprimer les éléments des collections. Celà nous amène aux classifications suivantes :

- collections indexées
- sets
- bags

Toutes les collections indexées (maps, lists, arrays) ont une clé primaire constituée des colonnes clé (`<key>`) et `<index>`. Avec ce type de clé primaire, la mise à jour de collection est en général très performante - la clé primaire peut être indexées efficacement et un élément particulier peut être localisé efficacement lorsqu'Hibernate essaie de le mettre à jour ou de le supprimer.

Les Sets ont une clé primaire composée de `<key>` et des colonnes représentant l'élément. Elle est donc moins efficace pour certains types de collections d'éléments, en particulier les éléments composites, les textes volumineux ou les champs binaires ; la base de données peut ne pas être capable d'indexer aussi efficacement une clé primaire aussi complexe. Cependant, pour les associations un-vers-plusieurs ou plusieurs-vers-plusieurs, spécialement lorsque l'on utilise des entités ayant des identifiants techniques, il est probable que cela soit aussi efficace (note : si vous voulez que `SchemaExport` créé effectivement la clé primaire d'un `<set>` pour vous, vous devez déclarer toutes les colonnes avec `not-null="true"`).

Le pire cas intervient pour les Bags. Dans la mesure où un bag permet la duplications des éléments et n'a pas de colonne d'index, aucune clé primaire ne peut être définie. Hibernate n'a aucun moyen de distinguer des enregistrements dupliqués. Hibernate résout ce problème en supprimant complètement les enregistrements (via un simple `DELETE`), puis en recréant la collection chaque fois qu'elle change. Ce qui peut être très inefficace.

Notez que pour une relation un-vers-plusieurs, la "clé primaire" peut ne pas être la clé primaire de la table en base de données - mais même dans ce cas, la classification ci-dessus reste utile (Elle explique comment Hibernate "localise" chaque enregistrement de la collection).

14.1.2. Les lists, les maps et les sets sont les collections les plus efficaces pour la mise à jour

La discussion précédente montre clairement que les collections indexées et (la plupart du temps) les sets, permettent de réaliser le plus efficacement les opérations d'ajout, de suppression ou de modification d'éléments.

Il existe un autre avantage qu'ont les collections indexées sur les Sets dans le cadre d'une association plusieurs vers plusieurs ou d'une collection de valeurs. A cause de la structure inhérente d'un `Set`, Hibernate n'effectue jamais d'`UPDATE` quand un enregistrement est modifié. Les modifications apportées à un `Set` se font via un `INSERT` et `DELETE` (de chaque enregistrement). Une fois de plus, ce cas ne s'applique pas aux associations un vers plusieurs.

Après s'être rappelé que les tableaux ne peuvent pas être chargés tardivement, nous pouvons conclure que les lists, les maps et les sets sont les types de collections les plus performants. (tout en remarquant, que pour certaines valeurs de collections, les sets peuvent être moins performants).

Les sets sont considérés comme le type de collection le plus répendu dans des applications basées sur Hibernate.

Il existe une fonctionnalité non documentée dans cette version d'Hibernate : les mapping `<idbag>` implémentent la sémantique des bags pour une collection de valeurs ou une association plusieurs vers plusieurs et sont plus performants que les autres types de collections dans le cas qui nous occupe !

14.1.3. Les Bags et les lists sont les plus efficaces pour les collections inverse

Avant que vous n'oubliez les bags pour toujours, il y a un cas précis où les bags (et les lists) sont bien plus performants que les sets. Pour une collection marquée comme `inverse="true"` (le choix le plus courant pour un relation un vers plusieurs bidirectionnelle), nous pouvons ajouter des éléments à un bag ou une list sans avoir besoin de l'initialiser (fetch) les éléments du sac! Ceci parce que `Collection.add()` ou `Collection.addAll()` doit toujours retourner vrai pour un bag ou une `List` (contrairement au `Set`). Cela peut rendre le code suivant beaucoup plus rapide.

```
Parent p = (Parent) sess.load(Parent.class, id);
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c);  //pas besoin de charger la collection !
    sess.flush();
```

14.1.4. Suppression en un coup

Parfois, effacer les éléments d'une collection un par un peut être extrêmement inefficace. Hibernate n'est pas totalement stupide, il sait qu'il ne faut pas le faire dans le cas d'une collection complètement vidée (lorsque vous appellez `list.clear()`, par exemple). Dans ce cas, Hibernate fera un simple `DELETE` et le travail est fait !

Supposons que nous ajoutions un élément dans une collection de taille vingt et que nous enlevions ensuite deux éléments. Hibernate effectuera un `INSERT` puis deux `DELETE` (à moins que la collection ne soit un bag). Ce qui est souhaitable.

Cependant, supposons que nous enlevions dix huit éléments, laissant ainsi deux éléments, puis que nous ajoutions trois nouveaux éléments. Il y a deux moyens de procéder.

- effacer dix huit enregistrements un à un puis en insérer trois
- effacer la totalité de la collection (en un `DELETE` SQL) puis insérer les cinq éléments restant un à un

Hibernate n'est pas assez intelligent pour savoir que, dans ce cas, la seconde méthode est plus rapide (Il plutôt heureux qu'Hibernate ne soit pas trop intelligent ; un tel comportement pourrait rendre l'utilisation de triggers de bases de données plutôt aléatoire, etc...).

Heureusement, vous pouvez forcer ce comportement lorsque vous le souhaitez, en liberant (c'est-à-dire en déréférençant) la collection initiale et en retournant une collection nouvellement instanciée avec les éléments restants. Ceci peut être très pratique et très puissant de temps en temps.

Nous avons déjà présenté l'utilisation de l'initialisation tardive pour les collections persistantes dans le chapitre sur le mapping des collections. Une fonctionnalité similaire existe pour les références aux objets ordinaires, elle utilise les proxys CGLIB. Nous avons également mentionné comment Hibernate met en cache les objets persistants au niveau de la `Session`. Des stratégies de cache plus aggressives peuvent être configurées classe par classe.

Dans la section suivante, nous vous montrerons comment utiliser ces fonctionnalités, qui peuvent être utlisées pour atteindre des performantes plus élevées, quand cela est nécessaire.

14.2. Proxy pour une Initialisation Tardive

Hibernate implémente l'initialisation tardive d'objets persistants via la génération de proxy par bytecode enhancement à l'exécution (grâce à l'excellente bibliothèque CGLIB).

Le fichier de mapping déclare une classe ou une interface à utiliser comme interface proxy pour la classe. L'approche recommandée est de définir la classe elle-même :

```
<class name="eg.Order" proxy="eg.Order">
```

Le type des proxys à l'exécution sera une sous classe de `Order`. Notez que les classes soumises à proxy doivent implémenter un contructeur par défaut avec au minimum la visibilité package.

Il y a quelques précautions à prendre lorsque l'on étend cette approche à des classes polymorphiques, exemple :

```
<class name="eg.Cat" proxy="eg.Cat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.DomesticCat">
        .....
    </subclass>
</class>
```

Tout d'abord, les instances de `Cat` ne pourront jamais être "castées" en `DomesticCat`, même si l'instance sous jacente est une instance de `DomesticCat`.

```
Cat cat = (Cat) session.load(Cat.class, id);  // instancie un proxy (n'interroge pas la base de donnée
if ( cat.isDomesticCat() ) {                  // interroge la base de données pour initialiser le prox
    DomesticCat dc = (DomesticCat) cat;       // Erreur !
    ....
}
```

Deuxièmement, il est possible de casser la notion d'== des proxy.

```
Cat cat = (Cat) session.load(Cat.class, id);           // instancie un proxy Cat
```

```
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id);  // un nouveau proxy Cat est requis !
System.out.println(cat==dc);                            // faux
```

Cette situation n'est pas si mauvaise qu'il n'y parait. Même si nous avons deux références à deux objets proxys différents, l'instance de base sera quand même le même objet :

```
cat.setWeight(11.0);  // interroge la base de données pour initialiser le proxy
System.out.println( dc.getWeight() );  // 11.0
```

Troisièmement, vous ne pourrez pas utiliser un proxy CGLIB pour une classe `final` ou pour une classe contenant la moindre méthode `final`.

Enfin, si votre objet persistant obtient une ressource à l'instanciation (par example dans les initialiseurs ou dans le contructeur par défaut), alors ces ressources seront aussi obtenues par le proxy. La classe proxy est vraiment une sous classe de la classe persistante.

Ces problèmes sont tous dus aux limitations fondamentales du modèle d'héritage unique de Java. Si vous souhaitez éviter ces problèmes, vos classes persistantes doivent chacune implémenter une interface qui déclare ses méthodes métier. Vous devriez alors spécifier ces interfaces dans le fichier de mapping :

```
<class name="eg.Cat" proxy="eg.ICat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.IDomesticCat">
        .....
    </subclass>
</class>
```

où `Cat` implémente l'interface `ICat` et `DomesticCat` implémente l'interface `IDomesticCat`. Ainsi, des proxys pour les instances de `Cat` et `DomesticCat` pourraient être retournées par `load()` ou `iterate()` (Notez que `find()` ne retourne pas de proxy).

```
ICat cat = (ICat) session.load(Cat.class, catid);
Iterator iter = session.iterate("from cat in class eg.Cat where cat.name='fritz'");
ICat fritz = (ICat) iter.next();
```

Les relations sont aussi initialisées tardivement. Ceci signifie que vous devez déclarer chaque propriété comme étant de type `ICat`, et non `Cat`.

Certaines opérations ne nécessitent pas l'initialisation du proxy

- `equals()`, si la classe persistante ne surcharge pas `equals()`
- `hashCode()`, si la classe persistante ne surcharge pas `hashCode()`
- Le getter de l'identifiant

Hibernate détectera les classes qui surchargent `equals()` ou `hashCode()`.

Les exceptions qui surviennent à l'initialisation d'un proxy sont encapsulées dans une `LazyInitializationException`.

Parfois, nous devons nous assurer qu'un proxy ou une collection est initialisée avant de fermer la `Session`. Bien sûr, nous pouvons toujours forcer l'initialisation en appelant par exemple `cat.getSex()` ou `cat.getKittens().size()`. Mais ceci n'est pas très lisible pour les personnes parcourant le code et n'est pas très générique. Les méthodes statiques `Hibernate.initialize()` et `Hibernate.isInitialized()` fournissent à l'application un moyen de travailler avec des proxys ou des collections initialisés. `Hibernate.initialize(cat)` forcera l'initialisation d'un proxy de `cat`, si tant est que sa `Session` est ouverte. `Hibernate.initialize( cat.getKittens() )` a le même effet sur la collection kittens.

14.3. Utiliser le batch fetching (chargement par batch)

Pour améliorer les performances, Hibernate peut utiliser le batch fetching ce qui veut dire qu'Hibernate peut charger plusieurs proxys non initialisés en une seule requête lorsque l'on accède à l'un de ces proxys. Le batch fetching est une optimisation intimement liée à la stratégie de chargement tardif. Il y a deux moyens d'activer le batch fetching : au niveau de la classe et au niveau de la collection.

Le batch fetching pour les classes/entités est plus simple à comprendre. Imaginez que vous ayez la situation suivante à l'exécution : vous avez 25 instances de `Cat` chargées dans une `Session`, chaque `Cat` a une référence à son `owner`, une `Person`. La classe `Person` est mappée avec un proxy, `lazy="true"`. Si vous itérez sur tous les cats et appelez `getOwner()` sur chacun d'eux, Hibernate exécutera par défaut 25 `SELECT`, pour charger les owners (initialiser le proxy). Vous pouvez paramétrer ce comportement en spécifiant une `batch-size` (taille de batch) dans le mapping de `Person` :

```
<class name="Person" lazy="true" batch-size="10">...</class>
```

Hibernate exécutera désormais trois requêtes, en chargeant respectivement 10, 10, et 5 entités. Vous pouvez voir que le batch fetching est une optimisation aveugle dans le mesure où elle dépend du nombre de proxys non initialisés dans une `Session` particulière.

Vous pouvez aussi activer le batch fetching pour les collections. Par exemple, si chaque `Person` a une collection chargée tardivement de `Cats`, et que 10 persons sont actuellement chargées dans la `Session`, itérer sur toutes les persons générera 10 `SELECT`s, un pour chaque appel de `getCats()`. Si vous activez le batch fetching pour la collection `cats` dans le mapping de `Person`, Hibernate pourra précharger les collections :

```
<class name="Person">
    <set name="cats" lazy="true" batch-size="3">
        ...
    </set>
</class>
```

Avec une taille de batch (`batch-size`) de 3, Hibernate chargera respectivement 3, 3, 3, et 1 collections en 4 `SELECT`s. Encore une fois, la valeur de l'attribut dépend du nombre de collections non initialisées dans une `Session` particulière.

Le batch fetching de collections est particulièrement utile si vous avez des arborescenses récursives d'éléments (typiquement, le schéma facture de matériels).

14.4. Le cache de second niveau

Une `Session` Hibernate est un cache de niveau transactionnel des données persistantes. Il est possible de configurer un cache de cluster ou de JVM (de niveau `SessionFactory` pour être exact) défini classe par classe et collection par collection. Vous pouvez même utiliser votr choix de cache en implémentant le pourvoyeur (provider) associé. Faites attention, les caches ne sont jamais avertis des modifications faites dans la base de données par d'autres applications (ils peuvent cependant être configurés pour régulièrement expirer les données en cache).

Par défaut, Hibernate utilise EHCache comme cache de niveau JVM (le support de JCS est désormais déprécié et sera enlevé des futures versions d'Hibernate). Vous pouvez choisir une autre implémentation en spécifiant le nom de la classe qui implémente `net.sf.hibernate.cache.CacheProvider` en utilisant la propriété `hibernate.cache.provider_class`.

Tableau 14.1. Fournisseur de cache

Cache	Classe pourvoyeuse	Type	Support en Cluster	Cache de requêtes supporté
Hashtable (ne pas utiliser en production)	`net.sf.hibernate.cache.HashtableCacheProvider`	mémoire		oui
EHCache	`net.sf.hibernate.cache.EhCacheProvider`	mémoire, disque		oui
OSCache	`net.sf.hibernate.cache.OSCacheProvider`	mémoire, disque		oui
SwarmCache	`net.sf.hibernate.cache.SwarmCacheProvider`	en cluster (multicast ip)	oui (invalidation de cluster)	
JBoss TreeCache	`net.sf.hibernate.cache.TreeCacheProvider`	en cluster (multicast ip), transactionnel	oui (replication)	oui (horloge sync. nécessaire)

14.4.1. Mapping de Cache

L'élément `<cache>` d'une classe ou d'une collection à la forme suivante :

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
/>
```

(1) `usage` spécifie la stratégie de cache : `transactionel`, `lecture-écriture`, `lecture-écriture non stricte` ou `lecture seule`

Alternativement (voir préférentiellement), vous pouvez spécifier les éléments `<class-cache>` et `<collection-cache>` dans `hibernate.cfg.xml`.

L'attribut `usage` spécifie une *stratégie de concurrence d'accès au cache*.

14.4.2. Strategie : lecture seule

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.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

14.4.3. Stratégie : lecture/écriture

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 transation est terminée à l'appel de `Session.close()` ou `Session.disconnect()`. Si vous souhaitez utiliser cette stratégie dans un cluster, vous devriez vous assurer que l'implémentation de cache utilisée supporte le vérrouillage. Ce que ne font *pas* les pourvoyeurs caches fournis.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

14.4.4. Stratégie : lecture/écriture non stricte

Si l'application besoin de mettre à jour les données de manière occasionnelle (qu'il est très peu probable que deux transactions essaient de mettre à jour le même élément simultanément) et qu'une isolation transactionnelle stricte n'est 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 transation est terminée à l'appel de `Session.close()` ou `Session.disconnect()`

14.4.5. Stratégie : transactionelle

La stratégie de cache `transactional` supporte un cache complètement transactionnel comme, 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`.

Aucun des caches livrés ne supporte toutes les stratégies de concurrence. Le tableau suivant montre quels caches sont compatibles avec quelles stratégies de concurrence.

Tableau 14.2. Stratégie de concurrence du cache

Cache	read-only (lecture seule)	nonstrict-read-write (lecture-écriture non stricte)	read-write (lecture-ériture)	transactional (transactionnel)
Hashtable (ne pas utilser en production)	oui	oui	oui	
EHCache	oui	oui	oui	
OSCache	oui	oui	oui	
SwarmCache	oui	oui		
JBoss TreeCache	oui			oui

14.5. Gérer le cache de la `Session`

A chaque fois que vous passez un objet à `save()`, `update()` ou `saveOrUpdate()` ou chaque fois que récupérez un objet via `load()`, `find()`, `iterate()`, ou `filter()`, cet objet est ajouté au cache interne de la `Session`. Quand `flush()` est appelé, l'état de cet objet est synchronisé avec la base de données. Si vous ne souhaitez pas que cette synchronisation se fasse ou si vous êtes en train de travailler avec un grand nombre d'objets et avez besoin de gérer la mémoire de manière efficace, la méthode `evict()` peut être utilisée pour enlever l'objet et ses collections du cache.

```
Iterator cats = sess.iterate("from eg.Cat as cat"); //un grand result set
while ( cats.hasNext() ) {
    Cat cat = (Cat) iter.next();
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

Hibernate enlèvera automatiquement toutes les entités associées si l'association est mappée avec `cascade="all"` ou `cascade="all-delete-orphan"`.

La `Session` dispose aussi de la méthode `contains()` pour déterminer si une instance appartient au cache de la session.

Pour retirer tous les objets du cache session, appelez `Session.clear()`

Pour le cache de second niveau, il existe des méthodes définies dans `SessionFactory` pour retirer des instances du cache, la classe entière, une instance de collection ou le rôle entier d'une collection.

14.6. Le cache de requêtes

Les résultats d'une requête peuvent aussi être placés en cache. Ceci n'est utile que pour les requêtes qui sont exécutées avec les mêmes paramètres. Pour utiliser le cache de requêtes, vous devez d'abord l'activer en mettant `hibernate.cache.use_query_cache=true`. Ceci active la création de deux régions de cache, une contenant les résultats des requêtes en cache (`net.sf.hibernate.cache.QueryCache`), l'autre contenant les modifications les plus récentes des tables interrogées (`net.sf.hibernate.cache.UpdateTimestampsCache`). Notez que le cache de requête ne met pas en cache l'état de chaque entité du résultat, il met seuleument en cache les valeurs des identifiants et les résultats de type valeur. Le cache requête est donc généralement utilisé en association avec le cache de second niveau.

La plupart des requêtes ne retirent pas de bénéfice pas du cache, donc par défaut les requêtes ne sont pas mises en cache. Pour activer le cache, appelez `Query.setCacheable(true)`. Cet appel permet de vérifier si les résultats sont en cache ou non, voire d'ajouter ces résultats si la requête est exécutée.

Si vous avez besoin de contrôler finement les délais d'expiration du cache, vous pouvez spécifier une région de cache nommée pour une requête particulière en appelant `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

Si une requête doit forcer le rafraîchissement de sa région de cache, vous pouvez forcer `Query.setForceCacheRefresh()` à `true`. C'est particulièrement utile dans les cas ou la base de données peut

être mise à jour par un autre processus (autre qu'Hibernate) et permet à l'application de rafraichir de manière sélective les régions de cache de requête en fonction de sa connaissance des évènements. C'est une alternative à l'éviction d'une région de cache de requête. Si vous avez besoin d'un contrôle fin du rafraîchissement pour plusieurs requêtes, utlisez cette fonction plutôt qu'une nouvelle région pour chaque requête.

Chapitre 15. Guide des outils

Des outils en ligne de commande permettre de gérer de cycles de développement complet de projets utilisant Hibernate. Ces outils font partie du projet Hibernate lui-même. Ils peuvent être utilisés conjointement avec d'autres outils qui supportent nativement Hibernate : XDoclet, Middlegen and AndroMDA.

La distribution principale d'Hibernate est livrée avec l'outil le plus important (il peut même être utilisé "à l'intérieur" d'Hibernate, à la volée) :

- Génération du schéma DDL depuis un fichier de mapping (aussi appelé `SchemaExport`, `hbm2ddl`)

D'autres outils directement fournis par le projet Hibernate sont distribués dans un package séparé, *Hibernate Extensions*. Ce package inclus des outils pour les tâches suivantes :

- Génération de source Java à partir d'un fichier de mapping (`CodeGenerator`, `hbm2java`)
- Génération de fichiers de mapping à partir des classes java compilées ou à partir des sources Java marquées avec XDoclet (`MapGenerator`, `class2hbm`)

Il existe un autre outil distribué avec les extensions Hibernate : `ddl2hbm`. Il est considéré comme obsolète et ne sera plus maintenu, Middlegen faisant un meilleur travail pour cette tâche.

D'autres outils tiers fournissent un support Hibernate :

- Middlegen (génération du fichier de mapping à partir d'un schéma de base de données existant)
- AndroMDA (génération du code des classes persistantes à partir de diagrammes UML et de leur représentation XML/XMI en utilisant une stratégie MDA - Model-Driven Architecture)

Ces outils tiers ne sont pas documentés dans ce guide. Référez-vous au site Hibernate pour des informations à jour (une photo du site est inclus dans la distribution principale d'Hibernate).

15.1. Génération de Schéma

Le DDL peut être généré à partir de vos fichiers de mapping par une ligne de commande. Un fichier .bat est localisé dans le répertoire `hibernate-x.x.x/bin` de la distribution principale.

Le schéma généré inclut les contraintes d'intégrité du référentiel (clés primaires et étrangères) pour les tables d'entités et de collections. Les tables et les séquences sont aussi créées pour les générateurs d'identifiants mappés.

Vous *devez* spécifier un `Dialecte` SQL via la propriété `hibernate.dialect` lorsque vous utilisez cet outil.

15.1.1. Personnaliser le schéma

Plusieurs éléments du mapping hibernate définissent un attribut optionnel nommé `length`. Vous pouvez paramétrer la longueur d'une colonne avec cet attribut (ou, pour les types de données numeric/decimal, la précision).

Certains éléments acceptent aussi un attribut `not-null` (utilisé pour générer les contraintes de colonnes `NOT`

NULL) et un attribut unique (pour générer une contrainte de colonne UNIQUE).

Quelques éléments acceptent un attribut index pour spécifier le nom d'un index pour cette colonne. Un attribut unique-key peut être utilisé pour grouper des colonnes dans une seule contrainte de clé. Actuellement, la valeur spécifiée pour l'attribut unique-key n'est *pas* utilisée pour nommer la contrainte, mais uniquement pour grouper les colonnes dans le fichier de mapping.

Exemples :

```
<property name="foo" type="string" length="64" not-null="true"/>

<many-to-one name="bar" foreign-key="fk_foo_bar" not-null="true"/>

<element column="serial_number" type="long" not-null="true" unique="true"/>
```

Sinon, ces éléments acceptent aussi un éléments fils <column>. Ceci est particulièrement utile pour des types multi-colonnes :

```
<property name="foo" type="string">
    <column name="foo" length="64" not-null="true" sql-type="text"/>
</property>

<property name="bar" type="my.customtypes.MultiColumnType"/>
    <column name="fee" not-null="true" index="bar_idx"/>
    <column name="fi" not-null="true" index="bar_idx"/>
    <column name="fo" not-null="true" index="bar_idx"/>
</property>
```

L'attribut sql-type permet à l'utilisateur de surcharger le mapping par défaut d'un type Hibernate vers un type de données SQL.

L'attribut check permet de spécifier une contrainte de vérification.

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>

<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

Tableau 15.1. Résumé

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	index_name	spécifie le nom d'un index (multi-colonnes)
unique-key	unique_key_name	spécifie le nom d'une contrainte d'unicité multi-colonnes
foreign-key	foreign_key_name	spécifie le 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 extrêmités inverse="true" se seront pas prises en compte par SchemaExport.

Attribut	Valeur	Interprétation
sql-type	column_type	surcharge le type par défaut (attribut de l'élément <column> uniquement)
check	SQL expression	créé une contrainte de vérification sur la table ou la colonne

15.1.2. Exécuter l'outil

L'outil SchemaExport génère un script DDL vers la sortie standard et/ou exécute les ordres DDL.

java -cp *classpath_hibernate* net.sf.hibernate.tool.hbm2ddl.SchemaExport *options fichiers_de_mapping*

Tableau 15.2. SchemaExport Options de la ligne de commande

Option	Description
--quiet	ne pas écrire le script vers la sortie standard
--drop	supprime seuleument 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	formatte proprement le SQL généré dans le script
--delimiter=x	paramètre un délimiteur de fin de ligne pour le script

Vous pouvez même intégrer SchemaExport dans votre application :

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

15.1.3. Propriétés

Les propriétés de la base de données peuvent être spécifiées

- comme propriétés système avec -D<*property*>
- dans hibernate.properties
- dans un fichier de propriétés déclaré avec --properties

Les propriétés nécessaires sont :

Tableau 15.3. Propriétés de connexion nécessaires à SchemaExport

Nom de la propriété	Description
hibernate.connection.driver_class	classe du driver JDBC
hibernate.connection.url	URL JDBC

Nom de la propriété	Description
hibernate.connection.username	utilisateur de la base de données
hibernate.connection.password	mot de passe de l'utilisateur
hibernate.dialect	dialecte

15.1.4. Utiliser Ant

Vous pouvez appeler SchemaExport depuis votre script de construction Ant :

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

Si vous ne spécifiez ni properties, ni fichier dans config, la tâche SchemaExportTask tentera d'utiliser les propriétés du projet Ant. Autrement dit, if vous ne voulez pas ou n'avez pas besoin d'un fichier externe de propriétés ou de configuration, vous pouvez mettre les propriétés de configuration hibernate.* dans votre build.xml ou votre build.properties.

15.1.5. Mises à jour incrémentales du schéma

L'outil SchemaUpdate mettra à jour un schéma existant en effectuant les changement par "incrément". Notez que SchemaUpdate dépends beaucoup de l'API JDBC metadata, il ne fonctionnera donc pas avec tous les drivers JDBC.

java -cp *classpath_hibernate* net.sf.hibernate.tool.hbm2ddl.SchemaUpdate *options fichiers_de_mapping*

Tableau 15.4. SchemaUpdate Options de ligne de commande

Option	Description
--quiet	ne pas écrire vers la sortie standard
--properties=hibernate.properties	lire les propriétés de la base de données à partir d'un fichier

Vous pouvez intégrer SchemaUpdate dans votre application :

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

15.1.6. Utiliser Ant pour des mises à jour de schéma par incrément

Vous pouvez appeler `SchemaUpdate` depuis le script Ant :

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

15.2. Génération de code

Le générateur de code Hibernate peut être utilisé pour générer les squelettes d'implémentation des classes depuis un fichier de mapping. Cet outil est inclus dans la distribution des extensions Hibernate (téléchargement séparé).

`hbm2java` analyse les fichiers de mapping et génère les sources Java complètes. Ainsi, en fournissant les fichiers `.hbm`, on n'a plus à écire à la main les fichiers Java.

`java -cp` *classpath_hibernate* `net.sf.hibernate.tool.hbm2java.CodeGenerator` *options fichiers_de_mapping*

Tableau 15.5. Options de ligne de commande pour le générateur de code

Option	Description
`--output=`*repertoire_de_sortie*	répertoire racine pour le code généré
`--config=`*fichier_de_configuration*	fichier optionnel de configuration

15.2.1. Le fichier de configuration (optionnel)

Le fichier de configuration fournit un moyen de spécifier de multiples "renderers" de code source et de déclarer des `<meta>` attributs qui seront globaux. Voir la section sur l'attribut `<meta>`.

```
<codegen>
    <meta attribute="implements">codegen.test.IAuditable</meta>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate
        package="autofinders.only"
        suffix="Finder"
        renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

Ce fichier de configuration déclare un méta attribut global "implements" et spécifie deux "renderers", celui par défaut (BasicRenderer) et un renderer qui génère des "finder" (requêteurs) (voir "Génération basique de finder" ci-dessous).

Le second renderer est paramétré avec un attribut package et suffixe.

L'attribut package spécifie que les fichiers sources générés depuis ce renderer doivent être placés dans ce package au lieu de celui spécifié dans les fichiers `.hbm`.

L'attribut suffixe spécifie le suffixe pour les fichiers générés. Ex: ici un fichier nommé `Foo.java` génèrera un fichier `FooFinder.java`.

Il est aussi possible d'envoyer des paramètres arbitraires vers les renderers en ajoutant les attributs `<param>` dans les éléments `<generate>`.

hbm2java supporte actuellement un tel paramètre appellé, `generate-concrete-empty-classes` qui informe le BasicRenderer de ne générer que les classes concrêtes qui étendent une classe de base pour toutes vos classes. Le fichier config.xml suivant illustre cette fonctionnalité.

```
            <codegen>
              <generate prefix="Base" renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
              <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer">
                <param name="generate-concrete-empty-classes">true</param>
                <param name="baseclass-prefix">Base</param>
              </generate>
            </codegen>
```

Notez que ce config.xml configure deux renderers. Un qui génère les classes Base, et un second qui génère les classes concrêtes creuses.

15.2.2. L'attribut `meta`

L'attribut `<meta>` est un moyen simple d'annoter les fichiers `hbm.xml` avec des informations utiles aux outils. Ces informations, bien que non nécessaires au noyau d'Hibernate, se trouvent dont à un endroit naturel.

Vous pouvez utiliser l'élément `<meta>` pour dire à `hbm2java` de générer des setters "protected", d'implémenter toujours un certain nombre d'interfaces ou même d'étendre une classe de base particulière...

L'exemple suivant :

```
<class name="Person">
    <meta attribute="class-description">
        Javadoc de la classe Person
        @author Frodon
    </meta>
    <meta attribute="implements">IAuditable</meta>
    <id name="id" type="long">
        <meta attribute="scope-set">protected</meta>
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <meta attribute="field-description">Le nom de la personne</meta>
    </property>
</class>
```

produira le code suivant (le code a été raccourci pour une meilleure compréhension). Notez la javadoc et les setters protected :

```
// package par défaut

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
```

```
import org.apache.commons.lang.builder.ToStringBuilder;

/**
 *         Javadoc de la classe Person
 *         @author Frodon
 *
 */
public class Person implements Serializable, IAuditable {

    /** identifier field */
    public Long id;

    /** nullable persistent field */
    public String name;

    /** full constructor */
    public Person(java.lang.String name) {
        this.name = name;
    }

    /** default constructor */
    public Person() {
    }

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
        this.id = id;
    }

    /**
     * Le nom de la personne
     */
    public java.lang.String getName() {
        return this.name;
    }

    public void setName(java.lang.String name) {
        this.name = name;
    }

}
```

Tableau 15.6. Attributs meta supportés

Attribut	Description
`class-description`	inséré dans les javadoc des classes
`field-description`	inséré dans les javadoc des champs/propriétés
`interface`	Si true une interface est générée au lieu d'une classe
`implements`	interface que la classe doit implémenter
`extends`	classe que la classe doit étendre (ignoré pour les classes filles)
`generated-class`	surcharge le nom de la classe générée
`scope-class`	visibilité de la classe
`scope-set`	visibilité des méthodes setter
`scope-get`	visibilité des méthodes getter

Attribut	Description
scope-field	visibilité du champs
use-in-tostring	inclus cette propriété dans toString()
implement-equals	inclus equals() et hashCode() dans cette classe.
use-in-equals	inclus cette propriété dans equals() et hashCode().
bound	ajoute le support de propertyChangeListener pour une propriété
constrained	support de bound + vetoChangeListener pour une propriété
gen-property	la propriété ne sera pas générée si false (à utiliser avec précaution)
property-type	Surcharge le type par défaut de la propriété. Utilisez le pour spécifier un type concret au lieu de Object
class-code	Code supplémentaire qui sera inséré en fin de classe
extra-import	Import supplémentaire qui sera inséré à la fin de tous les imports
finder-method	voir "Générateur de requêteurs basiques"
session-method	voir "Générateur de requêteurs basiques"

Les attributs déclarés via l'élément <meta> sont par défaut "hérités" dans les fichiers hbm.xml.

Ce qui veut dire ? Ceci veut dire que si, par exemple, vous voulez que toutes vos classes implémentent IAuditable, vous n'avez qu'à ajouter <meta attribute="implements">IAuditable</meta> au début du fichier hbm.xml, après <hibernate-mapping>. Toutes les classes définies dans les fichiers hbm.xml implémenteront IAuditable ! (A l'exception des classes qui ont meta attribut "implements", car les méta tags spécifiés localement surchargent/remplacent toujours les meta tags hérités).

Note : Ceci s'applique à tous les <meta> attributs. Ceci peut aussi être utilisé, par exemple, pour spécifier que tous les champs doivent être déclarés protected, au lieu de private par défaut. Pour cela, on ajoute <meta attribute="scope-field">protected</meta> juste après l'attribut <class> et tous les champs de cette classe seront protected.

Pour éviter d'hériter un <meta> attribut, vous pouvez spécifier inherit="false" pour l'attribut, par exemple <meta attribute="scope-class" inherit="false">public abstract</meta> restreindra la visibilité de classe à la classe courante, pas les classes filles.

15.2.3. Générateur de Requêteur Basique (Basic Finder)

Il est désormais possible de laisser hbm2java génèrer des requêteur (finders) basiques pour les propriétés mappées par Hibernate. Ceci nécessite deux choses dans les fichiers hbm.xml.

La première est une indication des champs pour lesquels les finders doivent être générés. Vous indiquez cela à l'aide d'un élément meta au sein de l'élément property :

```
<property name="name" column="name" type="string">
     <meta attribute="finder-method">findByName</meta>
</property>
```

Le texte inclus dans l'élément donnera le nom de la méthode finder.

La seconde est de créer un fichier de configuration pour hbm2java en y ajoutant le renderer adéquat :

```
<codegen>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate suffix="Finder" renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

Et utiliser ensuite le paramètre `hbm2java --config=xxx.xml` où `xxx.xml` est le fichier de configuration que vous venez de créer.

Un paramètre optionnel est un meta attribut au niveau de la classe de la forme :

```
<meta attribute="session-method">
    com.whatever.SessionTable.getSessionTable().getSession();
</meta>
```

Qui représente le moyen d'obtenir des sessions si vous utilisez le pattern *Thread Local Session* (documenté dans la zone Design Patterns du site web Hibernate).

15.2.4. Renderer/Générateur basés sur Velocity

Il est désormais possible d'utiliser velocity comme mécanisme de rendering alternatif. Le fichier config.xml suivant montre comment configurer hbm2java pour utiliser ce renderer velocity.

```
    <codegen>
     <generate renderer="net.sf.hibernate.tool.hbm2java.VelocityRenderer">
      <param name="template">pojo.vm</param>
     </generate>
    </codegen>
```

Le paramètre nommé `template` est un chemin de ressource vers le fichier de macro velocity que vous souhaitez utiliser. Ce fichier doit être disponible dans le classpath utilisé par hbm2java. N'oubliez donc pas d'ajouter le répertoire où se trouve pojo.vm à votre tâche ant ou script shell (le répertoire par défaut est `./tools/src/velocity`).

Soyez conscients que le `pojo.vm` actuel ne génère que les parties basiques des java beans. Il n'est pas aussi complet et riche que le renderer par défaut - il lui manque notamment le support de beaucoup de `meta` tags.

15.3. Génération des fichier de mapping

Un squelette de fichier de mapping peut être généré depuis les classes persistantes compilées en utilisant l'outil en ligne de commande appelé `MapGenerator`. Cet outil fait partie de la distribution des extensions Hibernate.

Le générateur de fichier de mapping fournit un mécanisme qui produit les mappings à partir des classes compilées. Il utilise la réflexion java pour trouver les *propriétés* et l'heuristique pour trouver un mapping approprié pour le type de la propriété. Le fichier généré n'est qu'un point de départ. Il n'y a aucun moyen de produire un mapping Hibernate complet sans informations supplémentaires de l'utlisateur. Cependant, l'outil génère plusieurs des parties répétitives à écrire dans les fichiers de mapping.

Les classes sont ajoutées une par une. L'outil n'acceptera que les classes qu'il considère comme *persistable par Hibernate*.

Pour être *persistable par Hibernate* une classe

- ne doit pas être de type primitif
- ne doit pas être un tableau
- ne doit pas être une interface
- ne doit pas être une classe imbriquée
- doit avoir un constructeur par défaut (sans argument)

Notez que les interfaces et classes imbriquées sont persistables par Hibernate, mais l'utilisateur ne le désirera généralement pas.

`MapGenerator` remontera la hiérarchie de classe pour essayer d'ajouter autant de superclasses (persistables par Hibernate) que possible à la même table dans la base de données. La "recherche" stoppe dès qu'une propriété, ayant son nom figurant dans la liste des *noms d'UID candidats* est trouvée.

La liste par défaut des noms de propriété candidats pour UID est: `uid`, `UID`, `id`, `ID`, `key`, `KEY`, `pk`, `PK`.

Les propriétés sont trouvées quand la classe possède un getter et un setter associé, quand le type du setter ayant un argument unique est le même que le type retourné par le getter sans argument, et que le setter retourne `void`. De plus le nom du setter doit commencer par `set`, le nom du getter par `get` (ou `is` si le type de la propriété est boolean). Le reste du nommage doit alors correspondre au nom de la propriété (à l'exception de l'initiale qui passe de minuscule à majuscule).

Les règles de détermination du type de base de données de chaque propriété sont :

1. Si le type java est `Hibernate.basic()`, alors la propriété est une simple colonne de ce type.
2. Pour les types utilisateurs `hibernate.type.Type` et `PersistentEnum` une simple colonne est aussi utilisée.
3. Si le type est un tableau, alors un tableau Hibernate est utilisé, et `MapGenerator` essaie d'utiliser la réflexion sur un élément du tableau.
4. Si le type est `java.util.List`, `java.util.Map`, ou `java.util.Set`, alors les types Hibernate correspondant sont utilisés, `MapGenerator` ne peut aller plus loin dans la découverte de ces types.
5. Si le type est une autre classe, `MapGenerator` repousse la décision de sa représentation en base de données quand toutes les classes auront été traitées. A ce moment, si la classe a été trouvée via la recherche des superclasses décrites plus haut, la propriété est une association `plusieurs-vers-un`. Si la classe a des propriétés, alors c'est un `composant`. Dans les autres cas elle est sérialisable, ou non persistable.

15.3.1. Exécuter l'outil

L'outil écrit des mappings XML vers la sortie standard et/ou un fichier.

Quand vous invoquez l'outil, vos classes compilées doivent être dans le classpath.

`java -cp` *classpath_contenant_hibernate_et_vos_classes* `net.sf.hibernate.tool.class2hbm.MapGenerator` *options et noms_des_classes*

Il y a deux modes opératoires : par ligne de commande ou interactif.

Le mode intéractif est lancé en plaçant l'argument `--interact` dans la ligne de commande. Ce mode fournit un prompt de réponse. En l'utilisant vous pouvez paramétrer le nom de la propriété UID pour chacune des classes via la commande `uid=XXX` où `XXX` est le nom de la propriété UID. Les autres commande sont simplement le nom de la classe entièrement qualifiée, ou la commande done qui émet l'XML et termine.

Dans le mode ligne de commande les arguments sont les options ci-dessous espacées du nom qualifié de la classe à traiter. La plupart des options sont faites pour être utilisées plusieurs fois, chacune affectant les classes

ajoutées.

Tableau 15.7. Options de la ligne de commande MapGenerator

Option	Description
`--quiet`	ne pas afficher le mapping O-R vers la sortie standard
`--setUID=uid`	paramètre la liste des UIDs candidats sur le singleton uid
`--addUID=uid`	ajouter uid au début de la liste des UIDs candidats
`--select=`*mode*	sélectionne le mode utilisé. *mode*(e.g., *distinct* or *all*) pour les classes ajoutées par la suite
`--depth=<small-int>`	limite le nombre de récursions utilisées pour les trouver les composants pour les classes ajoutées par la suite
`--output=mon_mapping.xml`	envoie le mapping O-R vers un fichier
nom.de.classe.Qualifie	ajoute la classe au mapping
`--abstract=`*nom.de.classe.Qualifie*	voir ci dessous

Le paramètre abstract configure l'outil map generator afin qu'il ignore les super classes spécifiques et donc pour que les classes hérités ne soient pas mappées dans une grande table Par exemple, regardons ces hiérarchies de classe :

```
Animal-->Mamiphère-->Humain

Animal-->Mamiphère-->Marsupial-->Kangourou
```

Si le paramètre `--abstract` n'est *pas* utilisé, toutes les classes seront mappées comme classes filles de `Animal`, ce qui donnera une grande table contenant toutes les propriétés de toutes les classes plus une colonne discriminante indiquant laquelle de classes fille est réellement stockée dans cet enregistrement. Si `mamiphère` est marquée comme `abstract`, `Humain` et `Marsupial` seront mappés à des déclarations de `<class>` séparées et stockées dans des tables différentes. `Kangaroo` sera une classe fille de `Marsupial` à moins que `Marsupial` soit aussi marquée comme `abstract`.

Chapitre 16. Exemple : Père/Fils

L'une des premières choses que les nouveaux utilisateurs essaient de faire avec Hibernate est de modéliser une relation père/fils. Il y a deux approches différentes pour cela. Pour un certain nombre de raisons, la méthode la plus courante, en particulier pour les nouveaux utilisateurs, est de modéliser les deux relations `Père` et `Fils` comme des classes entités liées par une association `<one-to-many>` du `Père` vers le `Fils` (l'autre approche est de déclarer le `Fils` comme un `<composite-element>`). Il est évident que le sens de l'association un vers plusieurs (dans Hibernate) est bien moins proche du sens habituel d'une relation père/fils que ne l'est celui d'un élément cmposite. Nous allons vous expliquer comment utiliser une association *un vers plusieurs bidirectionnelle avec cascade* afin de modéliser efficacement et élégamment une relation père/fils, ce n'est vraiment pas difficile !

16.1. Une note à propos des collections

Les collections Hibernate sont considérées comme étant une partie logique de l'entité dans laquelle elle sont contenues ; jamais des entités qu'elle contiennent. C'est une distinction crutiale ! Les conséquences sont les suivantes :

- Quand nous ajoutons / retirons un objet d'une collection, le numéro de version du propriétaire de la collection est incrémenté.
- Si un objet qui a été enlevé d'une collection est une instance de type valeur (ex : élément composite), cet objet cessera d'être persistant et son état sera complètement effacé de la base de données. Par ailleurs, ajouter une instance de type valeur dans une collection aura pour conséquence que son état persistera immédiatement.
- Si une entité est enlevée d'une collection (association un-vers-plusieurs ou plusieurs-vers-plusieurs), par défaut, elle ne sera pas effacée. Ce comportement est complètement logique - une modification de l'un des états internes d'une entité ne doit pas causer la disparition de l'entité associée ! De même, l'ajout d'une entité dans une collection n'engendre pas, par défaut, la persistence de cette entité.

Le comportement par défaut est donc que l'ajout d'une entité dans une collection créé simplement le lien entre les deux entités, et qu'effacer une entité supprime ce lien. C'est le comportement le plus approprié dans la plupart des cas. Ce comportement n'est cependant pas approprié lorsque la vie du fils est liée au cycle de vie du père.

16.2. un-vers-plusieurs bidirectionnel

Supposons que nous ayons une simple association `<one-to-many>` de `Parent` vers `Child`.

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Si nous executions le code suivant

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate exécuterait deux ordres SQL:

- un `INSERT` pour créer l'enregistrement pour `c`
- un `UPDATE` pour créer le lien de `p` vers `c`

Ceci est non seuleument inefficace, mais viole aussi toute contrainte `NOT NULL` sur la colonne `parent_id`.

La cause sous jacente est que le lien (la clé étrangère `parent_id`) de `p` vers `c` n'est pas considérée comme faisant partie de l'état de l'objet `Child` et n'est donc pas créé par l'`INSERT`. La solution est donc que ce lien fasse partie du mapping de `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(Nous avons aussi besoin d'ajouter la propriété `parent` dans la classe `Child`).

Maintenant que l'état du lien est géré par l'entité `Child`, nous spécifions à la collection de ne pas mettre à jour le lien. Nous utilisons l'attribut `inverse`.

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Le code suivant serait utilisé pour ajouter un nouveau `Child`

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

Maintenant, seul un `INSERT` SQL est nécessaire !

Pour alléger encore un peu les choses, nous devrions créer une méthode `addChild()` dans `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Le code d'ajout d'un `Child` serait alors

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

16.3. Cycle de vie en cascade

L'appel explicite de `save()` est un peu fastidieux. Nous pouvons simplifier cela en utilisant les cascades.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
```

```
</set>
```

Simplifie le code précédent en

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

De la même manière, nous n'avons pas à itérer sur les fils lorsque nous sauvons ou effacons un `Parent`. Le code suivant efface `p` et tous ces fils de la base de données.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

Par contre, ce code

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

n'effacera pas `c` de la base de données, il enlèvera seulement le lien vers `p` (et causera une violation de contrainte `NOT NULL`, dans ce cas). Vous devez explicitement utiliser `delete()` sur `Child`.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

Dans notre cas, un `Child` ne peut pas vraiment exister sans son père. Si nous effacons un `Child` de la collection, nous voulons vraiment qu'il soit effacé. Pour cela, nous devons utiliser `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

A noter : même si le mapping de la collection spécifie `inverse="true"`, les cascades sont toujours assurées par l'itération sur les éléments de la collection. Donc, si vous avez besoin qu'un objet soit enregistré, effacé ou mis à jour par cascade, vous devez l'ajouter dans la colleciton. Il ne suffit pas d'appeler explicitement `setParent()`.

16.4. Utiliser `update()` en cascade

Supposons que nous ayons chargé un `Parent` dans une `Session`, et que nous l'ayons ensuite modifié et que voulions persiter ces modifications dans une nouvelle session (en appelant `update()`). Le `Parent` contiendra une collection de fils et, puisque la cascade est activée, Hibernate a besoin de savoir quels fils viennent d'être instanciés et quels fils proviennent de la base de données. Supposons aussi que `Parent` et `Child` ont tous deux des identifiants (terchniques) du type `java.lang.Long`. Hibernate utilisera la propriété de l'identifiant pour déterminer quels fils sont nouveaux (vous pouvez aussi utiliser la propriété version ou timestamp, voir Section 9.4.2, « Mise à jour d'objets détachés »).

L'attribut `unsaved-value` est utilisé pour spécifier la valeur déterminant qu'une instance est nouvellement

instanciée. La valeur par défaut de `unsaved-value` est "null", ce qui est parfait pour les identifiants de type `Long`. Si nous avions une propriété identifiant d'un type primitif, nous devrions spécifier

```
<id name="id" type="long" unsaved-value="0">
```

pour le mapping de `Child` (Il existe aussi un attribut `unsaved-value` pour le mapping des propriétés version et timestamp).

Le code suivant, mettra à jour `parent` et `child` et insèrera `newChild`.

```
//parent et child ont été chargés dans une session précédente
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Ceci est très bien pour des identifiants générés, mais qu'en est il des identifiants assignés et des identifiants composés ? C'est plus difficile, puisque `unsaved-value` ne permet pas de distinguer un objet nouvellement instancié d'un objet chargé dans une session précédente (puisque celui-ci est assigné par l'utilisateur). Dans ce cas, vous devrez aider Hibernte, soit

- définir `unsaved-value="null"` ou `unsaved-value="negative"` dans le mapping de `<version>` ou `<timestamp>`.
- définir `unsaved-value="none"` et appeler explicitement `save()` sur les fils nouvellement instanciés avant d'appeler `update(parent)`
- définir `unsaved-value="any"` et appeler explicitement `update()` sur les fils précédement persistant avant d'appeler `update(parent)`

`none` est la valeur par défaut de `unsaved-value` pour les identifiant assignés ou composés.

Il existe une dernière possibilité. `Interceptor` possède une nouvelle méthode nommée `isUnsaved()` qui vous permet d'implémenter votre propre stratégie pour distinguer les objets nouvellement instanciés. Par exemple, vous pouvez définir une classe de base pour vos classes persistantes.

```
public class Persistent {
    private boolean _saved = false;
    public void onSave() {
        _saved=true;
    }
    public void onLoad() {
        _saved=true;
    }
    ......
    public boolean isSaved() {
        return _saved;
    }
}
```

(La propriété `saved` est non-persistante). Il faut maintenant implémenter`isUnsaved()`, `onLoad()` et `onSave()` comme suit :

```
public Boolean isUnsaved(Object entity) {
    if (entity instanceof Persistent) {
        return new Boolean( !( (Persistent) entity ).isSaved() );
    }
    else {
        return null;
```

```
    }
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onLoad();
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onSave();
    return false;
}
```

16.5. Conclusion

Il y a quelques principes à maîtriser dans ce chapitre et tout cela peut paraître déroutant la première fois. Cependant, dans la pratique, tout fonctionne parfaitement. La plupart des applications Hibernate utilisent le pattern père / fils.

Nous avons évoqué une alternative dans le premier paragraphe. Aucun des points traités précédemment n'existe dans le cas d'un mapping `<composite-element>` qui possède exactement la sémantique d'une relation père / fils. Malheureusement, il y a deux grandes limitations pour les classes éléments composites : les éléments composites ne peuvent contenir de collections, et ils ne peuvent être les fils d'entités autres (cependant, ils *peuvent* avoir une clé primaire technique, en utilisant un mapping `<idbag>`).

Chapitre 17. Exemple : Application de Weblog

17.1. Classes persistantes

Les classes persistantes representent un weblog, et un article posté dans un weblog. Il seront modélisés comme une relation père/fils standard, mais nous allons utiliser un "bag" trié au lieu d'un set.

```
package eg;
import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
```

```
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

17.2. Mappings Hibernate

Le mapping XML doit maintenant être relativement simple à vos yeux.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS"
        lazy="true">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            lazy="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">
```

```
    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

17.3. Code Hibernate

La classe suivante montre quelques utilisations que nous pouvons faire de ces classes.

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Query;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }
```

```
    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.save(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
```

```
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
```

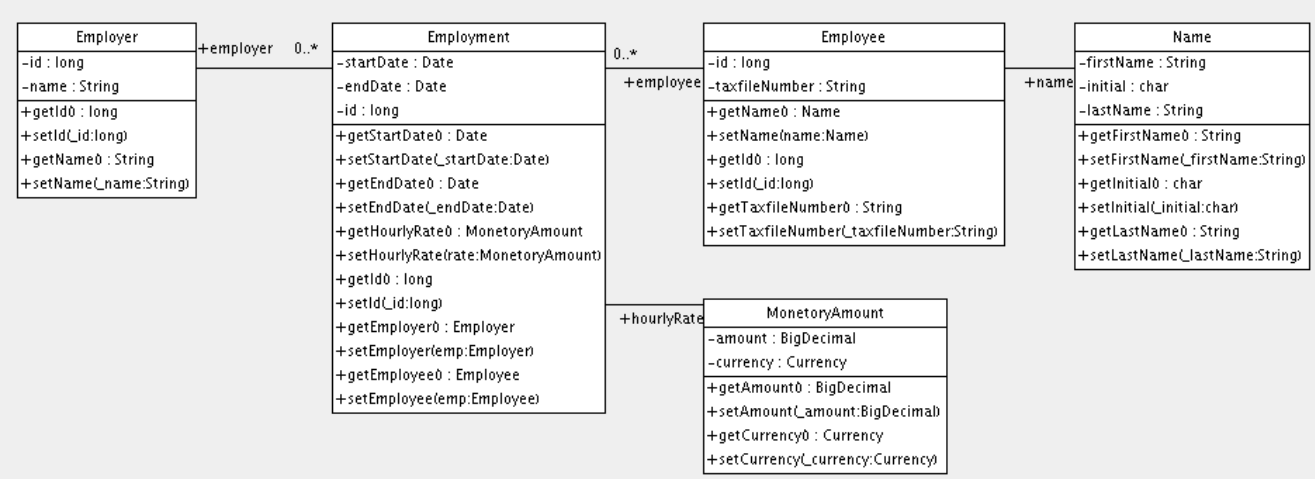
```
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.list().get(0);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

Chapitre 18. Exemple : Quelques mappings

Ce chapitre montre quelques mappings plus complexes.

18.1. Employeur/Employé (Employer/Employee)

Le modèle suivant de relation entre `Employer` et `Employee` utilise une vraie classe entité (`Employment`) pour représenter l'association. On a fait cela parce qu'il peut y avoir plus d'une période d'emploi pour les deux mêmes parties. Des composants sont utilisés pour modéliser les valeurs monétaires et les noms des employés.



Voici un document de mapping possible :

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetoryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
```

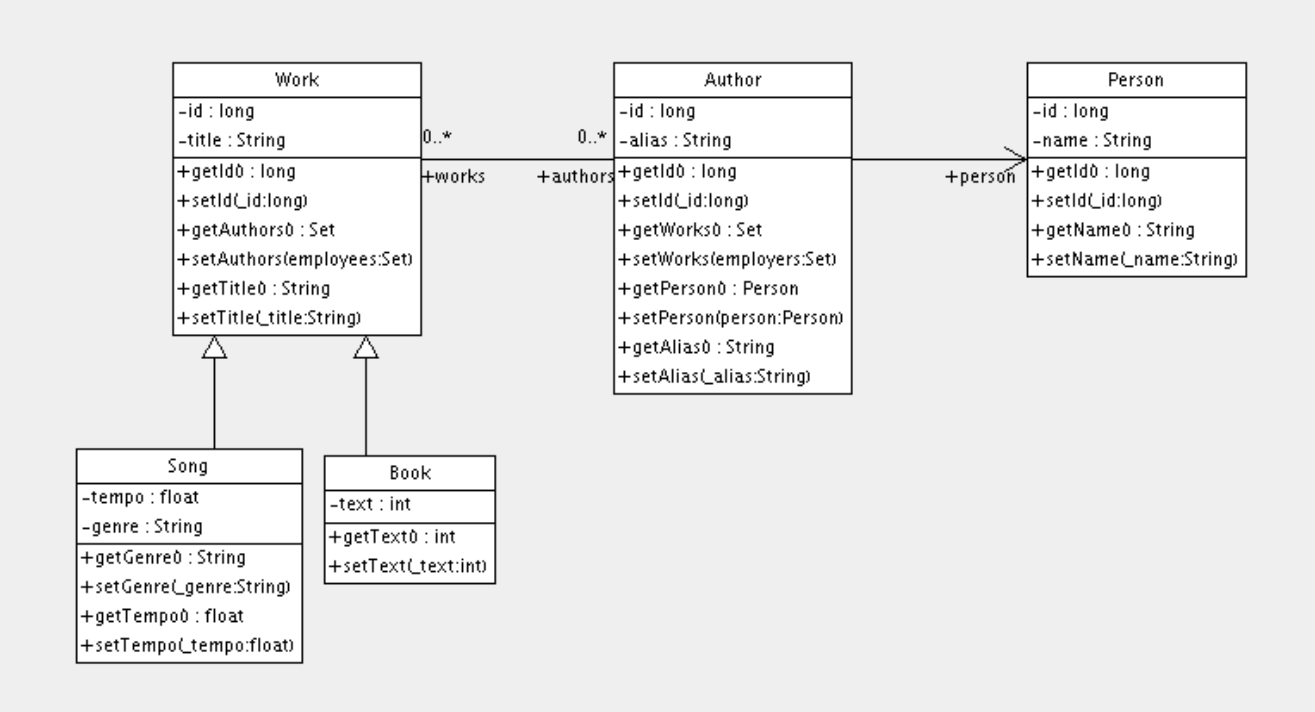
```
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

Et voici le schéma des tables générées par `SchemaExport`.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

18.2. Auteur/Travail (Author/Work)

Soit le modèle de la relation entre `Work`, `Author` et `Person`. Nous représentons la relation entre `Work` et `Author` comme une association plusieurs-vers-plusieurs. Nous avons choisi de représenter la relation entre `Author` et `Person` comme une association un-vers-un. Une autre possibilité aurait été que `Author` hérite de `Person`.



Le mapping suivant représente exactement ces relations :

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work" lazy="true">
            <key>
                <column name="work_id" not-null="true"/>
            </key>
            <many-to-many class="Author">
                <column name="author_id" not-null="true"/>
            </many-to-many>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- L'Author doit avoir le même identifiant que Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true" lazy="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
```

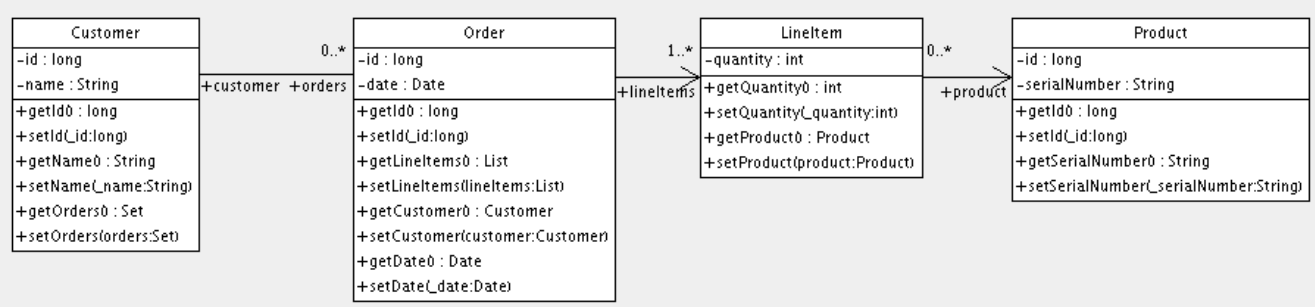
```
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Il y a quatre tables dans ce mapping. `works`, `authors` et `persons` qui contiennent respectivement les données de work, author et person. `author_work` est une table d'association qui lie authors à works. Voici le schéma de tables, généré par `SchemaExport`.

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

18.3. Client/Commande/Produit (Customer/Order/Product)

Imaginons maintenant le modèle de relation entre `Customer`, `Order`, `LineItem` et `Product`. Il y a une association un-vers-plusieurs entre `Customer` et `Order`, mais comment devrions nous représenter `Order` / `LineItem` / `Product`? J'ai choisi de mapper `LineItem` comme une classe d'association représentant l'association plusieurs-vers-plusieurs entre `Order` et `Product`. Dans Hibernate, on appelle cela un élément composite.



Le document de mapping :

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true" lazy="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items" lazy="true">
            <key column="order_id"/>
            <index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

`customers`, `orders`, `line_items` et `products` contiennent les données de customer, order, order line item et product. `line_items` est aussi la table d'association liant orders à products.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
```

```
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

Chapitre 19. Meilleures pratiques

Découpez finement vos classes et mappez les en utilisant `<component>`.

Utilisez une classe `Addresse` pour encapsuler `Rue`, `Region`, `CodePostal`. Ceci permet la réutilisation du code et simplifie la maintenance.

Déclarez des propriétés d'identifiants dans les classes persistantes.

Hibernate rend les propriétés d'identifiants optionnelles. Il existe beaucoup de raisons pour lesquelles vous devriez les utiliser. Nous recommandons que vous utilisiez des identifiants techniques (générés, et sans connotation métier) et de type non primitif. Pour un maximum de flexibilité, utilisez `java.lang.Long` ou `java.lang.String`.

Placez chaque mapping de classe dans son propre fichier.

N'utilisez pas un unique document de mapping. Mappez `com.eg.Foo` dans le fichier `com/eg/Foo.hbm.xml`. Cela prend tout son sens lors d'un travail en équipe.

Chargez les mappings comme des ressources.

Déployez les mappings en même temps que les classes qu'ils mappent.

Pensez à externaliser les chaînes de caractères.

Ceci est une bonne habitude si vos requêtes appellent des fonctions SQL qui ne sont pas au standard ANSI. Cette externalisation dans les fichiers de mapping rendra votre application plus portable.

Utilisez les variables "bindées".

Comme en JDBC, remplacez toujours les valeurs non constantes par "?". N'utilisez jamais la manipulation des chaînes de caractères pour remplacer des valeurs non constantes dans une requête ! Encore mieux, utilisez les paramètres nommés dans les requêtes.

Ne gérez pas vous mêmes les connexions JDBC.

Hibernate laisse l'application gérer les connexions JDBC. Vous ne devriez gérer vos connexions qu'en dernier recours. Si vous ne pouvez pas utiliser les systèmes de connexions livrés, réfléchissez à l'idée de fournir votre propre implémentation de `net.sf.hibernate.connection.ConnectionProvider`.

Pensez à utiliser les types utilisateurs.

Supposez que vous ayez une type Java, de telle bibliothèque, qui a besoin d'être persisté mais qui ne fournit pas les accesseurs nécessaires pour le mapper comme composant. Vous devriez implémenter `net.sf.hibernate.UserType`. Cette approche libère le code de l'application de l'implémentation des transformations vers / depuis les types Hibernate.

Utiliser du JDBC pur dans les goulets d'étranglement.

Dans certaines parties critiques de votre système d'un point de vue performance, quelques opérations (exemple : update et delete massifs) peuvent tirer partie d'un appel JDBC natif. Mais attendez de *savoir* que c'est un goulet d'étranglement. Ne supposez jamais qu'un appel JDBC sera forcément plus rapide. Si vous avez besoin d'utiliser JDBC directement, ouvrez une `Session` Hibernate et utilisez la connexion SQL sous-jacente. Ainsi vous pourrez utiliser la même stratégie de transation et la même gestion des connexions.

Comprendre le flush de `Session`.

De temps en temps la Session synchronise ses états persistants avec la base de données. Les performances seront affectées si ce processus arrive trop souvent. Vous pouvez parfois minimiser les flush non nécessaires en désactivant le flush automatique ou même en changeant l'ordre des opérations menées dans une transaction particulière.

Dans une architecture à trois couches, pensez à utiliser `saveOrUpdate()`.

Quand vous utilisez une architecture à base de servlet / session bean, vous pourriez passer des objets chargés dans le bean session vers et depuis la couche servlet / jsp. Utilisez une nouvelle session pour traiter chaque requête. Utilisez `Session.update()` ou `Session.saveOrUpdate()` pour mettre à jour l'état persistant de votre objet.

Dans une architecture à deux couches, pensez à utiliser la déconnexion de session.

Les transactions de bases de données doivent être aussi courtes que possible pour une meilleure scalabilité. Cependant, il est souvent nécessaire d'implémenter de longues transactions applicatives, une simple unité de travail du point de vue de l'utilisateur. La transaction applicative peut s'étaler sur plusieurs cycles de requêtes/réponses du client. Utilisez soit les objets détachés ou, dans une architecture deux tiers, déconnectez simplement la session Hibernate de la connexion JDBC et reconnectez la à chaque requête suivante. N'utilisez jamais une seule session pour plus d'un cas d'utilisation de type transaction applicative, sinon vous vous retrouverez avec des données obsolètes.

Connsidérer que les exceptions ne sont pas rattrapables.

Il s'agit plus d'une pratique obligatoire que d'une "meilleure pratique". Quand une exception intervient, il faut faire un rollback de la `Transaction` et fermer la `Session`. Sinon, Hibernate ne peut garantir l'intégrité des états persistants en mémoire. En particulier, n'utilisez pas `Session.load()` pour déterminer si une instance avec un identifiant donné existe en base de données, utilisez `find()` (ou `get()`) à la place. Quelques exceptions sont récupérables, par exemple `StaleObjectStateException` et `ObjectNotFoundException`.

Préférez le chargement tardif des associations.

Utilisez le chargement complet (simple ou par jointure ouverte) avec modération. Utilisez les proxies et/ou les collections chargées tardivement pour la plupart des associations vers des classes qui ne sont pas en cache de niveau JVM. Pour les assocations de classes en cache, où il y a une forte probabilité que l'élément soit en cache, désactivez explicitement le chargement par jointures ouvertes en utilisant `outer-join="false"`. Lorsqu'un chargement par jointure ouverte est approprié pour un cas d'utilisation particulier, utilisez une requête avec un `left join fetch`.

Pensez à abstraite votre logique métier d'Hibernate.

Cachez le mécanisme d'accès aux données (Hibernate) derrière une interface. Combinez les patterns *DAO* et *Thread Local Session*. Vous pouvez même avoir quelques classes persistées par du JDBC pur, associées à Hibernate via un `UserType` (ce conseil est valable pour des applications de taille respectables ; il n'est pas valable pour une application avec 10 tables).

Implémentez `equals()` et `hashCode()` en utilisant une clé métier.

Si vous comparez des objets en dehors de la session, vous devez implémenter `equals()` et `hashCode()`. A l'intérieur de la session, l'identité des objets java est assurée. Si vous implémentez ces méthodes, n'utilisez jamais les identifiants de la base de données ! Une instance transiante n'a pas de valeur d'identifiant et Hibernate en assignera une quand l'objet sera sauvé. Si l'objet est dans un Set quand il est en cours de sauvegarde, le hashcode changera donc, ce qui rompt le contrat. Pour implémenter `equals()` et `hashCode()`, utilisez une clé métier unique ce qui revient à comparer une combinaison de propriétés de classe. Souvenez vous que cette clé doit être stable et unique pendant la durée durant laquelle l'objet est dans un Set, et non pour tout son cycle de vie (pas aussi stable que la clé primaire de la base de données). ne comparez jamais des collections avec `equals()` (chargement tardif) et soyez prudents avec les autres classes dont vous pourriez n'avoir qu'un proxy.

N'utilisez pas d'associations de mapping exotiques.

De bons cas d'utilisation pour de vraies associations plusieurs-vers-plusieurs sont rares. La plupart du temps vous avez besoin d'informations additionnelles stockées dans la table d'association. Dans ce cas, il est préférable d'utiliser deux associations un-vers-plusieurs vers une classe de liaisons intermédiaire. En fait, nous pensons que la plupart des associations sont de type un-vers-plusieurs ou plusieurs-vers-un, vous devez être très attentifs lorsque vous utilisez autre chose et vous demander si c'est vraiment nécessaire.