Institut **S**upérieur d'**I**nformatique
Modélisation et leurs **A**pplications
Complexe des Cézeaux – BP 125
63173 AUBIERE CEDEX

Tutorial - Cours Java

3e année F5

Framework Hibernate

Présenté par :
Guillaume CRESTA
GATCHA Charles
MOUNISSAMY Sivakumar

Responsable tutorial :

M. Cédric TESSIER

Octobre 2005

Institut **S**upérieur d'**I**nformatique
Modélisation et leurs **A**pplications
Complexe des Cézeaux – BP 125
63173 AUBIERE CEDEX

Tutorial - Cours Java

3e année F5

Framework Hibernate

Présenté par :
Guillaume CRESTA
GATCHA Charles
MOUNISSAMY Sivakumar

Responsable tutorial :

M. Cédric TESSIER

Octobre 2005

SOMMAIRE

SOMMAIRE

I NTRODUCTI ON

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. Un outil de mapping objet/relationnel pour le monde de java a donc été mise en place : Hibernate. 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.

Suite à cela, de nombreuses versions d'Hibernate ont été mises à disposition des utilisateurs afin de le faire évoluer. Hibernate se base sur l'architecture Modèle – Vue – Contrôleur qui fut introduite comme partie du SmallTalk80 afin de limiter les efforts de programmation liés à l'élaboration de système. Hibernate est agnostique en terme d'architecture, il peut donc être utilisé aussi bien dans un développement client lourd que dans un environnement web léger de type Tomcat (Apache) ou dans un environnement J2EE complet (Weblogic, Websphere, JBoss).

Notre tutorial s'organisera donc de la façon suivante. Nous verrons dans une première partie l'introduction aux frameworks. Dans un second temps, nous aborderons une application Web afin d'observer la configuration d'Hibernate et la déclaration des objets. Pour finir, nous développerons les différents mappings utiles suivis de quelques exemples avant d'apporter une rapide conclusion à ce tutorial.

I INTRODUCTION AUX FRAMEWORK

I.1 Définition d'un framework

Le framework désigne le cadre dans lequel va s'insérer une application. En programmation orientée objet, il désigne l'infrastructure logicielle qui facilite la conception des applications par l'utilisation de bibliothèques de classes ou de générateurs de programmes. C'est une bibliothèque de classes fournissant une ossature générale pour le développement d'une application dans un domaine particulier. Ces composants sont organisés afin d'être utilisés en interaction les uns avec les autres et sont spécifiques généralement à un type d'application.

Les frameworks facilitent ainsi le travail du développement en fournissant un squelette d'application qu'il suffit de remplir pour l'adapter à ses besoins. La contrepartie est qu'un framework représente un sur ensemble de tous les besoins génériques, ce qui conduit à supporter un grand nombre de choses même si souvent, une toute petite partie est utile pour le cas à réaliser. Ceci complique également la tâche d'apprentissage et d'assimilation de l'environnement par le développeur.

Un framework est un ensemble de classes abstraites qui, dans le but de faciliter la création d'une partie d'un système logiciel, collaborent entre elles. Il partage le domaine visé en classes abstraites et permet de définir les responsabilités de chacune par rapport aux autres et les rapports entre elles.

I.2 Différents exemples de framework

I.2.1 MFC

Les MFC de Microsoft est un exemple de framework qui permettent de développer une application en C++ basée sur une architecture *Fenêtre Cadre-Document-Vue*.

MFC est donc une bibliothèque de classes C++ fournissant un cadre général pour la programmation sous Windows. MFC encapsule une grande partie des fonctions API Windows et améliore la logique de la création des interfaces.

L'objet principal du MFC est donc de fournir un cadre général pour développer facilement des interfaces graphiques. Pour cela, il fournit une architecture de programme basée sur la notion de document et vue. Pour les MFC, une interface graphique est un moyen parmi d'autres de visualiser des données (document) ; cette visualisation (vue) s'effectuant dans une fenêtre cadre elle-même pilotée par un programme principal.

I.2.2 Jarka Struts

Struts est un exemple de framework open source, basé sur l'architecture MVC (Modèle Vue Contrôleur). C'est un projet d'un framework faisant partie de Apache Jarka Project. Il sert à développer des applications pour le Web.

Le cœur du framework Struts est une couche contrôleur basée sur les technologies les plus acceptées comme le JSP, le JavaBeans, et le XML. Struts encourage les architectures basées sur l'approche Model 2, qui est une variante du modèle classique MVC. Struts fournit son propre composant contrôleur et intègre d'autres technologies pour offrir le Modèle et la Vue. Pour le modèle, Struts peut interagir avec toutes les technologies d'accès aux données comme les EJB, le JDBC. Pour la vue, Struts fonctionne bien avec les JSP, les Velocity Templates et d'autres systèmes de présentation.

I.2.3 Spring

Spring est un conteneur dit « léger », c'est-à-dire une infrastructure similaire à un serveur d'application J2EE. Il prend donc en charge la création d'objets et leurs mises en relation par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ceux-ci.

Le gros avantage par rapport aux serveurs d'application est qu'avec Spring, vos classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications J2EE et des EJBs). C'est en ce sens que Spring est qualifié de conteneur « léger ».

Outre cette espèce de fabrication d'objets, Spring propose tout un ensemble d'abstractions permettant de gérer entre autres :

- Le mode transactionnel
- L'appel d'EJB
- La création d'EJB
- La persistance d'objets
- La création d'une interface Web
- L'appel et la création de WebServices

Pour réaliser tout ceci, Spring s'appuie sur les principes du design pattern IoC et sur la programmation par aspects (AOP).

I.3 Framework Hibernate

I.3.1 Définition Hibernate

Hibernate est un logiciel écrit sous la responsabilité de Gavin King, qui fait entre autre partie de l'équipe de développement de JBoss.

Hibernate est un framework en open source, gérant la persistance des objets (qui peuvent être défini par les propriétés, les méthodes ou les évènements qu'il est susceptible de déclencher) dans une base de données relationnelle. La persistance des objets représente la possibilité d'enregistrement de l'état d'un objet, par exemple dans une base de données, afin de pouvoir le recréer plus tard. Quant à la base de données relationnelle, elle contient de nombreuses tables et l'information est organisée par différentes relations entre tables. Pour information, on appelle SGBDR un logiciel mettant en œuvre une telle base de données.

L'ensemble des données nécessaires au fonctionnement de l'application est sauvegardé dans une base de données. La manipulation des données peut se faire de différentes manières : par l'accès directement à la base en écrivant les requêtes SQL adéquates, utiliser un outil d'ORM (Object Relationnal Mapping) permettant de manipuler facilement les données et d'assurer leur persistance : c'est le cas d'Hibernate.

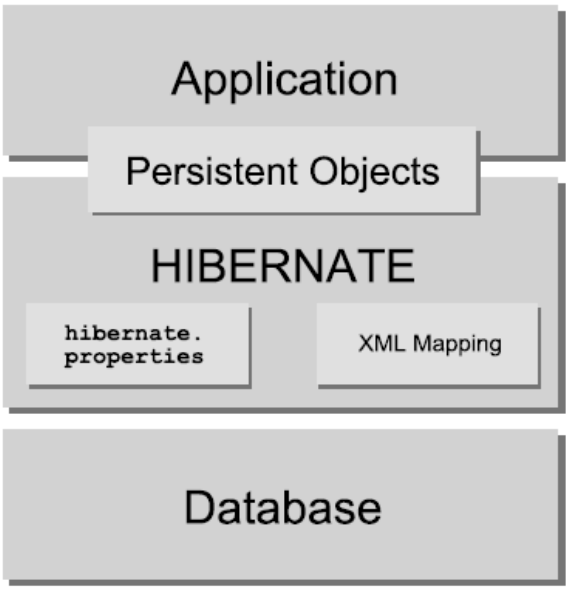
Pourquoi ajouter une couche entre l'application et la base de données ? L'objectif est de réduire le temps de développement de l'application en éliminant une grande partie du code SQL à écrire pour interagir avec la base de données et en encapsulant le code SQL résiduel. Les développeurs manipulent les classes dont les données doivent être persistantes comme des classes Java normales. Seule une initialisation correcte d'Hibernate doit être effectuée, et quelques règles respectées lors de l'écriture et de la manipulation des classes persistantes.

Hibernate se place à un autre niveau que le framework Struts qui lui gère l'interface homme-machine et se base sur l'architecture MVC. Hibernate est agnostique en terme d'architecture, il peut donc être utilisé aussi bien dans un développement client lourd que dans un environnement Web léger de type Tomcat (Apache) ou dans un environnement J2EE complet (Weblogic, Websphere, JBoss).

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 faire des requêtes sur 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 % des tâches de programmation liées à la persistance 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 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.

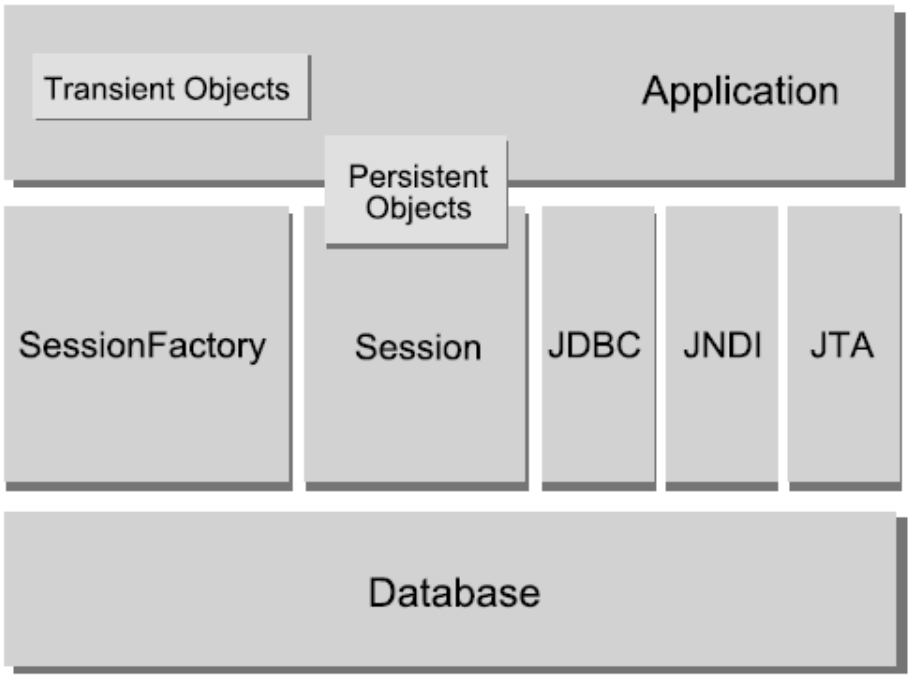
I.3.2 Architecture d'Hibernate

Nous allons voir maintenant l'architecture d'Hibernate. Ci-après, vous pouvez voir une vue haut niveau de l'architecture d'Hibernate.

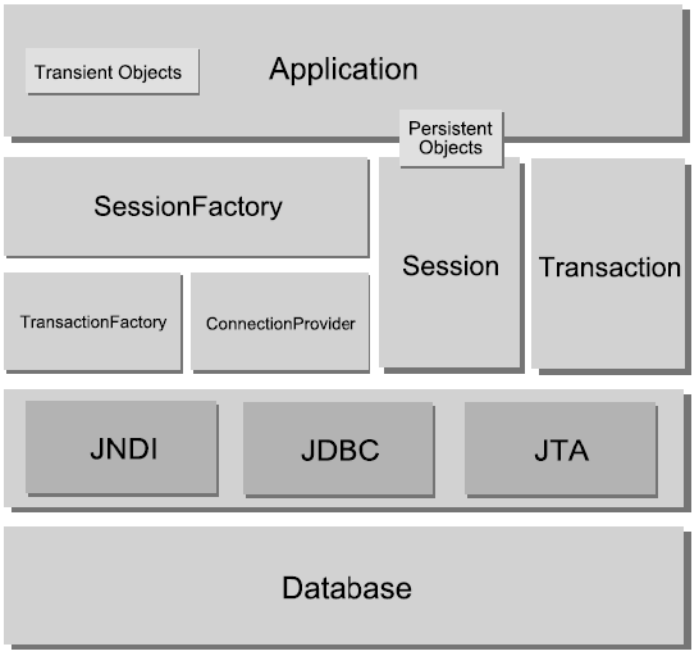


Ce diagramme montre Hibernate utilisant une base de données et des données de configuration pour fournir un service de persistance (et des objets persistants) à l'application. Hibernate est flexible et supporte différentes approches. En voici deux approches très extrêmes.

L'architecture légère laisse l'application fournir ses propres connexions JDBC et gérer ses propres transactions. Cette approche utilise le minimum des API Hibernate :



L'architecture la plus complète fait abstraction de l'application des API JDBC/JTA sous-jacentes et laisse Hibernate s'occuper des détails.



Nous allons maintenant définir les objets des diagrammes :

SessionFactory :

Un cache immuable (threadsafe) des mappings vers une base de données. Il peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au niveau du processus ou au niveau du cluster.

Session

Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Elle encapsule une connexion JDBC. Elle 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 POJO) ; la seule particularité est qu'ils sont associés avec une session. Dès que la session est fermée, ils seront détachés et libre d'être utilisés par n'importe laquelle des couches de l'application, c'est à dire de et vers la présentation en tant que Data Transfer Objects (objets de transfert de données).

Objets et collections passagers

Instances de classes persistantes qui ne sont actuellement pas associées à une session. Elles ont pu être mises en instances par l'application, et ne pas avoir (encore) été persistées ou par une session fermée.

Transaction

Objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique et qui abstrait l'application des transactions sous-jacentes qu'elles soient JDBC, JTA ou CORBA. Une session peut fournir plusieurs transactions dans certain cas.

ConnectionProvider

Lieu de fabrication de connexions JDBC. Elle abstrait l'application de la source de données ou du manager sous-jacent de pilotes. Elle n'est pas exposée à l'application, mais peut être étendue et implémentée par le développeur.

TransactionFactory

Fabrique d'instances de Transaction. Non exposée à l'application, mais peut être étendue et implémenté par le développeur.

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

I.3.3 Comparatif avec Hibernate

Les recherches ont fait ressortir trois framework de persistance dont les fonctionnalités sont intéressantes : Hibernate, JPOX et OJB. Nous avons alors établi un tableau récapitulatif des avantages et des inconvénients de chacun.

Framework	Avantages	Inconvénients
Hibernate	- Propose une API performante et robuste - Est mature et dispose d'un support fiable et d'une documentation abondante - Supporte la gestion des transactions - S'intègre facilement à Eclipse et Spring - Appropriation rapide	- Ne se base pas sur les spécifications et standards JDO ou ODMG - Nécessite un module pour la connectivité aux bases Oracle - Possibilité de perte de vitesse après l'acceptation des spécifications JDO 2.0
JPOX	- Une implémentation de référence des JSR JDO 1.0 et 2.0 - En gain de crédibilité - Projet promu par SUN	- Framework en version alpha - Documentation insuffisante
OJB	- Pas de RoadMap pour l'implémentation des spécifications JDO 2.0	- Implémentation des spécifications JDO 1 .0

JPOX reste plus complet qu'OJB en terme de gestion transactionnelle et de conformité aux dernières JDO. Il a été prouvé que le framework de persistance le plus intéressant reste malgré tout le JPOX.

Hibernate est beaucoup plus utilisé que JDO, une norme qui gère la persistance des objets. Par contre JDO ne se limite pas seulement aux bases de données relationnelles ; il peut gérer notamment le XML. Hibernate ne respecte pas les specs JDO. Il ne le fera d'ailleurs jamais, car Hibernate constitue l'implémentation de référence qui a servi à monter les specs EJB 3.0.

Hibernate est beaucoup moins lourd à développer et à administrer que les EJB entité dans les versions 2. La norme EJB 3 en phase d'élaboration, reprend les expériences et le modèle de programmation d'Hibernate ou de Toplink. Hibernate dans sa version 3 supporte la norme EJB 3 et serait désigné comme LE standard de l'avenir par SUN.

II APPLICATION WEB

II.1 Configuration d'Hibernate

Dans un projet, Hibernate peut permettre de travailler efficacement avec une base via java et aussi de changer de base de données sans rien toucher au code. Nous allons donc commencer par configurer Hibernate. Lorsque l'on programme en Java, on configure beaucoup.

Sur hibernate.org, télécharger la dernière version, à ce jour la 3.0. Pour l'application Web on utilise un logiciel préalablement installé : NetBeans. Dans un dossier « projects » créé lors de l'installation du logiciel, on va établir un nouveau dossier pour y stocker toutes nos librairies. Il faut donc faire le dossier « projects/lib/ » et y mettre toutes les librairies utiles pour faire fonctionner Hibernate. Les librairies se situent à la racine et dans le dossier « Hibernate » de l'archive Hibernate. On prend seulement ce qui est utilisée habituellement (description ci-dessous) :

```
Hibernate.jar, tous les commons-*.jar, jdbc2.0.jar, antlr, asm, cglib, dom4j, ehcache, jta, log4j, odmg4
```

Bibliothèque	Description
dom4j	Hibernate utilise dom4j pour lire la configuration XML et les fichiers XML de métadonnées du mapping
CGLIB	Hibernate utilise cette bibliothèque de génération de code pour étendre les classes à l'exécution
Commons collections, commons logging	Hibernate utilise diverses bibliothèques du projet Apache Jakarta Commons
ODMG4	Hibernate est compatible avec l'interface de gestion de la persistance telle que définie par l'ODMG. Elle est nécessaire si vous voulez mapper des collections même si on n'a pas l'intention d'utiliser l'API d'ODMG
EHCache	Hibernate peut utiliser diverses implémentations de cache de second niveau. EH est l'implémentation par défaut.
Log4j	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 dans le classpath. Un exemple de fichier est implenté dans la distribution Hibernate.

Ensuite, dans votre projet, faire un clic droit sur le dossier « Librairies », puis « add » puis cliquer sur « Manage librairies ». Il est déconseillé d'être désordonné à ce niveau là de l'arborescence. Il est préférable de bien ranger ses librairies dès le départ. Cliquez sur « new librairie » et faites en une pour Hibernate, ensuite dans « classpath » cliquez sur « add jar's »et il faut y mettre :

```
Hibernate.jar, antlr, asm, cglib, dom4j, ehcache
```

Il suffit de faire de même avec Commons en y plaçant tous les commons.jar puis une autre pour Log4j, une autre pour JTA… Pour finir, il faut ajouter par précaution une librairie avec le driver de la base de données (librairie MySQL et le JDBC2.0.jar par exemple). Ensuit, il reste à fermer la fenêtre de gestion des JAR, on se retrouve dans « add », et on choisit les packages que l'on désire créer et on clique sur « add library », un logiciel comme NetBeans va les ajouter au projet.
Après avoir inséré les librairies, on va configurer Hibernate. Tout d'abord, faites un clic droit sur « sources packages » et puis sur « new → xml document », nommez le fichier « hibernate.cfg » et copiez / collez le code suivant :

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

<hibernate-configuration>
    <session-factory>
        <property name="connection.datasource">
           java:comp/env/jdbc/mabase
        </property>
        <property name="show_sql">true</property>
        <property name="dialect">
              org.hibernate.dialect.MySQLDialect
        </property>
        <!>
        <mapping resource="User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Ceci est un fichier de configuration assez minimal. Il informe Hibernate d'utiliser notre Datasource, d'afficher les messages SQL, d'utiliser le dialecte MySQL (cela est changé en fonction de la base de donnée sur laquelle on travaille). Enfin nous avons créé un mapping (cf. § III) vers une ressource « user » qui nous servira à utiliser notre table Users de la dernière fois.

En effet, dans Hibernate il faut définir un fichier de configuration par objet, celui utilisant des POJO pour mapper les données avec la table. Hibernate se charge de remplir ces objets avec vos données, ou bien de les envoyer dans la base.

II.2 Déclaration de l'objet Utilisateur

A l'image de la méthode utilisée pour le fichier de configuration d'Hibernate, il faut faire celui de l'objet « User », et créer le fichier « User.hbm.xml » et faire en sorte qu'il soit semblable au code ci-après.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.cresta.hibernate.User" table="users">
        <!>
        <id name="id" type="int" column="id" unsaved-value="0">
            <generator class="identity"/>
        </id>
        <!>
        <property name="nom">
            <column name="nom" length="55" not-null="true"/>
        </property>
        <!>
        <property name="prenom">
            <column name="prenom" length="55" not-null="true"/>
        </property>
    </class>
</hibernate-mapping>
```

Ce fichier sert à faire le lien entre votre POJO et Hibernate. Ici on dit quel est le Bean (« com.cresta.hibernate.User ») et à quelle table il se rapporte, puis nous définissons ses colonnes pour qu'elles soient prise en charge par les Getter et Setter du Bean, il y a donc dans le code « id » pour l'identifiant, le « nom » et le « prenom ».

A présent, il faut créer la classe « User ». Pour cela, faire « file → new javaBean comp. » pour créer une classe de type Bean, dans par exemple « com.cresta.hibernate » et nommer la « User ». Dans la vue projet, on déplie la classe et on fait un clic droit sur « bean pattern » puis « add property », et on ajoute :

```
    private int id;
    private String prenom;
    private String nom;
```

Cela va nous générer directement le code pour le Bean qui doit être semblable au code figurant ci-dessous.

```
/* User.java */

package com.cresta.hibernate;

import java.io.Serializable;

public class User extends Object implements Serializable
{
    private int id;
    private String prenom;
    private String nom;

    public User() {  }

    public int getId()
    {   return id;  }

    public void setId(int value)
    {   id = value;  }

    public String getPrenom()
    {   return prenom;  }

    public void setPrenom(String value)
    {   prenom = value;  }

    public String getNom()
    {   return nom;  }

    public void setNom(String value)
    {   nom = value;  }
}
```

La configuration est complète et toutes les configurations ont été effectuées. Il suffit de faire à présent « compile » et « run » et le projet devrait démarrer sans aucun problème.

II.3 Servlet pour se connecter à une base de données

Nous allons maintenant créer une servlet pour se connecter à la base de données avec Hibernate.

Comme vue précédemment, la session dans Hibernate et le « persistence Manager » sont utilisés afin de mettre et récupérer des données depuis la base. Pour récupérer une Session, on l'effectue grâce à la SessionFactory comme ceci :

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

Une SessionFactory ne devrait être en général construite qu'une seule fois, par exemple avec une servlet au démarrage, grâce à un « load-on-startup » dans le Web.xml. Il faut créer une classe utilitaire qui va résoudre le problème, et vient directement de la documentation d'Hibernate. Pour cela, il faut faire un clic droit sur votre projet, puis « new » et « java class » et recopier le code ci-dessous.

```
package com.cresta.hibernate;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class HibernateUtil {
  private static Log log = LogFactory.getLog(HibernateUtil.class);
  private static final SessionFactory sessionFactory;
  static {
     try {
         // Create the SessionFactory
         sessionFactory =
               new configuration().configure().buildSessionFactory();
     } catch (Throwable ex) {
         // Make sure you log the exception, as it might be swallowed
         log.error("Initial SessionFactory creation failed." + ex);
         throw new ExceptionInInitializerError(ex);
     }
  }
  public static final ThreadLocal session = new ThreadLocal();
  public static Session currentSession() {
    Session s = (Session) session.get();
    // Open a new Session, if this Thread has none yet
    if (s == null) {
      s = sessionFactory.openSession();
      session.set(s);
    } return s;
  }
  public static void closeSession() {
    Session s = (Session) session.get();
   if (s != null)
      s.close(); session.set(null);
  }
}
```

A présent, il est possible d'utiliser Hibernate dans une servlet, afin de manipuler directement des objets contenant les données de la base. Afin de créer unes servlet, il faut faire « New → Servlet » dans NetBeans et nommer la « TestServlet » avec un mapping dans la configuration tel que « /testservlet ». Vérifier qu'elle fonctionne et qu'elle affiche quelque chose (il est conseillé d'enlever le commentaire figurant au passage « TODO output your page here » créé par défaut).

Nous aurons besoin de certains packages, qu'il faut importer :

```
import com.bourzeix.hibernate.HibernateUtil;
import com.bourzeix.hibernate.User;
import java.io.*;
import java.util.Iterator;
import javax.servlet.*;
import javax.servlet.http.*;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
```

Ensuite, il faut déclarer la Session et la Transaction Hibernate, ce sont les objets nécessaire à l'accès aux données :

```
Session session;
Transaction tx;
```

Afin de pouvoir lire la table par la suite, il va falloir y ajouter quelque chose. Il suffit donc de créer un utilisateur avec l'ajout du code suivant à notre servlet :

```
    //Creation de notre objet Session grace à notre HibernateUtil
        session = HibernateUtil.currentSession();
    //Ouverture de notre transaction avec Hibernate grâce à la session
        tx = session.beginTransaction();
//Ajout user utilisant notre bean User préalablement config dans Hibernate
        User toto = new User();
        toto.setNom("cresta");
        toto.setPrenom("guillaume");
    // On sauve, on renvoie, notre bean à la session Hibernate
        session.save(toto);
        tx.commit();   // Nous commitons la transaction vers la base
        HibernateUtil.closeSession(); //Enfin on ferme la session
```

Il suffit maintenant d'ouvrir plusieurs fois le servlet dans un navigateur web. Pour voir si tout se passe correctement, on ouvre une base de donnée pour voir si les utilisateurs sont présents. Normalement, on doit avoir autant d'utilisateurs que d'actualisation effectuée. Ils doivent tous avoir le même nom (en l'occurrence cresta) et tous le même prénom (guillaume dans notre exemple), mais l'identifiant est différent et a bien été géré automatiquement par Hibernate grâce à la configuration de notre objet « User ». C'est grâce au code figurant dans le fichier « user.hbm.xml »

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

Il reste donc à sélectionner et à afficher les utilisateurs grâce à notre servlet. Il suffit de copier le code ci-dessous :

```
        session = HibernateUtil.currentSession();
        tx = session.beginTransaction();
        Query query = session.createQuery("from User u");
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet TestServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet TestServlet at "
                + request.getContextPath() + "</h1>");
        for (Iterator it = query.iterate(); it.hasNext();) {
            User user = (User) it.next();
            out.println("<p>");
            out.println("Id : " + user.getId() );
            out.println("Nom : " + user.getNom() );
            out.println("Prénom : " + user.getPrenom() );
            out.println("</p>");}

        out.println("</body>");
        out.println("</html>");
        tx.commit();
        HibernateUtil.closeSession(); out.close();
```

On récupère la « session » puis on commence la transaction. On crée ensuite un objet « Query » grâce auquel on effectue une requête. On peut remarquer qu'Hibernate supporte des requêtes simplifiées. On aurait pu remplacer « from User u » par « select * from User u ». Par la suite, on utilise une itération pour boucler sur les résultats de la requête. A chaque passage, on crée un objet User et on utilise simplement ses Getter pour lire et afficher les valeurs. Pour finir, on ferme notre page HTML ainsi que toutes les connections et objets ouverts. Le résultat est donc le suivant :

```
Servlet TestServlet at /test
Id : 1 Nom : cresta Prénom : guillaume
Id : 2 Nom : cresta Prénom : guillaume
Id : 3 Nom : cresta Prénom : guillaume
Id : 4 Nom : cresta Prénom : guillaume
Id : 5 Nom : cresta Prénom : guillaume
Id : 6 Nom : cresta Prénom : guillaume
```

III Différents mapping objet/relationnel basiques

III.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).

III.1.1 Hibernate mapping

Cet élément possède trois attributs optionnels. L'attribut schéma 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 qualifiés. 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".

III.1.2 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)
```

L'élément class permet de déclarer une classe persistante.

(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 à 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. Lorsqu'un objet passager 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. Il faut alors déclarer les classes implémentant cette interface via l'élément <subclass>. Vous pouvez

persister n'importe quelle classe interne *statique*. Il suffit de 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.

Le polymorphisme *implicite* signifie que les instances de la classe seront retournées par une requête qui utilise les noms de la classe ou encore des interfaces implémentées par cette classe. 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> seront retournées. Dans la majorité 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.

Les paramètres *dynamic-update* et *dynamic-insert* ne sont pas hérités par les classes filles et peuvent donc être spécifiés dans les éléments <subclass>. Ces paramètres peuvent accroître les performances dans certains cas.

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.

Il est recommandé 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 (lors de l'utilisation de Session.update()).

III.1.3 Id (generator)

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.

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>
```

III.1.4 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.

III.1.5 Property

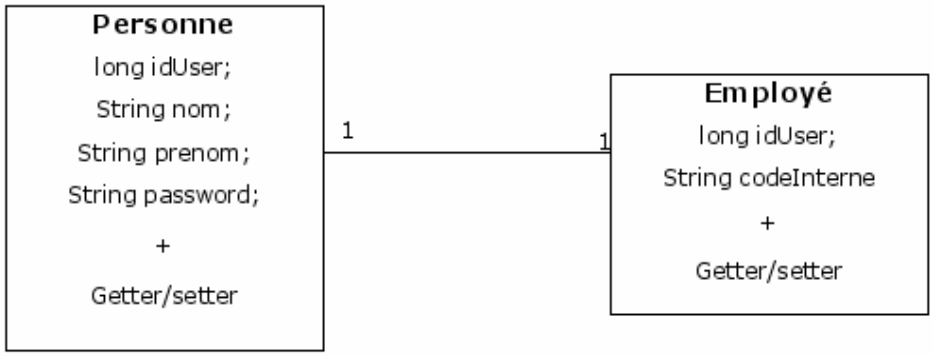
L'élément <property> déclare une propriété persistante de la classe, respectant la convention JavaBean.

```
<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 même colonne, 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.

III.1.6 One to one

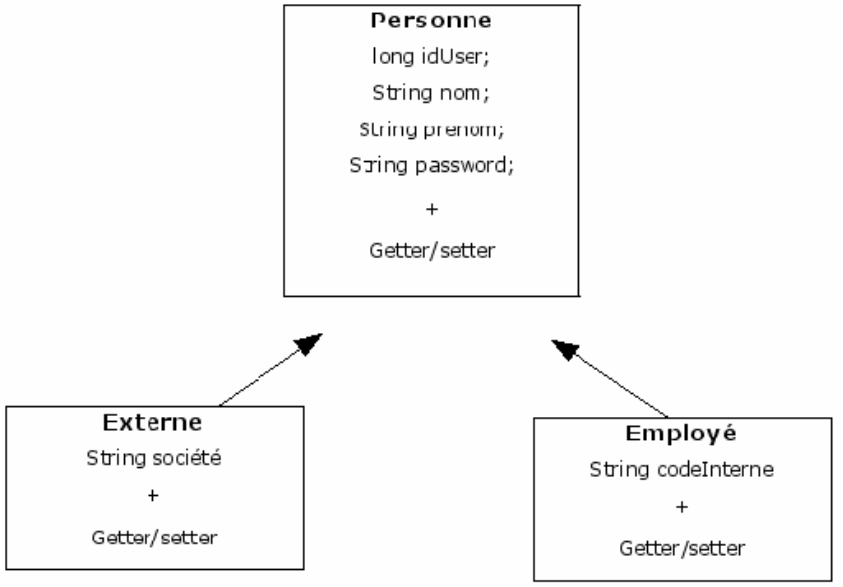


Exemple : une personne peut être associé à son statut d'employé par une relation. Une des possibilités de définition de la relation <one to one> dans les fichiers de mapping s'écrit comme présenté ci-dessous :

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

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

Les deux enregistrements associés auront la même clé primaire.

III.1.7 Héritage



Exemple : une personne peut être associée à son statut d'employé ou d'externe par une relation d'héritage. Cette technique peut se décompose en trois cas :

-table par hiérachie : création d'une seule table (personne avec 2 colonnes en plus société et codeInterne).

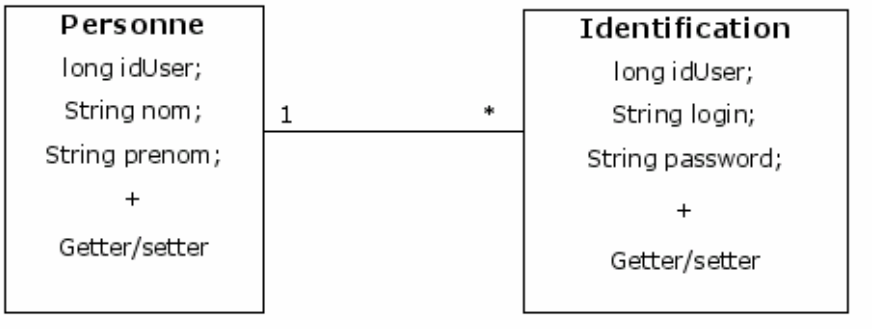
-table par classe fille : création de 3 tables (personne - employé - externe)

-table par classe concrète : création de deux tables (employé - externe)

Dans le premier cas, le fichier de mapping s'écrit :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<!-- Strategie table-per-class hierarchy mapping -->
<hibernate-mapping package="utilisateur4">
         <class name="Personne" table="PERSONNES" discriminator-value="P">
                <id name="idPers" column="idPers" type="long">
                        <generator class="sequence"/>
        </id>
        <discriminator column="sousclasse" type="character"/>
                <property name="nom" column="nom" type="string"/>
                <property name="prenom" column="prenom" type="string"/>
   <set name="identifications" inverse="true" cascade="all-delete-orphan">
                        <key column="pers"/>
            <one-to-many class="utilisateur4.Identification" />
                </set>
                <subclass name="Employe" discriminator-value="E">
                        <property name="codeInterne" column="codeInterne"
type="string"/>
                </subclass>
                <subclass name="Externe" discriminator-value="X">
                        <property name="societe" column="societe"
type="string"/>
                </subclass>
        </class>
</hibernate-mapping>
```

III.1.8 Many to one



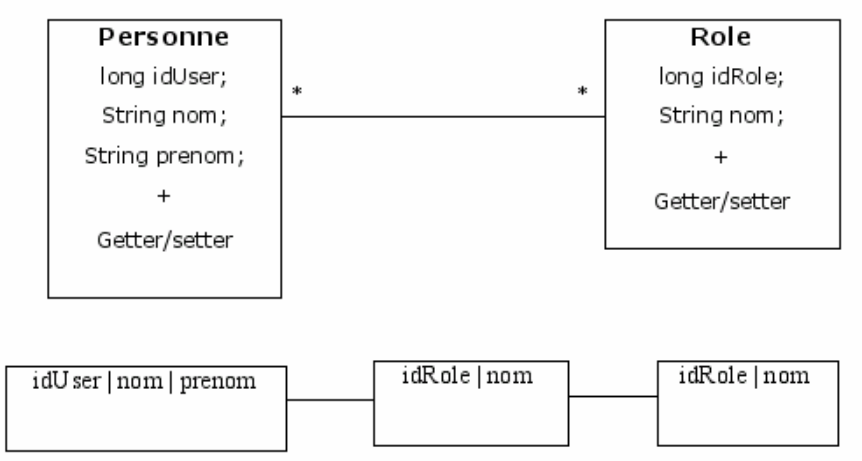
Exemple : une personne peut être associée à plusieurs identifications dans le cadre d'un logiciel gérant des profils. La classe « Personne » a un nouvel attribut, correspondant à une collection d'objet Identification, et la classe Identification à un nouveau champ correspondant à une personne. Le fichier de mapping de la classe Personne s'écrit :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<!-- Strategie table-per-class hierarchy mapping -->
<hibernate-mapping package="utilisateur3">
         <class name="Personne" table="PERSONNES" discriminator-value="P">
                <id name="idPers" column="idPers" type="long">
                        <generator class="sequence"/>
        </id>
        <discriminator column="sousclasse" type="character"/>
                <property name="nom" column="nom" type="string"/>
                <property name="prenom" column="prenom" type="string"/>
        <set name="identifications" inverse="true" cascade="all-delete-
orphan">
                 <key column="pers"/>
            <one-to-many class="utilisateur3.Identification" />
           </set>
        </class>
</hibernate-mapping>
```

La balise <set> permet d'identifier la collection qui contiendra les identifications. La clause cascade="all-delete-orphan" permet d'effacer les identifications correspondant à une personne lors de la suppression de la personne. Le fichier de mapping de la classe Identification s'écrit :

```
<?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="utilisateur3">
        <class name="Identification" table="UTILISATEUR2" >
                <id name="idUser" column="idUser" type="long">
                        <generator class="sequence" />
                </id>
                <property name="login" column="login" type="string"/>
                <property name="password" column="password" type="string"/>
<many-to-one name="pers" class="utilisateur3.Personne" not-null="true"/>
        </class> </hibernate-mapping>
```

III.1.9 Many to many



Exemple : une personne peut être associée à plusieurs rôles dans une entreprise et un rôle peut être affecté à plusieurs personnes.

La classe Personne a un nouvel attribut, correspondant à une collection d'objets rôle, et la classe rôle a une collection d'attributs correspondant à une collection d'objets Personne. Il faut trois tables pour pouvoir mapper cette relation. Le fichier de mapping de la classe Personne s'écrit :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<!-- Strategie table-per-class hierarchy mapping -->
<hibernate-mapping package="utilisateur5">
         <class name="Personne" table="PERSONNES" discriminator-value="P">
                <id name="idPers" column="idPers" type="long">
                        <generator class="sequence"/>
        </id>
        <discriminator column="sousclasse" type="character"/>
                <property name="nom" column="nom" type="string"/>
                <property name="prenom" column="prenom" type="string"/>
        <set name="roles" table="PERSONNES_ROLES">
                        <key column="idPers" />
                <many-to-many class="utilisateur5.Role" column="idRole" />
          </set>
</hibernate-mapping>
```

La table PERSONNES_ROLES permet de faire la jointure entre la table PERSONNES et la tables ROLES. Le fichier de mapping de la classe Role s'écrit :

```
<?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="utilisateur5">
        <class name="Role" table="ROLES" >
                <id name="idRole" column="idRole" type="long">
                        <generator class="sequence" />
                </id>
                <property name="nom" column="nom" type="string"/>

                <set  name="personnes" table="PERSONNES_ROLES"
inverse="true" >
                        <key column="idRole" />
                        <many-to-many  class="utilisateur5.Personne"
column="idPers" >
                </set>
        </class>
</hibernate-mapping>
```

III.1.10 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.

```
<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 apparaît dans l'INSERT SQL?

(4) update : Est ce que la colonne mappée apparaît 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 contenant.

III.1.11 Subclass

Enfin, les requêtes polymorphiques nécessitent une déclaration explicite de chaque classe héritée de la classe racine. Pour la stratégie de mapping 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) : Paraméter 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.

III.1.12 Import

Si l'application possède deux classes persistantes avec le même nom et que l'on ne veut 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.

III.2 Types Hibernate

III.2.1 Entités et valeurs

Pour comprendre le comportement des différents objets, 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 atteinte - et correspond plus généralement à la façon d'utiliser les objets dans les grands systèmes. Les entités supportent les références partagées et circulaires.
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 de type persistance et supprimées par atteinte. Puisque les objets de type valeur sont « persistés » et effacés avec les entités qui les contiennent, ils ne peuvent pas être découpés 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.

III.2.2 Les types de valeurs basiques

Les types basiques peuvent être 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. boolean, yes_no et true_false sont des encodages possibles pour les booléen Java ou java.lang.Boolean, sa classe encapsulante.
String effectue le mapping entre java.lang.String et varchar, Date (ou time ou timestamp) effectue le mapping entre java.util.Date (et ses classes filles) et les types SQL date (ou time ou timestamp), calendar (calendar_date) effectue le mapping entre java.util.Calendar et les types SQL timestamp et date (ou équivalent),
Les types de valeurs basiques ont des constantes Type correspondant dans net.sf.hibernate.Hibernate. Par exemple, Hibernate.STRING représente le type string.

III.2.3 Type persistant d'énumération

Un type enum est un concept java classique où une classe contient un 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.

III.3 Identificateur SQL mis entre guillemets

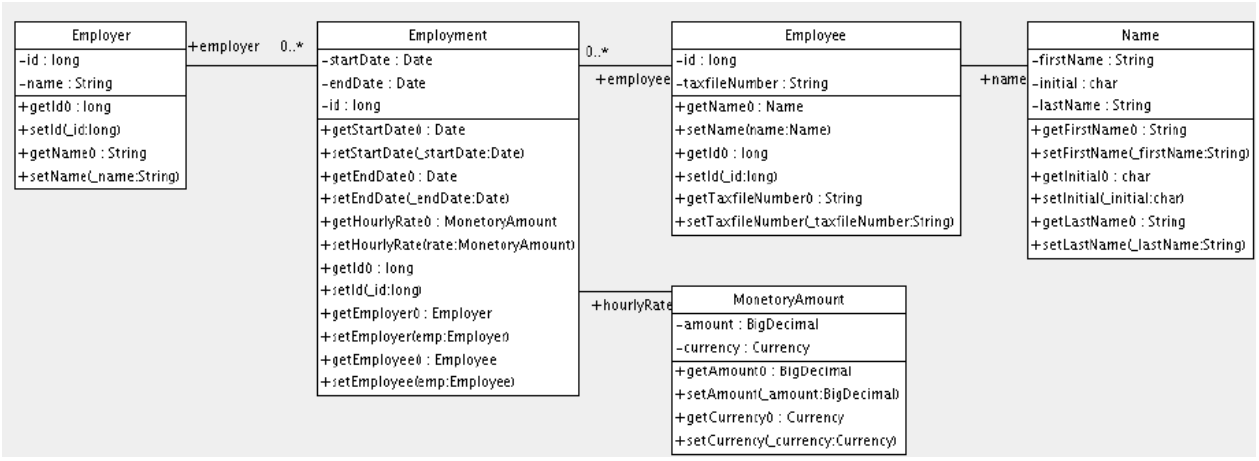
Vous pouvez forcer Hibernate à placer, dans le SQL généré, les noms des tables et des colonnes entre guillemets en incluant la table ou le nom de colonne entre guillemets simples dans le document de configuration. Hibernate utilisera la syntaxe appropriée dans le SQL généré en fonction du Dialect (généralement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

IV Exemples

IV.1 Employeurs / employés

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 :

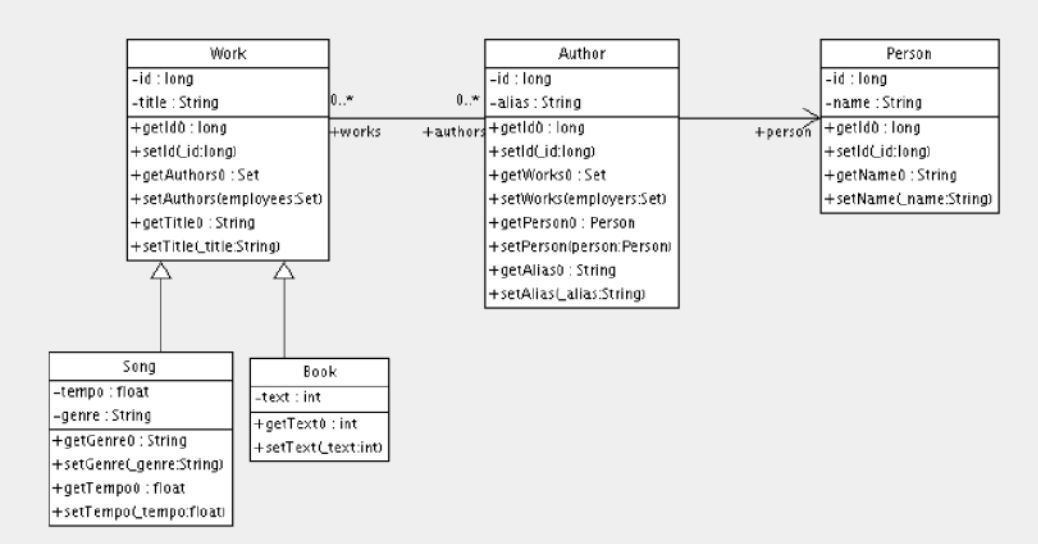
```
<hibernate-mapping>
      <class name="Employer" table="employers">
           <id name="id">
               <generator class="sequence">
                     <param name="sequence">employer id seq</param>
               </generator>
           </id>
           <property name="name"/>
      </class>
      <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
```

IV.2 Auteur / travail

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
```

CONCLUSION

Dans ce tutorial, nous avons présenté les principales fonctionnalités d'Hibernate ainsi que le moyen de le configurer et de le mettre en œuvre. Diverses applications ont été développées afin de bien vous présenter ce framework et de pouvoir voir son utilité et la structure de ces applications. Il faut garder à l'esprit qu'Hibernate en tant que couche d'accès aux données, est fortement intégré à votre application et il ne faut pas perdre de vue qu'il vous permet d'accéder à vos données. En général, toutes les autres couches dépendent du mécanisme de persistance quel qu'il soit.

Le framework est très puissant et finalement peu complexe grâce à des plugins comme Hibernate Synchroniser. Hibernate est capable de gérer finement les jointures entre les tables, la génération des clés primaires, la conversion de types. Ceci se fait simplement dans les fichiers de mapping xml. De même, Hibernate fournit un langage de requête efficace. En bref, ce type d'outil optimise le temps de développement du programmeur et permet de réaliser des applications plus homogènes, plus facilement migrables aussi. Il faut penser que pour changer de base de données, il suffit juste de toucher à hibernate.cfg.xml. Hibernate dans sa version 3 supporte la norme EJB 3, s'adapte à toutes les nouveautés, est en continuelle mise à jour et serait désigné comme LE standard de l'avenir par SUN vu les atouts qu'il propose. Dans un avenir proche, nous pouvons penser qu'il va bien évoluer et qu'il offrira des applications encore plus performantes et que les quelques inconvénients soulevés dans ce tutorial seront atténués.

En ce qui nous concerne, l'étude de ce framework a été une expérience très enrichissante, et nous espérons que nous aurons l'occasion de tester toutes ces potentialités dans un futur proche.

OUVRAGES CONSULTES

Sites Internet :

http://www.hibernate.org

http://www.alaide.com

http:// www.developpez.com

http://fr.wikipedia.org/wiki/Hibernate

Livres :

Documents remis par IBM Montpellier