Relationnel-Objet avec JPA et Hibernate

1. Introduction à Hibernate

L'objectif va être de tirer parti dans des applications distribuées de données pouvant provenir de diverses sources de données et ce, de manière facilitée pour le programmeur. Par principe, il s'agit aussi de prendre en charge les évolutions et les extensions possibles des applications.
Les solutions assurant la persistance des données au sein d'applications distribuées sont multiples :

- JDBC
- CMP (Container-Managed Persistence) EJB
- frameworks ORM : Hibernate, TopLink, OpenJPA, KODO
- Architetures Orientées Services (SOA)

Les outils ORM vont s'appuyer sur un modèle de métadonnées qui va guider la correspondance entre les classes Java et les tables de la base de données.
Dans le contexte d'Hibernate, ce modèle va être exprimé soit au travers de descripteurs XML, soit au travers d'annotations Java (depuis Java 5). Hibernate s'appuie sur la librairie standard JPA (Java Persistence Api) pour ce qui concerne la mise en correspondance au travers d'annotations. Hibernate propose une gamme de fonctionnalités suffisamment riches pour prendre par exemple, en charge la persistence de plusieurs POJOs dans une seule table.

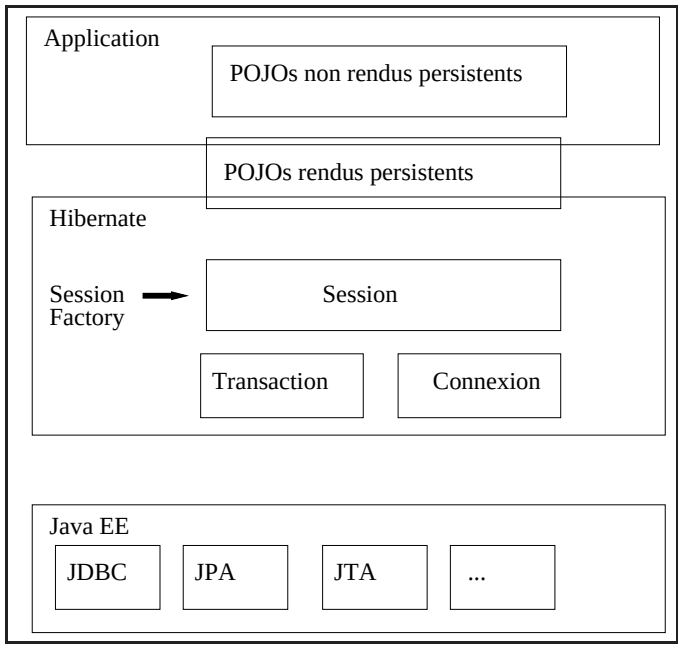

Figure 1 – Architecture Hibernate

1.1 Principe du mapping Relationnel-Objet et patron de conception DAO

La mise en correspondance du schéma relationnel de la base de données, avec la couche objets métier, peut s'envisager manuellement [1], au travers du patron de conception pour la persistance DAO (Data Access Object, cf Figure). Un ensemble de classes intermédiaires (suffixées par DAO) vont venir s'articuler entre les classes Java définies sans traitements spécifiques (POJOs [2] ou encore Data Transfer Objects (DTO)) et les tables de la base de données. Ces classes *DAO* vont prendre en charge les méthodes CRUD (Create, Retrieve, Update, Delete), en d'autres termes, de prendre en charge les opérations de consultation, d'insertion et de mise à jour permises sur les tuples des tables.

1.1.1 Limites de l'approche manuelle

Les activités liées à la gestion de la couche de persistance sont déléguées au développeur qui a sous sa responsabilité de :

— de l'écriture manuelle des requêtes, gestion des exceptions
— de la gestion manuelle des connexions et des transactions (prise en charge des pools de connexion, etc)
— de la création des entity beans (POJOs) et de la définition de leurs accesseurs
— des problèmes liés aux choix de mise en correspondance (héritage, dépendance de jointure, agrégation, ...)

1.1.2 Avantages des approches automatisables

L'utilisation d'un framework de persistance va rendre transparentes (ou presque), les activités liées au mapping :

— économie d'écriture dans le code
— gestion de la concurrence, du cache,
— mise à disposition de langages d'interrogation des objets (inspirés d'OQL)

Les notions importantes vont porter sur l'indépendance entre la couche métier et la base de données ou encore sur la persistance par accessibilité (tout objet n'est pas rendu persistant et il faut explicitement désigner les racines de persistance). La réflexion sur le DAO demeure en filigrane avec une mise en correspondance d'un schéma de base de données et d'un modèle de classes. Les approches retenues tournent autour de

1. annotations des entités bean au travers d'éléments de métadonnées standardisées (JPA et annotations Java)
2. définition de la correspondance au travers de fichiers XML (non standard et différent selon le framework retenu)
3. notion de navigabilité entre les instances du modèle objet, non retrouvée dans un modèle relationnel

L'approche s'appuyant sur les annotations est l'approche à privilégier :

1. standard et portable
2. plus concise que les fichiers XML
3. l'essentiel se situe au niveau de la phase de compilation

1. Comme vu dans les TPs précédents
2. Plain Object Java Objects

2. Hibernate en pratique avec les fichiers XML

Il s'agit ici d'une première exploitation du *framework* dit de persistance Hibernate (JEE) qui peut être vu comme une solution facilitant l'accès comme le stockage des données dans un environnement distribué. Dans ce cadre, Hibernate va réaliser un *mapping* objet-relationnel au travers d'un modèle (métadonnées) qui va guider à la fois stockage et accès.

3. illustration

Un exemple très simple portant sur une seule entité **personne** est donné. L'entité personne est traduite sous forme de relation dans le modèle relationnel et sous forme d'une classe Java descriptive sans véritables particularités qui va faciliter la prise en charge de multiples objets (mise à jour de leur état, sérialisation dans le contexte d'une application distribuée, mise en œuvre de mécanismes d'introspection) par des frameworks génériques.

personne(**identifiant**, nom, prenom, age, ville)

Les codes de la classe Java comme de création de la table vous sont donnés.

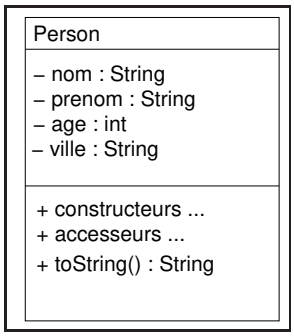

FIGURE 2 – Classe Person

3.1 Les structures et les fichiers sous-jacents à une entité

Pour chaque entité prise en charge par l'application (ici Personne), trois structures et fichiers sont à considérer : le fichier java (POJO), le fichier de description XML (person.hbm.xml) et le fichier de création de la table (.sql). Lorsqu'il existe une correspondance directe, un seul fichier suffit à créer les deux autres au travers d'outils tel que Hibernate Synchronizer ou encore SchemaExport.

3.2 Fichier de configuration hibernate.cfg.xml

hibernate.cfg.xml est un fichier de description des paramètres de connexion à la base de données et de fonctionnalités générales Hibernate. Il référence également les fichiers mettant en correspondance les structures objets et relationnelles au travers de la balise

<mapping resource="Personne.hbm.xml"/>

```
<?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>
  <!-- parametres de connexion bd : driver et url-->
  <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver
</property>
  <property name="hibernate.connection.url">jdbc:oracle:thin:@oracle.lirmm.fr:1521:the
</property>
  <!-- langage Oracle -->
  <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect
</property>
   <!-- parametres de connexion bd : infos usager -->
  <property name="hibernate.connection.username">user1</property>
  <property name="hibernate.connection.password">admbd1</property>
    <!--infos session -->
  <property name="hibernate.current_session_context_class">
org.hibernate.context.ThreadLocalSessionContext</property>
   <!--pas de cache de second niveau -->
   <property name="cache.provider_class">
org.hibernate.cache.NoCacheProvider</property>
   <!--affichage code sql console -->
<property name="show_sql">true</property>
  <!--pointe les fichiers de correspondance : autant d'elements que necessaire -->
<mapping resource="Address.hbm.xml"/>
<mapping resource="Personne.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

Listing 1 – Un tuple

3.3 Fichier décrivant le *mapping* objet-relationnel

La balise *class* possède les propriétés nécessaires à assurer la correspondance entre la classe nommée **Person** et la table nommée **personne**. La balise *id* permet de pointer explicitement l'attribut identifiant la relation et de passer outre le problème d'impédance mismatch. Les attributs caractérisant l'entité sont traités au travers de balises *property*

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

<hibernate-mapping>
   <class name="Person" table="personne">
   <id column="identifiant" name="id" type="integer">
<generator class="native"/>
</id>
<property type="string" length="20" name="nom" not-null="false"/>
<property type="string" length="20" name="prenom" not-null="false" />
<property type="string" length="20" name="ville" not-null="false" />
<property type="integer" name="age" not-null="false"/>
</class>
</hibernate-mapping>
```

Listing 2 – Un tuple

4. Exploiter Hibernate

Hibernate va donc faciliter la création, la sauvegarde, la mise à jour ou encore la suppression d'objets/données. Deux points sont particulièrement d'intérêt : le premier concerne la création d'une session de travail respectant la configuration décrite, le second porte sur les méthodes nécessaires à l'exploitation des classes/tables.

5. HQL

Hibernate intègre un langage nommé HQL proche de OQL (ODMG) qui va permettre de consulter plus finement les tables. Un exemple est donné :

```
public void readPersons(String ville) {
     Session session = factory.getCurrentSession();
     session.beginTransaction();
     List persons = session.createQuery(
           "from Person as p where p.ville = :ville").
              setString("ville", ville).
              list();
     for (Object o : persons) {
        System.out.println("liste " + o);
     }
     session.getTransaction().commit();
  }
```

Listing 3 – Un tuple

6. Hibernate en pratique avec les annotations

Une architecture synthétisant les composants mis en jeu est proposée.

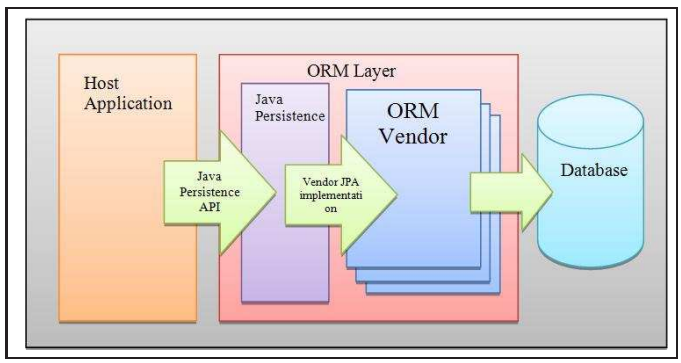

FIGURE 3 – Architecture générale tirant parti de JPA

6.1 Entité JPA

Une entité JPA est une classe Java (concrête ou abstraite) satisfaisant les propriétés suivantes :
— posséder un constructeur vide, public ou protected
— ne peut pas être une interface ou un type de données énumération (stéréotype).

Les variables d'instance de l'entité JPA sont de préférence manipulées (valuées) au travers des accesseurs définis au sein de l'entité. Les entités JPA peuvent être organisées au travers d'une hiérarchie de classes, il est alors nécessaire d'avoir un unique attribut de type identifiant (annotation Id) pour l'ensemble des classes.
La notion d'annotation disponible depuis Java 1.5 est mise à profit [3]. Les annotations permettent d'enrichir un code par des métadonnées qui peuvent être ensuite être exploitées par un code tierce pour jouer différents rôles : changer le comportement d'objets à l'exécution du programme, faciliter la document d'un programme (JavaDoc), ...
Dans notre cas, les annotations vont assurer la mise en correspondance du schéma relationnel et des classes applicatives Java. L'annotation @Entity nous indique que cette classe est une classe persistante. L'annotation @Table permet de désigner le nom de la table mise en correspondance. L'annotation @Column va permettre de caractériser une variable d'instance en particulier et possède des atributs comme name, unique ou nullable.

```
@Entity import javax.persistence.Entity;
@Table import javax.persistence.Table;
@Column import javax.persistence.Column;
@Id import javax.persistence.Id;
@GeneratedValue import javax.persistence.GeneratedValue;
@Version import javax.persistence.Version;
@OrderBy import javax.persistence.OrderBy;
@Transient import javax.persistence.Transient;
@Lob  import javax.persistence.Lob;

Pour les Mapping :

@OneToOne import javax.persistence.OneToOne;
@ManyToOne import javax.persistence.ManyToOne;
@OneToMany import javax.persistence.OneToMany;
@ManyToMany import javax.persistence.ManyToMany;
@PrimaryKeyJoinColumn import javax.persistence.PrimaryKeyJoinColumn;
@JoinColumn import javax.persistence.JoinColumn;
@JoinTable import javax.persistence.JoinTable;
@MapsId import javax.persistence.MapsId;
Hibernate Inheritance Mapping Annotations
@Inheritance import javax.persistence.Inheritance;
@DiscriminatorColumn import javax.persistence.DiscriminatorColumn;
@DiscriminatorValue import javax.persistence.DiscriminatorValue;
```

Listing 4 – Exemples d'annotations

Les annotations ont des attributs :

```
   @Entity
   @Table(name="tbl_City")
   public class City implements Serializable {
--
   @Column(name="NOM", nullable=false, length=512)
      public String getNom() { return nom; }
--
   @ManyToOne(cascade = CascadeType.ALL, targetEntity = Ville.class)
      @JoinColumn(name="LOCALISATION")
}
```

3. Les annotations sont une alternative aux fichiers de configuration en XML

Listing 5 – Exemples d'attributs

```
@Entity
public class Personne {
   // Required by JPA
   private Personne() {}

   public Personne(String numSS, String nom, Date DateN, String genre) {
      this.numSS = numSS;
      this.nom = nom;
      this.DateN = DateN;
      this.genre = genre;
   }

   public String getNom() {
      return nom;
   }
....

   @Id
   private String numSS;
   @Column(name="nom", length=15)
   private String nom;
   private Date   DateN;
   private String   genre;
}
```

Listing 6 – Portion de classe

6.2 Le gestionnaire d'entités ou EntityManager

Le gestionnaire d'entités représenté par l'interface EntityManager gère les objets @Entity qui appartiennent au contexte de persistance défini avec le pont relationnel/objet de spécifications JPA. Cinq opérations sont définies au moyen de méthodes de l'EntityManager sur une entité : PERSIST, REMOVE, REFRESH (synchronisation d'un objet persistant), DETACH (détachement d'un objet persistant du contexte transactionnel) et MERGE.

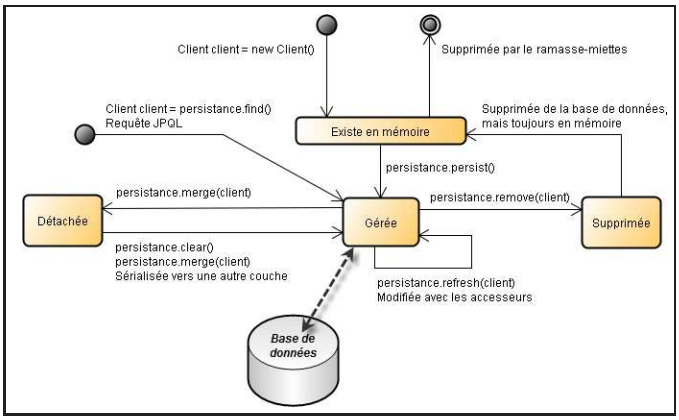

FIGURE 4 – Cycle de vie d'une "entity Bean"

```
import javax.persistence.*;
import java.util.*;

   public class AjoutePersonne {
      public static void main(String[] args) {

         EntityManagerFactory emf =
             Persistence.createEntityManagerFactory("jpa-societe-pu");
         EntityManager em = emf.createEntityManager();
         System.out.println("Entity manager prêt");

         EntityTransaction tx = em.getTransaction();
         tx.begin();
         System.out.println("Début de la transaction");

         for(String numSS : args) {

            Date dateN = new GregorianCalendar(1967, Calendar.APRIL, 3).getTime();
            Personne pier = new Personne(numSS, "Martin" ,dateN, "H");
            System.out.println(pier.toString() + " créé");


            em.persist(pier);
         }
           tx.commit();
         System.out.println("Transaction confirmée");
         em.close();
         emf.close();
      }
   }
```

Listing 7 – Une insertion

L'interface EntityManager s'appuie sur un fichier XML nommé **persistence.xml** qui permet de paramétrer les variables d'accès à la base de données

```
<persistence>
   <persistence-unit name="jpa-societe-pu">
      <!-- Définit Hibernate comme fournisseur de persistence -->
```

```
      <provider>org.hibernate.ejb.HibernatePersistence</provider>

      <properties>
         <property name="hibernate.archive.autodetection" value="class" />
         <!-- <property name="hibernate.show_sql" value="true" /> -->
         <property name="hibernate.format_sql" value="true" />

         <!-- Configuration de la BDD -->
         <property name="hibernate.connection.driver_class"
             value="oracle.jdbc.driver.OracleDriver" />
         <property name="hibernate.connection.url"
             value="jdbc:oracle:thin:@faraman:1521:xe" />
         <property name="hibernate.connection.username" value="isa" />
         <property name="hibernate.connection.password" value="isa" />

         <!-- Spécifie le 'dialecte' SQL utilisé pour communiquer avec la BDD -->
         <property name="hibernate.dialect"
             value="org.hibernate.dialect.Oracle9iDialect" />

         <!-- Indique a Hibernate de (re-)créer la BDD au lancement de l'application
             value=create-->
         <property name="hibernate.hbm2ddl.auto" value="" />
      </properties>

   </persistence-unit>
</persistence>
```

Listing 8 – persistence.xml

6.3 Le langage de requête

JPQL (Java Persistence Query Language) s'inspire de SQL3, et permet d'interroger une base de données à partir des entités JPA définies dans le contexte de persistance. JPQL permet de travailler sur de la consultation, de la mise à jour et de la suppression sur les données (objets versus tuples) du système.

```
import javax.persistence.*;
import java.util.*;

public class ListePersonnnes {
   public static void main(String[] args) {

      EntityManagerFactory emf =
          Persistence.createEntityManagerFactory("jpa-societe-pu");
      EntityManager em = emf.createEntityManager();
      System.out.println("Entity manager prêt");

      EntityTransaction tx = em.getTransaction();
      tx.begin();
      System.out.println("Début de la transaction");

      Query query = em.createQuery("SELECT p FROM Personne p ORDER BY dateN DESC");
      System.out.println("Les personnes");
      @SuppressWarnings("unchecked")
      List<Personne> personnes = (List<Personne>)query.getResultList();
      for(Personne p : personnes) {
         System.out.println(p);
      }
```

```
        tx.commit();
        System.out.println("Transaction confirmée");
        em.close();
        emf.close();
    }
}
```

Listing 9 – Un tuple

7. Exercices

7.1 Exercice 1

Le schéma relationnel exploité dans les précédents TPs à savoir Monument/Lieu/Département/Célébrité est repris et vous construirez un nouveau projet Hibernate Annotations qui aura les mêmes objectifs que le projet Java sur le mapping manuel, à savoir tirer parti au sein d'une couche métier du contenu d'une base de données Oracle. Vous construirez des classes applicatives qui permettent de retourner ou de mettre à jour les monuments ou encore les lieux dans lesquels ils sont localisés (aidez vous des exemples sur les personnes et les villes qui vous sont fournis)

7.2 Exercice 2

Vous exploiterez le langage JPQL pour :
- retourner les monuments qui appartiennent à l'état
- retourner les monuments qui sont de type château
- retourner les monuments qui sont localisés dans un lieu dont le code est passé en paramètre
- retourner le nombre de monuments par lieu
- retourner le nombre de monuments par type de monument

Vous pouvez définir des classes utilitaires pour gérer tout ou une partie des traitements métiers demandés.