Support Formation JAVA JPA Hibernate

Pour	Formation
Date	12/11/2024
Objet	Support Formation JAVA JPA Hibernate.

I) Glossaire

- **API** : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédié pour implémenter une fonctionnalité donnée.

- **ORM** : Object-Relational Mapping (**MOR** : Mapping Objet-Relationnel en français) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.

- **JPA** : Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java.

- **JPQL** : Le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. La grammaire de ce langage est définie par la spécification J.P.A.

- **HQL** : **Hibernate Query Language** est aussi un langage de requête orienté objet au même titre que JPQL. La principale différence avec le langage JQL est que le en HQL le « **Select** » sur l'objet n'est pas nécessaire. En fin de compte pour le JPQL on aura : **Select person from Personne person** alors que pour le HQL on aura : **from Personne**.

- **Bean** : le « **Bean** » (ou haricot en français) est une technologie de composants logiciels écrits en langage Java. Les **Beans** sont utilisés pour encapsuler plusieurs objets dans un seul objet. Le « **Bean** » regroupe alors tous les attributs des objets encapsulés. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.

- **EJB** : **Enterprise Java Beans** (EJB) est une architecture de composants logiciels côté serveur pour la plateforme de développement Java EE. Cette architecture propose un cadre pour créer des composants distribués (c'est-à-dire déployés sur des serveurs distants) écrit en langage de programmation Java hébergés au sein d'un serveur applicatif permettant de représenter des données (EJB dit entité), de proposer des services avec ou sans conservation d'état entre les appels (EJB dit session), ou encore d'accomplir des tâches de manière asynchrone (EJB dit message). Tous les EJB peuvent évoluer dans un contexte transactionnel ce qui peut permettre de gérer les transactions avec les sources de données (fichier Xml, fichier Csv, fichier Json, base de données etc.…).

- **POJO** : POJO est un acronyme qui signifie Plain Old Java Object que l'on peut traduire en français par bon vieil objet Java. Cet acronyme est principalement utilisé pour faire référence à la simplicité d'utilisation d'un objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB. Ainsi, un POJO n'implémente pas d'interface spécifique à un Framework comme c'est le cas par exemple pour un composant EJB.

- **POJI** : POJI est un acronyme qui signifie Plain Old Java Interfaces que l'on peut traduire en français par bon vieil Interface Java correspond à une interface standard Java. Ils sont habituellement utilisés dans le contexte JEE pour fournir des services.

- **Servlet** : Une servlet est une classe Java qui permet de créer dynamiquement des données au sein d'un serveur HTTP. Ces données sont le plus généralement présentées au format HTML, mais elles peuvent également l'être au format XML ou tout autre format destiné aux navigateurs web. Les servlets utilisent l'API Java Servlet (package **javax.servlet**). Une servlet s'exécute dynamiquement sur le serveur web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données, transactions d'e-commerce, etc. Une servlet peut être chargé automatiquement lors du démarrage du serveur web ou lors de la première requête du client, une fois chargés, les servlets restent actifs dans l'attente d'autres requêtes du client.

- **Filtre de servlet** : Un filtre HTTP de servlet est un composant d'une application web qui agit comme un intercepteur sur une servlet. Un filtre est une classe Java qui implémente l'interface javax.servlet.Filter. Il est déclaré dans le descripteur de l'application web.xml, et posé sur une ou plusieurs servlets. Lorsqu'une requête HTTP doit être traitée par une servlet sur laquelle un filtre est appliqué, alors le serveur va exécuter la méthode **doFilter** du Filtre avant d'exécuter la méthode **doGet** ou **doPost** de la servlet.

- **Pattern IoC** : L'inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les Frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du Framework ou de la couche logicielle sous-jacente. En effet selon un problème, il existe différentes formes, ou représentation d'IoC, le plus connu étant l'injection de dépendances (dependency injection) qui est un patron de conception permettant, en programmation orientée objet, de découpler les dépendances entre objets.

- **Pattern AOP** : L'AOP (Aspect Oriented Programming) ou POA (Programmation Orientée Aspect) est un paradigme de programmation ayant pour but de compléter la programmation orientée objet et permettre d'implémenter de façon plus propre les problématiques transverses à l'application. En effet, elle permet de factoriser du code dans des greffons et de les injecter en divers endroits sans pour autant modifier le code source des endroits en question.

- **JNDI** : JNDI signifie Java Naming and Directory Interface, cette API permet d'accéder à différents services de nommage ou de répertoire de façon uniforme, d'organiser et rechercher des informations ou des objets par nommage (java naming and directory interface), de faire des opérations sur des annuaires (java naming and directory interface) tels que : LDAP : un annuaire léger, X500 : normes d'annuaires lourdes à mettre en œuvre, NIS : annuaire obsolète.

- **Pool de connexions** : Un pool de connexions est un mécanisme permettant de réutiliser les connexions créées. En effet, la création systématique de nouvelles instances de Connection peut parfois devenir très lourd en consommation de ressources. Pour éviter cela, un pool de connexions ne ferme pas les connexions lors de l'appel à la méthode close(). Au lieu de fermer directement la connexion, celle-ci est « retournée » au pool et peut être utilisée ultérieurement.La gestion du pool se fait en général de manière transparente pour l'utilisateur

II) La Réflexivité

La réflexivité ou encore l'introspection consiste à découvrir de manière dynamique des informations propres à une classe Java ou à un objet. Ce mécanisme est notamment utilisé au niveau de la machine virtuelle Java lors de l'exécution de votre programme.

Le paquetage de l'API qui gère la réflexivité est « java.lang.reflect ».
La réflexivité permet notamment l'introspection et rend possible l'accès aux classes, à leurs champs, méthodes, constructeurs et à toutes les informations les caractérisant, même celles qu'on pensait inaccessibles. Elle est également très utile pour instancier des classes de manière dynamique dans le processus de sérialisation d'un Bean Java. Elle est aussi utilisée dans la génération de code (exemple : ORM tel qu'Hibernate).

Avec la réflexivité il est possible :

1) D'identifier les classes parents d'une classe donnée

C'est-à-dire connaître l'ensemble des classes dont hérite une classe. Considérons l'exemple précédemment de la classe « Vehicule » cf. « Les Objets Java » nous avions la classe « Voiture » qui hérite de la classe « Vehicule » et une classe « Voiture4X4 » qui hérite de « Voiture ».

Affichons dans le programme ci-dessous tous les parents de la classe « Voiture4X4 » :

```
24
25      public static void displaySuperclass() {
26          String methodName = "displaySuperclass";
27          Class theClass = null;
28          try {
29              theClass = Class.forName("javaapplicationreflect.Voiture4X4");
30          } catch (ClassNotFoundException ex) {
31              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
32          }
33          while ((theClass = theClass.getSuperclass()) != null) {
34              System.out.println("Nom de la classe Mère: " + theClass.getSimpleName());
35              System.out.println("Nom Complet de la classe Mère: " + theClass.getName()+"\n");
36          }
37      }
```

On obtient donc à l'affichage :

```
Output - JavaApplicationReflect (run) × | Search Results | Test Results
run:
Nom de la classe Mère: Voiture
Nom Complet de la classe Mère: javaapplicationreflect.Voiture

Nom de la classe Mère: Vehicule
Nom Complet de la classe Mère: javaapplicationreflect.Vehicule

Nom de la classe Mère: Object
Nom Complet de la classe Mère: java.lang.Object

BUILD SUCCESSFUL (total time: 0 seconds)
```

L'affichage nous montre bien que la classe « Voiture4X4 » hérite des classes « Voiture », « Vehicule » et bien entendu la classe « Object » qui est par définition la mère de toute les classes Java.

2) Connaitre toutes les interfaces implémentées par une classe donnée

Considérons l'exemple d'une interface « IAnimale » avec sa classe d'implémentation « Animal ».
« Animal » :

```
13  public class Animal implements IAnimal, Serializable {
14
15      public String nom;
16      public double poids;
17      public String couleur;
18
19      @Override
20      public void crie() {
21          System.out.println("L'animal crie");
22      }
23
24      @Override
25      public void marcher() {
26          System.out.println("L'animal marche");
27      }
28
29      public String getNom() {
30          return nom;
31      }
32
33      public void setNom(String nom) {
34          this.nom = nom;
35      }
36
37      public double getPoids() {
38          return poids;
39      }
40
41      public void setPoids(double poids) {
42          this.poids = poids;
43      }
44
45      public String getCouleur() {
46          return couleur;
47      }
48
49      public void setCouleur(String couleur) {
50          this.couleur = couleur;
51      }
52  }
53
```

« IAnimale » :

```
10  */
11  public interface IAnimal {
12
13      void crie();
14
15      void marcher();
16  }
17
18
```

Considérons le programme Java ci-dessous :

```
39
40      public static void displayInterfaces() {
41          String methodName = "displayInterfaces";
            Class theClass = null;
43          try {
44              theClass = Class.forName("javaapplicationreflect.Animal");
45              Class[] interfaces = theClass.getInterfaces();
46              for (Class theInterface : interfaces) {
47                  System.out.println("Nom de l'interface : " + theInterface.getSimpleName());
48                  System.out.println("Nom complet de l'interface : " + theInterface.getName() + "\n");
49              }
50          } catch (ClassNotFoundException ex) {
51              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
52          }
53      }
54
```

On obtient à l'affichage :

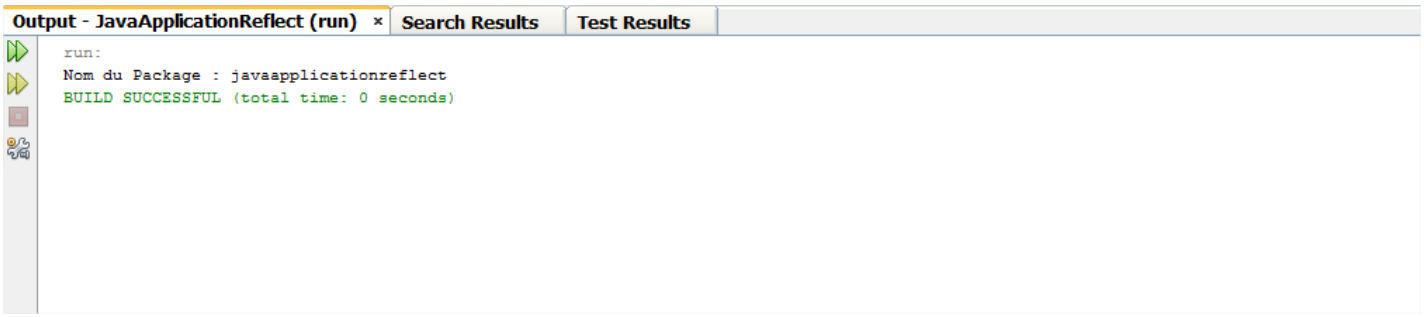
```
Output - JavaApplicationReflect (run) × | Search Results | Test Results
run:
Nom de l'interface : IAnimal
Nom complet de l'interface : javaapplicationreflect.IAnimal

Nom de l'interface : Serializable
Nom complet de l'interface : java.io.Serializable

BUILD SUCCESSFUL (total time: 0 seconds)
```

3) Connaitre le nom du paquetage d'une classe donnée

Toujours avec la classe « Animal » nous pouvons récupérer son nom de paquetage.

```
61
62      public static void displayPackage() {
63          String methodName = "displayPackage";
            Class theClass = null;
65          try {
66              theClass = Class.forName("javaapplicationreflect.Animal");
67              Package pack = theClass.getPackage();
68              System.out.println("Nom du Package : " + pack.getName());
69          } catch (ClassNotFoundException ex) {
70              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
71          }
72      }
73
```

On obtient à l'affichage :



4) Récupérer la liste des méthodes d'une classe.

Nous pouvons récupérer la liste des méthodes de la classe la classe « Voiture4X4 ».

```
76
77      public static void displayMethods() {
78          String methodName = "displayMethods";
            Class theClass = null;
80          try {
81              theClass = Class.forName("javaapplicationreflect.Voiture4X4");
82              Method[] methodes = theClass.getMethods();
83              for (Method methode : methodes) {
84                  System.out.println("Nom de methode : " + methode.getName());
85              }
86
87              Class[] argument = new Class[]{double.class};
88
89              Method method = theClass.getMethod("setPuissanceMoteurArriere", argument);
90              System.out.println("Signature methode setPuissanceMoteurArriere : " + method);
91
92          } catch (ClassNotFoundException | SecurityException | NoSuchMethodException ex) {
93              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
94          }
95      }
```

Ligne 83 : on récupère la liste des méthodes de la classe « Voiture4X4 ».
Ligne 89 : on récupère la signature de la méthode « setPuissanceMoteurArriere » en lui passant comme argument « new Class [] {double.class} » car, en effet, la méthode « setPuissanceMoteurArriere » a comme signature dans la classe « Voiture4X4 » :

```
public void setPuissanceMoteurArriere(double puissanceMoteurArriere) {
    this.puissanceMoteurArriere = puissanceMoteurArriere;
}
```

5) Récupérer la liste des attributs d'une classe.

Il est aussi possible de récupérer la liste des attributs d'une classe.

```
98
99      public static void displayAllFields() {
100         String methodName = "displayAllFields";
            Class theClass = null;
102         try {
103             theClass = Class.forName("javaapplicationreflect.Vehicule");
104             java.lang.reflect.Field[] fields = theClass.getDeclaredFields();
105             for (Field field : fields) {
106                 System.out.println("Nom du champ : " + field);
107             }
108         } catch (ClassNotFoundException | SecurityException ex) {
109             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
110         }
111     }
```

```
Output - JavaApplicationReflect (run) × | Search Results | Test Results
run:
Nom du champ : protected java.lang.String javaapplicationreflect.Vehicule.marque
Nom du champ : protected java.lang.String javaapplicationreflect.Vehicule.couleur
Nom du champ : protected int javaapplicationreflect.Vehicule.annee
Nom du champ : protected double javaapplicationreflect.Vehicule.taille
Nom du champ : protected boolean javaapplicationreflect.Vehicule.estAssure
BUILD SUCCESSFUL (total time: 0 seconds)
```

6) Récupérer la liste des constructeurs d'une classe.

On aussi récupérer la liste de constructeurs.

```
131
132     public static void displayConstructors() {
133         String methodName = "displayConstructors";
            Class theClass = null;
135         try {
136             theClass = Class.forName("javaapplicationreflect.Vehicule");
137             Constructor[] constructors = theClass.getConstructors();
138             for (Constructor constructor : constructors) {
139                 System.out.println("constructor : " + constructor);
140             }
141             Class[] arguments = new Class[]{String.class, String.class};
142             Constructor constructor = theClass.getConstructor(arguments);
143             System.out.println("constructor String String : " + constructor);
144         } catch (ClassNotFoundException | SecurityException | NoSuchMethodException ex) {
145             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
146         }
147     }
148
```

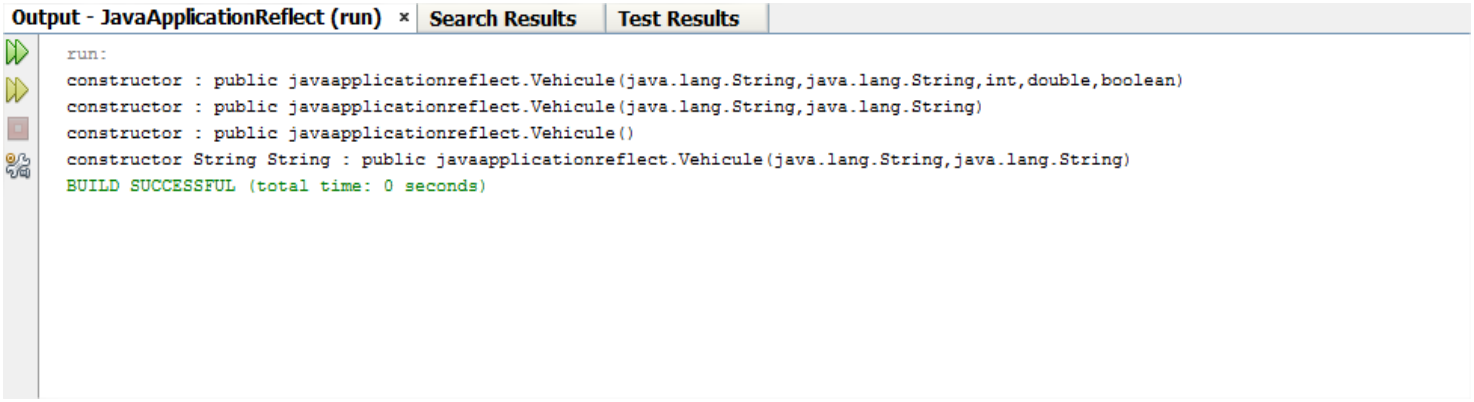
Ligne 137 : On récupère la liste des constructeurs de la classe « Vehicule ».

Ligne 89 : on récupère la signature de constructeur « Vehicule(String marque, String couleur) » en lui passant comme argument « new Class [] {String.class, String.class } ».

Nous rappelons que le code du constructeur est le suivant :

```
public Vehicule(String marque, String couleur) {
      this.marque = marque;
      this.couleur = couleur;
      System.out.println("Contructeur Vehicule(string,string)");
}
```

On obtient le résultat suivant :



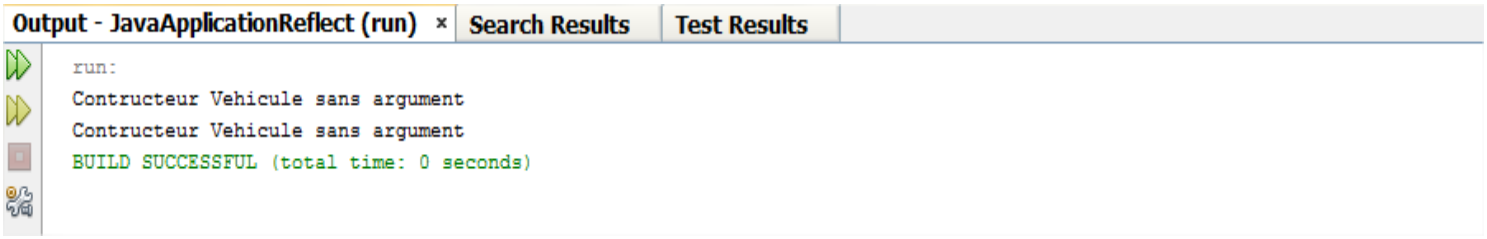
7) Instancier une classe de manière dynamique.

Dans l'exemple ci-dessous, nous allons fabriquer une instance de « Vehicule » de manière inhabituelle en utilisant la retro inspection.

```
150
151     public static void instanciateVehicule() {
152         String methodName = "instanciateVehicule";
153         try {
154             Vehicule renault = new Vehicule();
155             // equivalent à
156             Class theClass = Class.forName("javaapplicationreflect.Vehicule");
157             Object objectCitroen = theClass.newInstance();
158         } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
159             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
160         }
161     }
```

Ligne 154 et 157 : ces deux lignes sont totalement équivalentes.

D'ailleurs même la console affiche :



Ce qui correspont exactement à l'instanciation de deux objects de type « Vehicule ».

On peut caster « objectCitroen » en « Vehicule ».

```
150
151     public static void instanciateVehicule() {
152         String methodName = "instanciateVehicule";
153         try {
154             Vehicule citroen = null;
155             Vehicule renault = new Vehicule();
156             // equivalent à
157             Class theClass = Class.forName("javaapplicationreflect.Vehicule");
158             Object objectCitroen = theClass.newInstance();
159             if (objectCitroen instanceof Vehicule) {
160                 citroen = (Vehicule) objectCitroen;
161             }
162         } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
163             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
164         }
165     }
```

III) Généralités sur les ORM

ORM signifie Object-Relational Mapping (**MOR** : Mapping Objet-Relationnel en français). C'est est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.
La spécification JPA a été implémentée par divers produits (**Hibernate**, **Toplink**, **EclipseLink**, **OpenJpa** etc.…).

Le standard JPA utilise le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. La grammaire de ce langage est définie par la spécification J.P.A.

Nous allons voir dans le cadre de ce cours l'implémentation **Hibernate** de la spécification J.P.A.

IV) Implémentation JPA/Hibernate

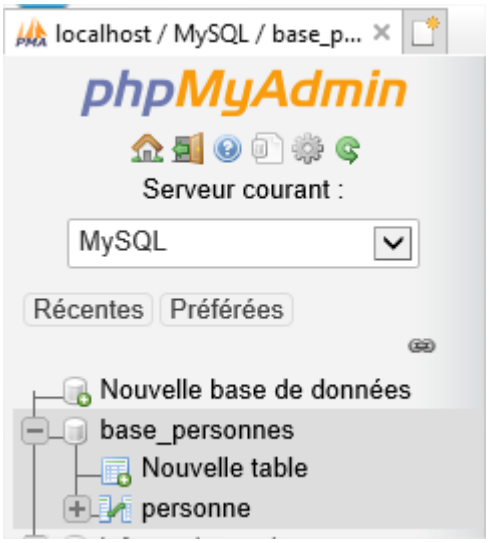
Créer le projet Maven « **maven-dao-personnes-jpa-hibernate** » qui sera notre projet de test de l'ORM **JPA/Hibernate**.

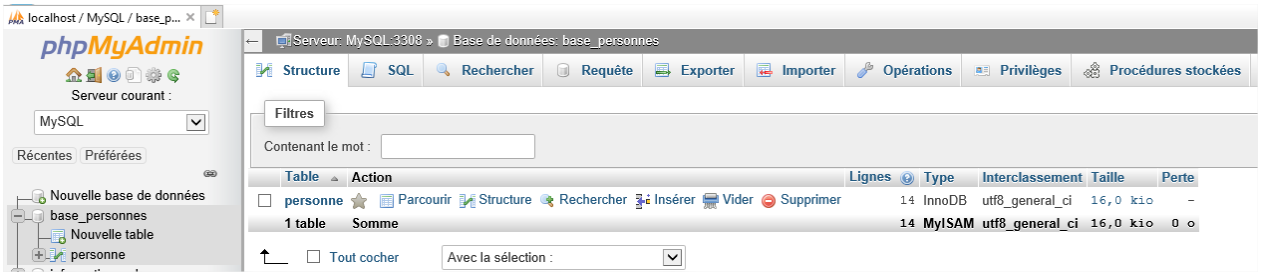
1. *Création de la base de données « base_personnes »*

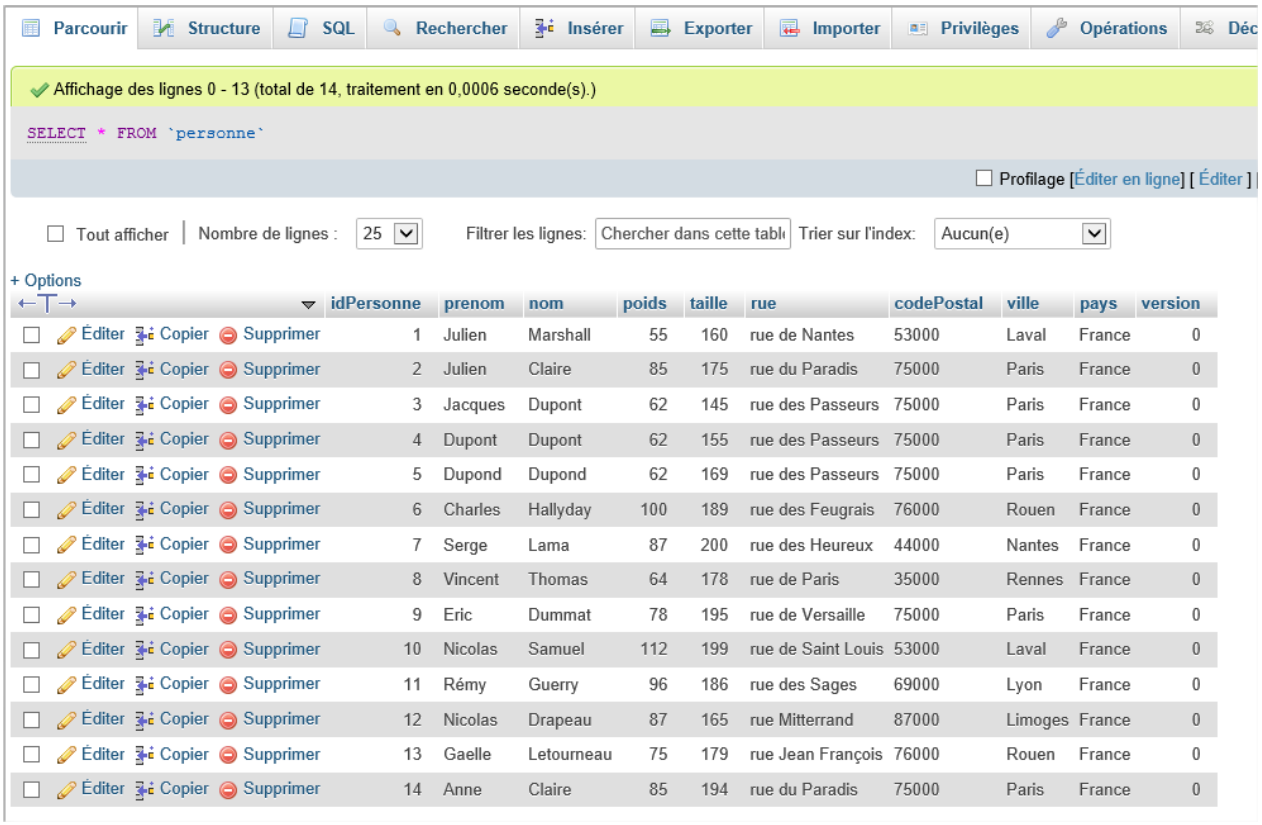
Créer la base MySQL avec le nom [**base_personnes**] avec l'outil de votre choix. La base sera par la suite la propriété de l'utilisateur « **application** » et du mot de passe « **passw0rd** ».

Jouer le script **script_base_personnes.sql** qui se trouve à la racine de votre projet.

Avec phpMyAdmin on obtient :







Parcourir | Structure | SQL | Rechercher | Insérer | Exporter | Importer | Privilèges | Opérations | Déc

Affichage des lignes 0 - 13 (total de 14, traitement en 0,0006 seconde(s).)

```
SELECT * FROM `personne`
```

Profilage [Éditer en ligne] [Éditer]

Tout afficher | Nombre de lignes : 25 Filtrer les lignes: Chercher dans cette tabl Trier sur l'index: Aucun(e)

+ Options

	idPersonne	prenom	nom	poids	taille	rue	codePostal	ville	pays	version
Éditer Copier Supprimer	1	Julien	Marshall	55	160	rue de Nantes	53000	Laval	France	0
Éditer Copier Supprimer	2	Julien	Claire	85	175	rue du Paradis	75000	Paris	France	0
Éditer Copier Supprimer	3	Jacques	Dupont	62	145	rue des Passeurs	75000	Paris	France	0
Éditer Copier Supprimer	4	Dupont	Dupont	62	155	rue des Passeurs	75000	Paris	France	0
Éditer Copier Supprimer	5	Dupond	Dupond	62	169	rue des Passeurs	75000	Paris	France	0
Éditer Copier Supprimer	6	Charles	Hallyday	100	189	rue des Feugrais	76000	Rouen	France	0
Éditer Copier Supprimer	7	Serge	Lama	87	200	rue des Heureux	44000	Nantes	France	0
Éditer Copier Supprimer	8	Vincent	Thomas	64	178	rue de Paris	35000	Rennes	France	0
Éditer Copier Supprimer	9	Eric	Dummat	78	195	rue de Versaille	75000	Paris	France	0
Éditer Copier Supprimer	10	Nicolas	Samuel	112	199	rue de Saint Louis	53000	Laval	France	0
Éditer Copier Supprimer	11	Rémy	Guerry	96	186	rue des Sages	69000	Lyon	France	0
Éditer Copier Supprimer	12	Nicolas	Drapeau	87	165	rue Mitterrand	87000	Limoges	France	0
Éditer Copier Supprimer	13	Gaelle	Letourneau	75	179	rue Jean François	76000	Rouen	France	0
Éditer Copier Supprimer	14	Anne	Claire	85	194	rue du Paradis	75000	Paris	France	0

2. *Création du fichier persistance.xml*

Créer le fichier **persistence.xml** dans le dossier **maven-personnes-dao-jpa-hibernate/src/main/resources/META-INF**.
Son contenu sera :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:/
    <persistence-unit name="PersonnesPU" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.cours.dao.entities.Personne</class>
        <properties>
            <!--MYSQL 5 : javax.persistence.jdbc.url : jdbc:mysql://localhost:3306/base_personnes?serverTimezone=UTC avec eventuellement useSSL=false -->
            <!--MYSQL 8 : javax.persistence.jdbc.url : jdbc:mysql://localhost:3308/base_personnes?serverTimezone=UTC -->
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3308/base_personnes?serverTimezone=UTC"/>
            <property name="javax.persistence.jdbc.user" value="application"/>
            <property name="javax.persistence.jdbc.password" value="passw0rd"/>
            <!--MYSQL 5 : javax.persistence.jdbc.driver : com.mysql.jdbc.Driver -->
            <!--MYSQL 8 : javax.persistence.jdbc.driver : com.mysql.cj.jdbc.Driver -->
            <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.use_sql_comments" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

En version copiable :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="PersonnesPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>com.cours.dao.entities.Personne</class>
    <properties>
      <!--
MYSQL 5 : javax.persistence.jdbc.url : jdbc:mysql://localhost:3306/base_personnes?serverTimezone=UTC avec eventuellement useSSL=false -->
      <!--
MYSQL 8 : javax.persistence.jdbc.url : jdbc:mysql://localhost:3308/base_personnes?serverTimezone=UTC -->
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3308/base_personnes?serverTimezone=UTC"/>
      <property name="javax.persistence.jdbc.user" value="application"/>
      <property name="javax.persistence.jdbc.password" value="passw0rd"/>
      <!--MYSQL 5 : javax.persistence.jdbc.driver : com.mysql.jdbc.Driver -->
      <!--MYSQL 8 : javax.persistence.jdbc.driver : com.mysql.cj.jdbc.Driver -->
      <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.use_sql_comments" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Ligne 2 : la balise racine du fichier XML est **persistence**.
Ligne 3 : **persistence-unit** sert à définir une unité de persistance. Il peut y avoir plusieurs unités de persistance. Chacune d'elles a un nom (attribut name) et un type de transactions (attribut transaction-type). L'application aura accès à l'unité de persistance via le nom de celle-ci, ici JPA. Le type de transaction **RESOURCE_LOCAL** indique que l'application gère elle-même les transactions avec le SGBD. Ce sera le cas ici. Lorsque l'application s'exécute dans un conteneur **EJB3 (**Entreprise Java Bean**)**, elle peut utiliser le service de transactions de celui-ci. Dans ce cas, on mettra transaction-type=**JTA** (Java Transaction API).
JTA est la valeur par défaut lorsque l'attribut transaction-type est absent.

Ligne 4 : la balise **provider** sert à définir une classe implémentant l'interface [**org.hibernate.jpa.HibernatePersistenceProvider**], interface qui permet à l'application d'initialiser la couche de persistance. Parce qu'on utilise une implémentation JPA/Hibernate, la classe utilisée ici est une classe d'Hibernate.

Ligne 5 : la balise **properties** introduit des propriétés propres au provider particulier choisi. Ainsi selon qu'on a choisi **Hibernate, EclipseLink, Toplink, Kodo**, ... on aura des propriétés différentes. Celles qui suivent sont propres à **Hibernate**.
Ligne 9 : l'url de la base de données utilisée
Ligne 10 : l'utilisateur de la connexion au SGBD.
Ligne 11 : le mot de passe de la connexion au SGBD.
Ligne 14 : la classe du pilote JDBC du SGBD, ici MySQL.

3. *La classe JPA Personne*

La classe entité « Personne » est la classe image de la table **Personne** de la base « **base_personnes** ». Il est possible de le générer avec les IDE. Pour ma par je l'ai généré avec **NetBeans 8.0.2** mais il est possible de le générer avec Eclipse avec un outil qui s'appelle HibernateTool. Dans tous les cas ce qui va nous intéréssé ici c'est son contenu.

La classe **Personne** générée sera :

```java
package com.cours.dao.entities;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author elhad
 */
@Entity
@Table(name = "Personne")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Personne.findAll", query = "SELECT p FROM Personne p"),
    @NamedQuery(name = "Personne.findByIdPersonne", query = "SELECT p FROM Personne p WHERE p.idPersonne = :idPersonne"),
    @NamedQuery(name = "Personne.findByPrenom", query = "SELECT p FROM Personne p WHERE p.prenom = :prenom"),
    @NamedQuery(name = "Personne.findByNom", query = "SELECT p FROM Personne p WHERE p.nom = :nom"),
    @NamedQuery(name = "Personne.findByPoids", query = "SELECT p FROM Personne p WHERE p.poids = :poids"),
    @NamedQuery(name = "Personne.findByTaille", query = "SELECT p FROM Personne p WHERE p.taille = :taille"),
    @NamedQuery(name = "Personne.findByRue", query = "SELECT p FROM Personne p WHERE p.rue = :rue"),
    @NamedQuery(name = "Personne.findByCodePostal", query = "SELECT p FROM Personne p WHERE p.codePostal = :codePostal"),
    @NamedQuery(name = "Personne.findByVille", query = "SELECT p FROM Personne p WHERE p.ville = :ville"),
    @NamedQuery(name = "Personne.findByPays", query = "SELECT p FROM Personne p WHERE p.pays = :pays"),
    @NamedQuery(name = "Personne.findByVersion", query = "SELECT p FROM Personne p WHERE p.version = :version")})
public class Personne implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;
    @Column(name = "prenom")
    private String prenom;
    @Column(name = "nom")
    private String nom;
    // @Max(value=?)  @Min(value=?)//if you know range of your decimal fields consider using these annotations to enforce field validation
    @Column(name = "poids")
    private Double poids;
    @Column(name = "taille")
    private Double taille;
    @Column(name = "rue")
    private String rue;
    @Column(name = "codePostal")
    private String codePostal;
    @Column(name = "ville")
    private String ville;
    @Column(name = "pays")
    private String pays;
    @Column(name = "version")
    private Integer version;

    public Personne() {
    }

    public Personne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }

    public Integer getIdPersonne() {
        return idPersonne;
    }
```

```java
    public void setIdPersonne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public Double getPoids() {
        return poids;
    }

    public void setPoids(Double poids) {
        this.poids = poids;
    }

    public Double getTaille() {
        return taille;
    }

    public void setTaille(Double taille) {
        this.taille = taille;
    }

    public String getRue() {
        return rue;
    }

    public void setRue(String rue) {
        this.rue = rue;
    }

    public String getCodePostal() {
        return codePostal;
    }

    public void setCodePostal(String codePostal) {
        this.codePostal = codePostal;
    }

    public String getVille() {
        return ville;
    }

    public void setVille(String ville) {
        this.ville = ville;
    }

    public String getPays() {
        return pays;
    }

    public void setPays(String pays) {
        this.pays = pays;
    }

    public Integer getVersion() {
        return version;
    }

    public void setVersion(Integer version) {
        this.version = version;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (idPersonne != null ? idPersonne.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Personne)) {
            return false;
        }
        Personne other = (Personne) object;
        if ((this.idPersonne == null && other.idPersonne != null) || (this.idPersonne != null && !this.idPersonne.equals(other.idPersonne))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "com.cours.dao.entities.Personne[ idPersonne=" + idPersonne + " ]";
    }
}
```

En version copiable :

```
package com.cours.dao.entities;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author elhad
 */
@Entity
@Table(name = "Personne")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Personne.findAll", query = "SELECT p FROM Personne p"),
    @NamedQuery(name = "Personne.findByIdPersonne", query = "SELECT p FROM Personne p WHERE
p.idPersonne = :idPersonne"),
    @NamedQuery(name = "Personne.findByPrenom", query = "SELECT p FROM Personne p WHERE p.prenom =
:prenom"),
    @NamedQuery(name = "Personne.findByNom", query = "SELECT p FROM Personne p WHERE p.nom =
:nom"),
    @NamedQuery(name = "Personne.findByPoids", query = "SELECT p FROM Personne p WHERE p.poids =
:poids"),
    @NamedQuery(name = "Personne.findByTaille", query = "SELECT p FROM Personne p WHERE p.taille =
:taille"),
    @NamedQuery(name = "Personne.findByRue", query = "SELECT p FROM Personne p WHERE p.rue = :rue"),
    @NamedQuery(name = "Personne.findByCodePostal", query = "SELECT p FROM Personne p WHERE
p.codePostal = :codePostal"),
    @NamedQuery(name = "Personne.findByVille", query = "SELECT p FROM Personne p WHERE p.ville =
:ville"),
    @NamedQuery(name = "Personne.findByPays", query = "SELECT p FROM Personne p WHERE p.pays =
:pays"),
    @NamedQuery(name = "Personne.findByVersion", query = "SELECT p FROM Personne p WHERE p.version =
:version")})
public class Personne implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;
    @Column(name = "prenom")
    private String prenom;
    @Column(name = "nom")
    private String nom;
    // @Max(value=?)  @Min(value=?)//if you know range of your decimal fields consider using these annotations to enforce field validation
    @Column(name = "poids")
    private Double poids;
    @Column(name = "taille")
    private Double taille;
    @Column(name = "rue")
    private String rue;
    @Column(name = "codePostal")
    private String codePostal;
    @Column(name = "ville")
    private String ville;
    @Column(name = "pays")
    private String pays;
    @Column(name = "version")
    private Integer version;

    public Personne() {
    }

    public Personne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }

    public Integer getIdPersonne() {
        return idPersonne;
    }

    public void setIdPersonne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public String getNom() {
        return nom;
```

```
}

public void setNom(String nom) {
    this.nom = nom;
}

public Double getPoids() {
    return poids;
}

public void setPoids(Double poids) {
    this.poids = poids;
}

public Double getTaille() {
    return taille;
}

public void setTaille(Double taille) {
    this.taille = taille;
}

public String getRue() {
    return rue;
}

public void setRue(String rue) {
    this.rue = rue;
}

public String getCodePostal() {
    return codePostal;
}

public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
}

public String getVille() {
    return ville;
}

public void setVille(String ville) {
    this.ville = ville;
}

public String getPays() {
    return pays;
}
```

```
    public void setPays(String pays) {
        this.pays = pays;
    }

    public Integer getVersion() {
        return version;
    }

    public void setVersion(Integer version) {
        this.version = version;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (idPersonne != null ? idPersonne.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Personne)) {
            return false;
        }
        Personne other = (Personne) object;
        if ((this.idPersonne == null && other.idPersonne != null) || (this.idPersonne != null &&
!this.idPersonne.equals(other.idPersonne))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "com.cours.dao.entities.Personne[ idPersonne=" + idPersonne + " ]";
    }
}
```

Ajouter l'annotation @Version en dessous de @Column(name = "Version"). Cette annotation va servir à gérer les accés concurents dans l'application.

Pour faciliter le débogage la signature de la methode toString sera :

```
    @Override
  public String toString() {
    return String.format("[idPersonne=%s, prenom=%s, nom=%s, poids=%s, taille=%s, rue=%s, codePostal=%s , ville=%s,  pays=%s,
version=%s]", idPersonne, prenom, nom, poids, taille, rue, codePostal, ville, pays, version);
  }
```

Ajouter aussi le constructeur :

```
public Personne(String prenom, String nom, double poids, double taille, String rue, String ville, String codePostal, String pays) {
    this.prenom = prenom;
    this.nom = nom;
    this.poids = poids;
    this.taille = taille;
    this.rue = rue;
    this.ville = ville;
    this.codePostal = codePostal;
    this.pays = pays;
}
```

Nous allons maintenant expliquer les lignes importantes de la classe **Personne**.

Ligne 26 : l'annotation **@Entity** est la première annotation indispensable. Elle se place avant la ligne qui déclare la classe et indique que la classe en question doit être gérée par la couche de persistance JPA. En l'absence de cette annotation, toutes les autres annotations JPA seraient ignorées.

Ligne 27 : l'annotation **@Table** désigne la table de la base de données dont la classe est une représentation. Son principal argument est name qui désigne le nom de la table. En l'absence de cet argument, la table portera le nom de la classe, ici [Personne]. Dans notre exemple, l'annotation **@Table** est donc superflue.
Ligne 43 : l'annotation **@Id** sert à désigner le champ dans la classe qui est image de la clé primaire de la table. Cette annotation est obligatoire. Elle indique ici que le champ id de la **ligne 46** est l'image de la clé primaire de la table.
Ligne 46 : l'annotation **@Column** sert à faire le lien entre un champ de la classe et la colonne de la table dont le champ est l'image. L'attribut name indique le nom de la colonne dans la table. En l'absence de cet attribut, la colonne porte le même nom que le champ. Dans notre exemple, l'argument name n'était donc pas obligatoire. L'argument nullable=false indique que la colonne associée au champ ne peut avoir la valeur NULL et que donc le champ doit avoir nécessairement une valeur.
Ligne 44 : l'annotation **@GeneratedValue** indique comment est générée la clé primaire lorsqu'elle est générée automatiquement par le SGBD.
Strategy = **GenerationType.IDENTITY** : La génération de la clé primaire se fera à partir d'une Identité propre au SGBD. Il utilise un type de colonne spéciale à la base de données.
Exemple pour MySQL, il s'agit d'un AUTO_INCREMENT.

Strategy = **GenerationType.AUTO** : La génération de la clé primaire est laissée à l'implémentation.
Strategy = **GenerationType. TABLE** : La génération de la clé primaire se fera en utilisant une table dédiée hibernate_sequence qui stocke les noms et les valeurs des séquences.
Cette stratégie doit être utilisée avec une autre annotation qui est @TableGenerator.
Exemple:
@GeneratedValue (strategy = GenerationType.TABLE, generator = « clientGenerator »)
@TableGenerator (name = "clientGenerator", pkColumnName = "nom_colonne_pk", valueColumnName = "nom_colonne_valeur_pk", allocationSize = 1)

Strategy = **GenerationType. SEQUENCE** : La génération de la clé primaire se fera par une séquence définie dans le SGBD, auquel on ajoutera l'attribut generator. Cette stratégie doit être utilisée avec une autre annotation qui est @SequenceGenerator. Cette annotation possède l'attribut name pour le nom du generator, l'attribut sequenceName pour le nom de la séquence et enfin allocationSize qui est l'incrémentation de la valeur de la séquence.
Exemple:

@GeneratedValue (strategy = GenerationType.SEQUENCE, generator = « generator_client »)

@SequenceGenerator (name = « generator_client », sequenceName = « WINDEV_SEQ », allocationSize = 1)
Ligne 66 : l'annotation **@Version** désigne le champ qui sert à gérer les accès concurrents à une même ligne de la table. Pour comprendre ce problème d'accès concurrents à une même ligne de la table « **personne** », supposons qu'une application web permette la mise à jour d'une personne et examinons le cas suivant :
Au temps T1, un utilisateur U1 entre en modification d'une personne P. A ce moment, le pods est à100. Il passe ce nombre à 110 mais avant qu'il ne valide sa modification, un utilisateur U2 entre en modification de la même personne P. Puisque U1 n'a pas encore validé sa modification, U2 voit sur son écran le poids à 100. U2 passe le nom de la personne P en majuscules. Puis U1 et U2 valident leurs modifications dans cet ordre. C'est la modification de U2 qui va gagner : dans la base, le nom va passer en majuscules et le poids va rester à 100 alors même que U1 croit l'avoir changé en 110.
La notion de version de personne nous aide à résoudre ce problème. On reprend le même cas d'usage :
Au temps T1, un utilisateur U1 entre en modification d'une personne P. A ce moment, le poids est à 100 et la version V1. Il passe le poids à 110 mais avant qu'il ne valide sa modification, un utilisateur U2 entre en modification de la même personne P. Puisque U1 n'a pas encore validé sa modification, U2 voit le poids à 100 et la version à V1. U2 passe le nom de la personne P en majuscules. Puis U1 et U2 valident leurs modifications dans cet ordre. Avant de valider une modification, on vérifie que celui qui modifie une personne P détient la même version que la personne P actuellement enregistrée. Ce sera le cas de l'utilisateur U1. Sa modification est donc acceptée et on change alors la version de la personne modifiée de V1 à V2 pour noter le fait que la personne a subi un changement. Lors de la validation de la modification de U2, on va s'apercevoir que U2 détient une version V1 de la personne P, alors qu'actuellement la version de celle-ci est V2. On va alors pouvoir dire à l'utilisateur U2 que quelqu'un est passé avant lui et qu'il doit repartir de la nouvelle version de la personne P. Il le fera, récupèrera une personne P de version V2 qui a maintenant un enfant, passera le nom en majuscules, validera. Sa modification sera acceptée si la personne P enregistrée a toujours la version V2. Au final, les modifications faites par U1 et U2 seront prises en compte alors que dans le cas d'usage sans version, l'une des modifications était perdue.

Nous allons mettre à jour la balise « **dependencies** » (après la balise « **packaging** ») dans notre fichier « **pom.xml** » pour ajouter les dépendances de la librairie **JPA/Hibernate** et celui du driver JDBC de MySQL.

Le fichier « **pom.xml** » aura pour contenu finale :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cours.hibernate</groupId>
  <artifactId>maven-personnes-dao-jpa-hibernate</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <!-- Hibernate dependencies -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>4.3.11.Final</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>4.3.11.Final</version>
    </dependency>
    <!-- Fin Hibernate dependencies -->
    <!-- Debut MySql dependencies -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>8.0.18</version>
    </dependency>
    <!-- Fin MySql dependencies -->
  </dependencies>
</project>
```

4. *Création de la classe MainInitJPAProject*

Créer de la classe **com.cours.main.MainInitJPAProject** avec le contenu :

```java
package com.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

/**
 *
 * @author ElHadji
 */
public class MainInitJPAProject {

    private static final String persistenceUnit = "PersonnesPU";

    public static void main(String[] args) {
        String methodName = "main";
        EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
        EntityManager em = emf.createEntityManager();
        System.out.println("AVANT BEGIN TRANSACTION");
        em.getTransaction().begin();
        System.out.println("APRES BEGIN TRANSACTION");
        em.getTransaction().commit();
        System.out.println("APRES COMMIT TRANSACTION");
        // libération ressources
        em.close();
        emf.close();
        System.out.println("APRES LES CLOSES");
    }
}
```

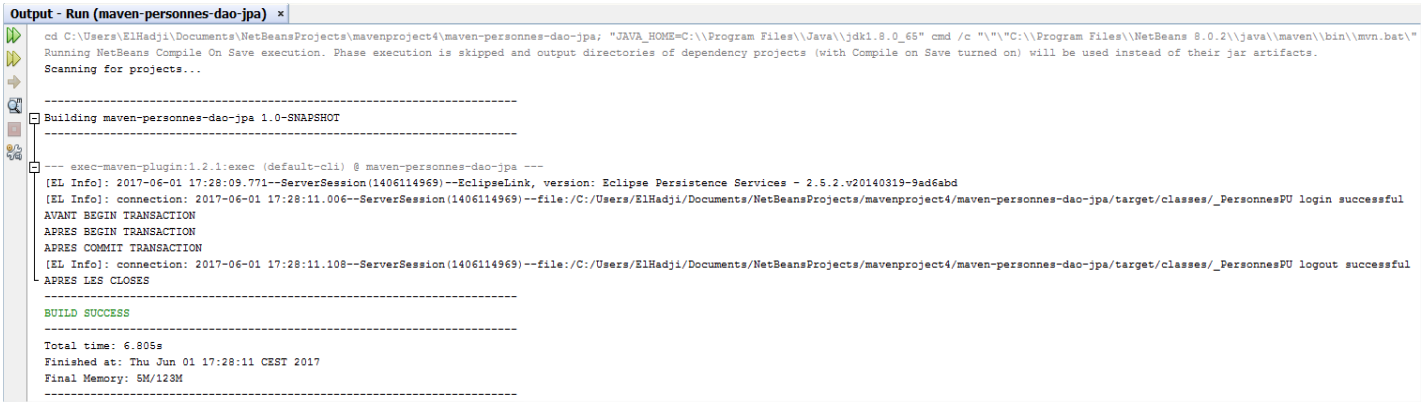
En version copiable :

```java
package com.main;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

/**
 *
 * @author ElHadji
 */
public class MainInitJPAProject {
   private static final String persistenceUnit = "PersonnesPU";

   public static void main(String[] args) {
      String methodName = "main";
      EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
      EntityManager em = emf.createEntityManager();
      System.out.println("AVANT BEGIN TRANSACTION");
      em.getTransaction().begin();
      System.out.println("APRES BEGIN TRANSACTION");
      em.getTransaction().commit();
      System.out.println("APRES COMMIT TRANSACTION");
      // libération ressources
      em.close();
      emf.close();
```

```
        System.out.println("APRES LES CLOSES");
    }
}
```

Ligne 13 : création de l'EntityManager qui gère la couche de persistance.
Ligne 16 : création d'une transaction.
Ligne 18 : commit de la transaction courante.
Ligne 21-22 : fermiture des ressources persistantes.

Au lancement du programme on obtient :



On voit d'après les logs qu'on arrive à instancier un **EntityManager** valide, à debuter une transaction et la commiter. Ceci prouve bien que la configuration de l'ORM **JPA/Hibernate** est correcte. Si la configuration était incorrecte alors nous aurions eu des erreurs de type exceptions.

5. *Ajout des methodes CRUD*

Ajouter les methodes **findAllPersonnes**, **findPersonneById**, **createPersonne**, **updatePersonne** et **removePersonne** dans la classe **com.cours.main.MainInitJPAProject**.

La methode **findAllPersonnes** sera :

```
1   public static void findAllPersonnes() {
2           String methodName = "findAllPersonnes";
3           List<Personne> personnes = null;
4           try {
5               EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
6               EntityManager em = emf.createEntityManager();
7               personnes = em.createNamedQuery("Personne.findAll").getResultList();
8               // Autre methode : personnes = em.createQuery("select person from Personne person order by person.nom asc").getResultList();
9               System.out.println("Voici la liste des personnes : " + personnes);
10              em.close();
11              emf.close();
12          } catch (Exception e) {
13              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
14          }
15  }
```

En version copiable :

```
public static void findAllPersonnes() {
    String methodName = "findAllPersonnes";
    List<Personne> personnes = null;
    try {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
      EntityManager em = emf.createEntityManager();
      personnes = em.createNamedQuery("Personne.findAll").getResultList();
// Autre methode : personnes = em.createQuery("select person from Personne person order by person.nom asc").getResultList();
      System.out.println("Voici la liste des personnes : " + personnes);
      em.close();
      emf.close();
    } catch (Exception e) {
      System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
    }
}
```

Ligne 7-8: Recupération de la liste des personnes par l'intermediaire des méthodes **createNamedQuery** et **createQuery** de l'interface « **javax.persistence.EntityManager** ».

La méthode **findPersonneById** sera :

```
1   public static void findPersonneById(int idPerson) {
2           String methodName = "findPersonneById";
3           Personne person;
4           try {
5               EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
6               EntityManager em = emf.createEntityManager();
7               person = (Personne) em.find(Personne.class, idPerson);
8               // Autre methode :  person = (Personne) em.createQuery("select person from Personne person where person.idPersonne=:toto").setParameter("toto", idPerson).getSingleResult();
9               System.out.println("La personne " + person + " a ete trouve.");
10              em.close();
11              emf.close();
12          } catch (Exception e) {
13              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
14          }
15  }
```

En version copiable:

```
public static void findPersonneById(int idPerson) {
    String methodName = "findPersonneById";
    Personne person;
    try {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
      EntityManager em = emf.createEntityManager();
      person = (Personne) em.find(Personne.class, idPerson);
/* Autre methode :  person = (Personne) em.createQuery("select person from Personne person where person.idPersonne=:toto")
.setParameter("toto", idPerson).getSingleResult();*/
      System.out.println("La personne " + person + " a ete trouve.");
      em.close();
      emf.close();
    } catch (Exception e) {
      System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
    }
}
```

Ligne 7: Recupération de la personne qui a pour identifiant **idPerson** par l'intermediaire de la méthode **find** de l'interface « **javax.persistence.EntityManager** ».

La méthode **createPersonne** sera :

```
1   public static void createPersonne(Personne person) {
2           String methodName = "createPersonne";
3           try {
4               EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
5               EntityManager em = emf.createEntityManager();
6               em.getTransaction().begin();
7               em.persist(person);
8               em.getTransaction().commit();
9               em.close();
10              emf.close();
11              System.out.println("Une nouvelle personne a été cree  avec l'id:" + person.getIdPersonne());
12          } catch (Exception e) {
13              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
14          }
15  }
16
```

En version copiable:

```
public static void createPersonne(Personne person) {
    String methodName = "createPersonne";
    try {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      em.persist(person);
      em.getTransaction().commit();
      em.close();
      emf.close();
      System.out.println("Une nouvelle personne a été cree  avec l'id:" + person.getIdPersonne());
    } catch (Exception e) {
      System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
    }
}
```

Ligne 7: Création d'une nouvelle personne en base de données par l'intermediaire de la méthode **persist** de l'interface « **javax.persistence.EntityManager** ».

La méthode **updatePersonne** sera :

```
1   public static void updatePersonne(Personne person) {
2           String methodName = "updatePersonne";
3           try {
4               EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
5               EntityManager em = emf.createEntityManager();
6               em.getTransaction().begin();
7               em.merge(person);
8               em.getTransaction().commit();
9               em.close();
10              emf.close();
11              System.out.println("La personne  d'id " + person.getIdPersonne() + " a mis à jour.");
12          } catch (Exception e) {
13              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
14          }
15  }
```

En version copiable :

```
public static void updatePersonne(Personne person) {
    String methodName = "updatePersonne";
    try {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      em.merge(person);
      em.getTransaction().commit();
      em.close();
      emf.close();
      System.out.println("La personne  d'id " + person.getIdPersonne() + " a mis à jour.");
    } catch (Exception e) {
      System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
    }
}
```

Ligne 7: Mise à jour de la personne **person** par l'intermediaire de la méthode **merge** de l'interface « **javax.persistence.EntityManager** ».

La méthode **removePersonne** sera :

```
1   public static void removePersonne(Personne person) {
2           String methodName = "removePersonne";
3           try {
4               int idPerson = person.getIdPersonne();
5               EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
6               EntityManager em = emf.createEntityManager();
7               em.getTransaction().begin();
8               em.remove(em.merge(person));
9               em.getTransaction().commit();
10              em.close();
11              emf.close();
12              System.out.println("La personne  d'id " + idPerson + " a ete supprimé.");
13          } catch (Exception e) {
14              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
15          }
16  }
```

En version copiable :

```
public static void removePersonne(Personne person) {
    String methodName = "removePersonne";
    try {
      int idPerson = person.getIdPersonne();
      EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      em.remove(em.merge(person));
      em.getTransaction().commit();
      em.close();
      emf.close();
      System.out.println("La personne  d'id " + idPerson + " a ete supprimé.");
    } catch (Exception e) {
      System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e);
    }
}
```

Ligne 7: Suppression de la personne **person** par l'intermediaire de la méthode **remove** de l'interface « **javax.persistence.EntityManager** ».

La méthode main devient :

```
1   public static void main(String[] args) {
2           String methodName = "main";
3           findAllPersonnes();
4           Personne personneCrud = new Personne("Marc", "Dupont", 75, 150, "rue des passeurs", "Laval", "53000", "France");
5           createPersonne(personneCrud);
6           findPersonneById(personneCrud.getIdPersonne());
7           personneCrud.setPrenom("Marc Bis");
8           personneCrud.setNom("Dupont Bis");
9           updatePersonne(personneCrud);
10          findPersonneById(personneCrud.getIdPersonne());
11          removePersonne(personneCrud);
12          findPersonneById(personneCrud.getIdPersonne());
13  }
14
```

En version copiable :

```
public static void main(String[] args) {
    String methodName = "main";
    findAllPersonnes();
    Personne personneCrud = new Personne("Marc", "Dupont", 75, 150, "rue des passeurs", "Laval", "53000", "France");
    createPersonne(personneCrud);
    findPersonneById(personneCrud.getIdPersonne());
    personneCrud.setPrenom("Marc Bis");
    personneCrud.setNom("Dupont Bis");
    updatePersonne(personneCrud);
    findPersonneById(personneCrud.getIdPersonne());
    removePersonne(personneCrud);
    findPersonneById(personneCrud.getIdPersonne());
}
```

La méthode main devient :

```
1   public static void main(String[] args) {
2           String methodName = "main";
3           findAllPersonnes();
4           Personne personneCrud = new Personne("Marc", "Dupont", 75, 150, "rue des passeurs", "Laval", "53000", "France");
5           createPersonne(personneCrud);
6           findPersonneById(personneCrud.getIdPersonne());
7           personneCrud.setPrenom("Marc Bis");
8           personneCrud.setNom("Dupont Bis");
9           updatePersonne(personneCrud);
10          findPersonneById(personneCrud.getIdPersonne());
11          removePersonne(personneCrud);
12          findPersonneById(personneCrud.getIdPersonne());
13  }
14
```

En version copiable :

```
public static void main(String[] args) {
    String methodName = "main";
    findAllPersonnes();
    Personne personneCrud = new Personne("Marc", "Dupont", 75, 150, "rue des passeurs", "Laval", "53000", "France");
    createPersonne(personneCrud);
    findPersonneById(personneCrud.getIdPersonne());
    personneCrud.setPrenom("Marc Bis");
    personneCrud.setNom("Dupont Bis");
    updatePersonne(personneCrud);
    findPersonneById(personneCrud.getIdPersonne());
    removePersonne(personneCrud);
    findPersonneById(personneCrud.getIdPersonne());
}
```

A l'exécution de la méthode **main**, on obtient sur la console :

```
------------------------------------------------------------------------
Building maven-personnes-dao-jpa 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-personnes-dao-jpa ---
juin 04, 2017 5:34:27 PM org.hibernate.annotations.common.Version <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {4.0.1.Final}
juin 04, 2017 5:34:27 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {4.0.1.Final}
juin 04, 2017 5:34:27 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
juin 04, 2017 5:34:27 PM org.hibernate.cfg.Environment buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
juin 04, 2017 5:34:28 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000402: Using Hibernate built-in connection pool (not for production use!)
juin 04, 2017 5:34:28 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000115: Hibernate connection pool size: 20
juin 04, 2017 5:34:28 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000006: Autocommit mode: true
juin 04, 2017 5:34:28 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000401: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/base_personnes_jpa]
juin 04, 2017 5:34:28 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000046: Connection properties: {user=application, password=****, autocommit=true, release_mode=auto}
juin 04, 2017 5:34:29 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
juin 04, 2017 5:34:29 PM org.hibernate.engine.transaction.internal.TransactionFactoryInitiator initiateService
INFO: HHH000268: Transaction strategy: org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory
juin 04, 2017 5:34:29 PM org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory <init>
INFO: HHH000397: Using ASTQueryTranslatorFactory
Voici la liste des personnes : [[idPersonne=1, prenom=Julien, nom=Marshall, poids=55.0, taille=160.0, rue=rue de Nantes, codePostal=53000 , ville=Laval,  pays=France, version=0], [idPersonne=2, prenom=Julien, nom=Claire, poids=85.0,
juin 04, 2017 5:34:31 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl stop
INFO: HHH000030: Cleaning up connection pool [jdbc:mysql://localhost:3306/base_personnes_jpa]
juin 04, 2017 5:34:31 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000402: Using Hibernate built-in connection pool (not for production use!)
juin 04, 2017 5:34:31 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000115: Hibernate connection pool size: 20
juin 04, 2017 5:34:31 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000006: Autocommit mode: true
juin 04, 2017 5:34:31 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000401: using driver [com.mysql.jdbc.Driver] at URL [jdbc:mysql://localhost:3306/base_personnes_jpa]
juin 04, 2017 5:34:31 PM org.hibernate.service.jdbc.connections.internal.DriverManagerConnectionProviderImpl configure
INFO: HHH000046: Connection properties: {user=application, password=****, autocommit=true, release_mode=auto}
juin 04, 2017 5:34:31 PM org.hibernate.dialect.Dialect <init>
```

V) Exercice d'application : DAO Hibernate + Test Unitaire JUnit

Dans le projet précédant nous avons vu en vrac comment utiliser **l'EntityManagerFactory** et l'**EntityManager**. Nous allons maintenant réorganiser un peu le projet **maven-dao-personnes-jpa-hibernate** à travers l'élaboration des classes **PersonneDao** et **ServiceFacade** avec leurs interfaces respectives **IPersonneDao** et **IServiceFacade**.

Nous utiliserons une architecture DAO (Patron de conception DAO : Data Access Object).
Mettez en place cette architecture DAO avec tous ses composants.

Il faudra donc implémenter toutes les méthodes de la classe **PersonneDao,** puis lancer quelques tests dans la classe **MainApp**, ensuite lancer le test unitaire JUnit **JUnitDao** (vous vous assurerez que tout est au vert c'est-à-dire votre implémentation fonctionne correctement).

VI) Les relations entre entités

Maintenant que vous comprenez les opérations CRUD de base, nous allons voir comment concevoir un modèle de données complet avec JPA.
Comment matérialiser les clés étrangères ? Comment JPA assure-t-il la cohérence de nos objets avec des relations ?

1. *Relation OneToOne sans table d'association*

Considérons une relation **OneToOne** liant une entité **Personne** et une entité **Adresse**. Ceci veut dire que la classe **Personne** a pour attribut l'entité **Adresse** comme le montre l'exemple ci-dessous.

Du point de SQL cela donne :

```
DROP DATABASE IF EXISTS base_personnes_one_to_one;
CREATE DATABASE base_personnes_one_to_one DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_personnes_one_to_one.* TO'application'@'localhost';
USE base_personnes_one_to_one;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Personne;
DROP TABLE IF EXISTS Adresse;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Personne (
   idPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
   idAdresse INTEGER UNIQUE NOT NULL REFERENCES Adresse(idAdresse),
   prenom VARCHAR(100),
   nom VARCHAR(100),
   poids double,
   taille double,
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Les classes **Personne** et **Adresse** auront pour contenu :

```
// Relation OneToOne sans table d'association

@Entity
@Table(name = "Personne")
@XmlRootElement
public class Personne implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;


    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "idAdresse", nullable = false)
    private Adresse adresse;

    //Autres attribut + getters et setters

}



@Entity
@Table(name = "Adresse")
@XmlRootElement
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idAdresse")
    private Integer idAdresse;

    /**
     * **** Debut bidirectionnel ****
     */
    @OneToOne(mappedBy = "adresse")
    private Personne personne;
    /**
     * **** Fin bidirectionnel ****
     */

    //Autres attribut + getters et setters
}
```

2. *Relation OneToOne avec table d'association*

Considérons toujours la relation **OneToOne** liant une entité **Personne** et une entité **Adresse** avec une table d'association.

Du point de SQL cela donne :

```
DROP DATABASE IF EXISTS base_personnes_one_to_one_table_assoc;
CREATE DATABASE base_personnes_one_to_one_table_assoc DEFAULT CHARACTER SET utf8 COLLATE utf
8_general_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_personnes_one_to_one_table_assoc.* TO'application'@'localhost';
USE base_personnes_one_to_one_table_assoc;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Personne;
DROP TABLE IF EXISTS Adresse;
DROP TABLE IF EXISTS Personne_Adresse_Associations;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Personne (
   idPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
   prenom VARCHAR(100),
   nom VARCHAR(100),
   poids double,
   taille double,
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Personne_Adresse_Associations (
   idPersonne INTEGER UNIQUE NOT NULL REFERENCES Personne(idPersonne),
   idAdresse  INTEGER UNIQUE NOT NULL REFERENCES Adresse(idAdresse),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Les classes **Personne** et **Adresse** auront pour contenu :

```
// Relation OneToOne avec table d'association
@Entity
@Table(name = "Personne")
@XmlRootElement
public class Personne implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;


    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(name = "Personne_Adresse_Associations",
    joinColumns = @JoinColumn(name = "idPersonne"),
    inverseJoinColumns = @JoinColumn(name = "idAdresse"))
    private Adresse adresse;

    //Autres attribut + getters et setters

}

@Entity
@Table(name = "Adresse")
@XmlRootElement
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idAdresse")
    private Integer idAdresse;

    /**
     * **** Debut bidirectionnel ****
     */
    @OneToOne
    @JoinTable(name = "Personne_Adresse_Associations",
            joinColumns = @JoinColumn(name = "idAdresse"),
            inverseJoinColumns = @JoinColumn(name = "idPersonne"))
    private Personne personne;
    /**
     * **** Fin bidirectionnel ****
     */
    //Autres attribut + getters et setters
}
```

3. *Relation OneToMany sans table d'association*

Considérons une relation **OneToMany** liant une entité **Personne** et une entité **Adresse**. Ceci veut dire que la classe **Personne** a pour attribut une liste de l'entité **Adresse** comme le montre l'exemple ci-dessous.

Du point de SQL cela donne :

```
DROP DATABASE IF EXISTS base_personnes_one_to_many;
CREATE DATABASE base_personnes_one_to_many DEFAULT CHARACTER SET utf8 COLLATE utf8_general_
ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_personnes_one_to_many.* TO'application'@'localhost';
USE base_personnes_one_to_many;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Personne;
DROP TABLE IF EXISTS Adresse;

CREATE TABLE Personne (
   idPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
   prenom VARCHAR(100),
   nom VARCHAR(100),
   poids double,
   taille double,
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   idPersonne INTEGER NOT NULL REFERENCES Personne(idPersonne),
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Les classes **Personne** et **Adresse** auront pour contenu :

```
// Relation OneToMany sans table d'association
@Entity
@Table(name = "Personne")
@XmlRootElement
public class Personne implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;


    @OneToMany(targetEntity = Adresse.class, mappedBy = "personne",
    cascade = CascadeType.ALL,fetch = FetchType.LAZY)
    private List<Adresse> adresses = new ArrayList<Adresse>();

    //Autres attribut + getters et setters

}

@Entity
@Table(name = "Adresse")
@XmlRootElement
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idAdresse")
    private Integer idAdresse;

    /**
     * **** Debut bidirectionnel ****
     */
    @ManyToOne
    @JoinColumn(name = "idPersonne", nullable = false)
    private Personne personne;
    /**
     * **** Fin bidirectionnel ****
     */

    //Autres attribut + getters et setters
}
```

4. *Relation OneToMany avec table d'association*

Considérons une relation **OneToMany** liant une entité **Personne** et une entité **Adresse** avec une table d'association comme le montre l'exemple ci-dessous.

Du point de SQL cela donne :

```
DROP DATABASE IF EXISTS base_personnes_one_to_many_table_assoc;
CREATE DATABASE base_personnes_one_to_many_table_assoc DEFAULT CHARACTER SET utf8 COLLATE u
tf8_general_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_personnes_one_to_many_table_assoc.* TO'application'@'localhost';
USE base_personnes_one_to_many_table_assoc;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Personne;
DROP TABLE IF EXISTS Adresse;
DROP TABLE IF EXISTS Personne_Adresse_Associations;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Personne (
   idPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
   prenom VARCHAR(100),
   nom VARCHAR(100),
   poids double,
   taille double,
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Personne_Adresse_Associations (
   idAdresse INTEGER  NOT NULL UNIQUE REFERENCES  Adresse(idAdresse),
   idPersonne INTEGER NOT NULL REFERENCES Personne(idPersonne),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Les classes **Personne** et **Adresse** auront pour contenu :

```
// Relation OneToMany avec table d'association
@Entity
@Table(name = "Personne")
@XmlRootElement
public class Personne implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;


    @OneToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "Personne_Adresse_Associations",
    joinColumns = @JoinColumn(name = "idPersonne"),
    inverseJoinColumns = @JoinColumn(name = "idAdresse"))
    private Adresse adresse;

    //Autres attribut + getters et setters
}

@Entity
@Table(name = "Adresse")
@XmlRootElement
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idAdresse")
    private Integer idAdresse;

    /**
     * **** Debut bidirectionnel ****
     */
    @ManyToOne
    @JoinTable(name = "Personne_Adresse_Associations",
            joinColumns = @JoinColumn(name = "idAdresse"),
            inverseJoinColumns = @JoinColumn(name = "idPersonne"))
    private Personne personne;
    /**
     * **** Fin bidirectionnel ****
     */

    //Autres attribut + getters et setters
}
```

5. *Relation ManyToMany sans table d'association*

Considérons une relation **ManyToMany** liant une entité **Personne** et une entité **Adresse**. Ceci veut dire que la classe **Personne** a pour attribut une liste de l'entité **Adresse** et la classe **Adresse** a pour attribut une liste de l'entité **Personne** comme le montre l'exemple ci-dessous :

Du point de SQL cela donne :

```
DROP DATABASE IF EXISTS base_personnes_many_to_many;
CREATE DATABASE base_personnes_many_to_many DEFAULT CHARACTER SET utf8 COLLATE utf8_genera
l_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_personnes_many_to_many.* TO'application'@'localhost';
USE base_personnes_many_to_many;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Personne;
DROP TABLE IF EXISTS Adresse;
DROP TABLE IF EXISTS Personne_Adresse_Associations;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE Personne (
   idPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
   prenom VARCHAR(100),
   nom VARCHAR(100),
   poids double,
   taille double,
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;


CREATE TABLE Personne_Adresse_Associations (
   idPersonne INTEGER NOT NULL REFERENCES Personne(idPersonne),
   idAdresse INTEGER  NOT NULL REFERENCES  Adresse(idAdresse),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Les classes **Personne** et **Adresse** auront pour contenu :

```
// Relation ManyToMany
@Entity
@Table(name = "Personne")
@XmlRootElement
public class Personne implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;


    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(name = "Personne_Adresse_Associations",
            joinColumns = @JoinColumn(name = "idPersonne"),
            inverseJoinColumns = @JoinColumn(name = "idAdresse"))
    private List<Adresse> adresses = new ArrayList<Adresse>();

    //Autres attribut + getters et setters
}

@Entity
@Table(name = "Adresse")
@XmlRootElement
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idAdresse")
    private Integer idAdresse;

    /**
     * **** Debut bidirectionnel ****
     */
    @ManyToMany
    @JoinTable(name = "Personne_Adresse_Associations",
            joinColumns = @JoinColumn(name = "idAdresse"),
            inverseJoinColumns = @JoinColumn(name = "idPersonne"))
    private List<Personne> personnes = new ArrayList<>();
    /**
     * **** Fin bidirectionnel ****
     */
    //Autres attribut + getters et setters
}
```

VII) Les relations d'héritage avec JPA

JPA support parfaitement l'héritage. Il possède trois stratégies distinctes pour stocker les objets d'une hiérarchie de classes Java en base de données. La différence entre ces trois stratégies réside dans la manière dont les données vont être stockées en mémoire. C'est trois stratégies sont :

- **Stratégie « une table pour la hiérarchie de classes »** : une seule table permettra de stocker les données pour toutes les classes de votre hiérarchie d'héritage. Un discriminant permettra de connaître le type concrètement utilisé.
- **Stratégie « tables jointes »** : une table permet de stocker les données communes à la hiérarchie. Ensuite, il y a autant de tables que de sous-classes. Les tables associées aux sous-classes réalisent des jointures sur la table associée au type de base.
- **Stratégie « une table par classes concrète »** : on créé autant de tables que de classes concrètes dans la hiérarchie de classes. Chaque table définie autant de colonnes que la classe porte d'attributs (tout héritage considéré). Les colonnes correspondant aux attributs de la classe de base sont donc dupliquées dans chaque table.

1. *Stratégie « une table pour la hiérarchie de classes »*

Avec cette stratégie, vous n'avez besoin que d'une seule table. Nous allons l'appeler **Adresse**. Pour distinguer le type d'adresse, nous allons avoir une colonne de plus que le nombre d'attributs proposé ci-dessus : cette colonne sera appelée le discriminant et en fonction de sa valeur nous aurons à faire à un type d'adresse ou à un autre. Le discriminant sera, en base de données, ce qu'il y aura de plus proche de notre notion d'héritage.

Considérons la base de données **base_heritage_discriminator** dont le script aura comme contenu :

```
DROP DATABASE IF EXISTS base_heritage_discriminator;
CREATE DATABASE base_heritage_discriminator DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_heritage_discriminator.* TO'application'@'localhost';
USE base_heritage_discriminator;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Adresse;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   discriminator tinyint NOT NULL,
   version INTEGER DEFAULT 1
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue de Nantes','Laval','53000','France',1
);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue du Paradis','Paris','75000','France',1
);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue des Passeurs','Paris','75000','France'
,1);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue des Feugrais','Rouen','76000','Franc
e',2);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue du Paradis','Paris','75000','France',2
);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue des Feugrais','Rouen','76000','Franc
e',2);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue de Versaille','Paris','75000','France',
3);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue des Sages','Lyon','69000','France',3);
INSERT INTO Adresse(rue,ville,codePostal,pays,discriminator) VALUES ('rue des Pauvres','Laval','53000','France'
,3);
```

Considérons les classes **Adresse, AdresseFacturation, AdresseLivraison et AdresseVaccance** dont les contenus seront :

```
@Entity
@Table( name = "Adresse" )
@Inheritance( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn( name="discriminator", discriminatorType = DiscriminatorType.INTEGER )
//@DiscriminatorValue("0")      // Si la classe Adresse n'était pas abstraite
public abstract class Adresse {

   @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int idAdresse = 0;

   // Suite de la classe ...

}

@Entity
@DiscriminatorValue("1")
public class AdresseFacturation extends Adresse {

   // Suite de la classe ...

}

@Entity
@DiscriminatorValue("2")
public class AdresseLivraison extends Adresse {

   // Suite de la classe ...

}


@Entity
@DiscriminatorValue("3")
public class AdresseVaccance extends Adresse {

   // Suite de la classe ...

}
```

2. *Stratégie « tables jointes »*

Il s'agit certainement de la stratégie la plus couramment mise en œuvre. Pour faire simple, il va falloir définir une table par classe et chaque lien l'héritage donnera lieu à une relation entre les deux tables correspondantes.

Considérons la base de données **base_heritage_join_table** dont le script aura comme contenu :

```
DROP DATABASE IF EXISTS base_heritage_join_table;
CREATE DATABASE base_heritage_join_table DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_heritage_join_table.* TO'application'@'localhost';
USE base_heritage_join_table;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Adresse;
DROP TABLE IF EXISTS AdresseFacturation;
DROP TABLE IF EXISTS AdresseLivraison;
DROP TABLE IF EXISTS AdresseVaccance;

CREATE TABLE Adresse (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

 CREATE TABLE AdresseFacturation (
   idAdresse int NOT NULL REFERENCES Adresse(idAdresse),
   moyenPaiement VARCHAR(100),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;


 CREATE TABLE AdresseLivraison (
   idAdresse int NOT NULL REFERENCES Adresse(idAdresse),
   informationAccesLogement VARCHAR(100),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

 CREATE TABLE AdresseVaccance (
   idAdresse int NOT NULL REFERENCES Adresse(idAdresse),
   debutVaccance TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   finVaccance TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Considérons les classes **Adresse, AdresseFacturation, AdresseLivraison et AdresseVaccance** dont les contenus seront :

```
@Entity
@Table( name = "Adresse" )
@Inheritance( strategy = InheritanceType.JOINED )
public abstract class Adresse {

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Integer idAdresse = 0;

   // Suite de la classe ...
}

@Entity
@Table( name = "AdresseFacturation" )
@PrimaryKeyJoinColumn( name = "idAdresse" )
public class AdresseFacturation extends Adresse {

   // Suite de la classe ...
}

@Entity
@Table( name = "AdresseLivraison" )
@PrimaryKeyJoinColumn( name = "idAdresse" )
public class AdresseLivraison extends Adresse {

   // Suite de la classe ...
}


@Entity
@Table( name = "AdresseVaccance" )
@PrimaryKeyJoinColumn( name = "idAdresse" )
public class AdresseVaccance extends Adresse {

   // Suite de la classe ...
}
```

3. *Stratégie « une table par classes concrète »*

On ne pas avoir de jointures, mais on va créer une table en base de données par classe concrète. Le contre coup, c'est que pour certains calculs (par exemple, total des encaissements du jour), plusieurs tables devront être requêtées pour, au final, agréger les résultats.

Considérons la base de données **base_heritage_concrete_table** dont le script aura comme contenu :

```
DROP DATABASE IF EXISTS base_heritage_concrete_table;
CREATE DATABASE base_heritage_concrete_table DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
CREATE USER IF NOT EXISTS'application'@'localhost' IDENTIFIED BY 'passw0rd';
GRANT ALL PRIVILEGES ON base_heritage_concrete_table.* TO'application'@'localhost';
USE base_heritage_concrete_table;

SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS AdresseFacturation;
DROP TABLE IF EXISTS AdresseLivraison;
DROP TABLE IF EXISTS AdresseVaccance;

 CREATE TABLE AdresseFacturation (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   moyenPaiement VARCHAR(100),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

 CREATE TABLE AdresseLivraison (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   informationAccesLogement VARCHAR(100),
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;

 CREATE TABLE AdresseVaccance (
   idAdresse INTEGER PRIMARY KEY AUTO_INCREMENT,
   rue VARCHAR(100),
   codePostal VARCHAR(100),
   ville VARCHAR(100),
   pays VARCHAR(100),
   debutVaccance TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   finVaccance TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   version INTEGER DEFAULT 1
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Considérons les classes **AdresseFacturation, AdresseLivraison et AdresseVaccance** dont les contenus seront :

```
@Entity
@Inheritance( strategy = InheritanceType.TABLE_PER_CLASS )
public abstract class Adresse {

   @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Integer idAdresse = 0;

   // Suite de la classe ...
}

@Entity
@Table( name = "AdresseFacturation" )
public class AdresseFacturation extends Adresse {

   // Suite de la classe ...
}

@Entity
@Table( name = "AdresseLivraison" )
public class AdresseLivraison extends Adresse {

   // Suite de la classe ...
}


@Entity
@Table( name = "AdresseVaccance" )
public class AdresseVaccance extends Adresse {

   // Suite de la classe ...
}
```

VIII) Utilisation de la SessionFactory et Session d'Hibernate

Au lieu d'utiliser **EntityManagerFactory** et **EntityManager** Il est possible d'utiliser **SessionFactory** et **Session** d'Hibernate.
Créer le projet Maven Jar **maven-dao-personnes-hibernate-session-factory**.

Le fichier **pom.xml** aura pour contenu :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cours.dao.jpa.hibernate.session.factory</groupId>
  <artifactId>maven-dao-personnes-jpa-hibernate-session-factory</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>maven-dao-personnes-hibernate-session-factory</name>
  <description>Java Maven Gestion Personnes JPA Hibernate</description>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <mainClass>com.cours.main.MainApp</mainClass>
    <mysql.version>8.0.18</mysql.version>
    <hibernate.version>4.3.11.Final</hibernate.version>
    <jaxb.version>2.0</jaxb.version>
    <log4j.version>1.2.17</log4j.version>
    <commons.logging.version>1.2</commons.logging.version>
    <dbunit.version>2.5.0</dbunit.version>
    <ibatis.version>2.1.7.597</ibatis.version>
    <junit.version>4.13.1</junit.version>
  </properties>
  <dependencies>
    <!-- Debut mysql dependencies -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysql.version}</version>
    </dependency>
    <!-- Fin mysql dependencies -->
    <!-- Hibernate dependencies -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
```

```
      <artifactId>hibernate-entitymanager</artifactId>
      <version>${hibernate.version}</version>
    </dependency>
    <dependency>
      <groupId>javax.xml.bind</groupId>
      <artifactId>jaxb-api</artifactId>
      <version>${jaxb.version}</version>
    </dependency>
    <dependency>
      <groupId>com.sun.xml.bind</groupId>
      <artifactId>jaxb-impl</artifactId>
      <version>${jaxb.version}</version>
    </dependency>
    <!-- Fin Hibernate dependencies -->
    <!-- Debut log4j dependencies -->
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>${log4j.version}</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>apache-log4j-extras</artifactId>
      <version>${log4j.version}</version>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>${commons.logging.version}</version>
    </dependency>
    <!-- Fin log4j dependencies -->
    <dependency>
      <groupId>org.dbunit</groupId>
      <artifactId>dbunit</artifactId>
      <version>${dbunit.version}</version>
    </dependency>
    <dependency>
      <groupId>com.ibatis</groupId>
      <artifactId>ibatis2-common</artifactId>
      <version>${ibatis.version}</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>${junit.version}</version>
    </dependency>
  </dependencies>
  <build>
```

```
    <finalName>maven-dao-personnes-hibernate-session-factory</finalName>
  </build>
</project>
```

Créer le fichier **maven-dao-personnes-jpa-hibernate-session-factory/src/main/resources/hibernate.cfg.xml** dont le contenu sera :

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>

      <property name="hibernate.dialect">
         org.hibernate.dialect.MySQLDialect
      </property>

      <property name="hibernate.connection.driver_class">
         com.mysql.cj.jdbc.Driver
      </property>

      <!-- Assume test is the database name -->

      <property name="hibernate.connection.url">
         jdbc:mysql://localhost:3308/base_personnes?serverTimezone=UTC
      </property>

      <property name="hibernate.connection.username">
         application
      </property>

      <property name="hibernate.connection.password">
         passw0rd
      </property>
      <mapping class="com.cours.entities.Personne" />
   </session-factory>
</hibernate-configuration>
```

Créer le fichier **maven-dao-personnes-jpa-hibernate-session-factory/src/main/java/com/cours/dao/HibernateUtil.java** dont le contenu sera :

```
package com.cours.dao;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
   private static final Log log = LogFactory.getLog(HibernateUtil.class);
   private static SessionFactory sessionFactory;

   private static SessionFactory buildSessionFactory() {
      try {
         Configuration configuration = new Configuration().addResource("hibernate.cfg.xml").configure();
         StandardServiceRegistryBuilder registryBuilder = new StandardServiceRegistryBuilder()
               .applySettings(configuration.getProperties());
         sessionFactory = configuration.buildSessionFactory(registryBuilder.build());
      } catch (Exception e) {
         log.error("Erreur lors de la construction de la SessionFactory , Exception :  " + e);
      }
      return sessionFactory;
   }

   public static SessionFactory getSessionFactory() {
      if (sessionFactory == null)
         sessionFactory = buildSessionFactory();
      return sessionFactory;
   }

   public static void closeSessionFactoryResources(SessionFactory sessionFactory) {
      try {
         if (sessionFactory != null && !sessionFactory.isClosed()) {
            sessionFactory.close();
         }
      } catch (Exception e) {
         log.error("Erreur lors de la fermeture de la SessionFactory , Exception :  " + e);
      }
   }

   public static void rollBack(Transaction transaction) {
      try {
         // RollBack
         if (transaction != null && transaction.isActive()) {
            transaction.rollback();
```

```
            }
        } catch (Exception exRollBack) {
            log.error("Une erreur s'est produite lors du RollBack, Exception : " + exRollBack.getMessage()
                    + " , Exception : " + exRollBack);
        }
    }
}
```

Créer le fichier **maven-dao-personnes-jpa-hibernate-session-factory/src/main/java/com/cours/entities/Personne.java** dont le contenu sera :

```
package com.cours.entities;

import java.io.Serializable;

import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.persistence.Version;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author ElHadji
 */
@Entity
@Table(name = "Personne")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Personne.findAll", query = "SELECT person FROM Personne person"),
    @NamedQuery(name = "Personne.findByIdPersonne", query = "SELECT person FROM Personne person WHER
E person.idPersonne = :idPersonne")})
public class Personne implements Serializable, Comparable<Personne> {

    private static final long serialVersionUID = 7579896135168760237L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idPersonne")
    private Integer idPersonne;
    @Column(name = "prenom")
    private String prenom;
    @Column(name = "nom")
    private String nom;
    @Column(name = "poids")
    private Double poids;
    @Column(name = "taille")
    private Double taille;
    @Column(name = "rue")
    private String rue;
    @Column(name = "codePostal")
```

```
    private String codePostal;
    @Column(name = "ville")
    private String ville;
    @Column(name = "pays")
    private String pays;
    @Column(name = "version")
    @Version
    private Integer version;

    public Personne() {

    }

    public Personne(String prenom, String nom, Double poids, Double taille, String rue, String ville, String codePos
tal,
            String pays) {
        this.prenom = prenom;
        this.nom = nom;
        this.poids = poids;
        this.taille = taille;
        this.rue = rue;
        this.codePostal = codePostal;
        this.ville = ville;
        this.pays = pays;
    }

    public Personne(Integer idPersonne, String prenom, String nom, Double poids, Double taille, String rue,
            String ville, String codePostal, String pays) {
        this.idPersonne = idPersonne;
        this.prenom = prenom;
        this.nom = nom;
        this.poids = poids;
        this.taille = taille;
        this.rue = rue;
        this.codePostal = codePostal;
        this.ville = ville;
        this.pays = pays;
    }

    public Integer getIdPersonne() {
        return idPersonne;
    }

    public void setIdPersonne(Integer idPersonne) {
        this.idPersonne = idPersonne;
    }

    public String getNom() {
        return nom;
```

```
}

public void setNom(String nom) {
    this.nom = nom;
}

public Double getPoids() {
    return poids;
}

public void setPoids(Double poids) {
    this.poids = poids;
}

public Double getTaille() {
    return taille;
}

public void setTaille(Double taille) {
    this.taille = taille;
}

public String getPrenom() {
    return prenom;
}

public void setPrenom(String prenom) {
    this.prenom = prenom;
}

public String getRue() {
    return rue;
}

public void setRue(String rue) {
    this.rue = rue;
}

public String getVille() {
    return ville;
}

public void setVille(String ville) {
    this.ville = ville;
}

public String getCodePostal() {
    return codePostal;
}
```

```
  public void setCodePostal(String codePostal) {
    this.codePostal = codePostal;
  }

  public String getPays() {
    return pays;
  }

  public void setPays(String pays) {
    this.pays = pays;
  }

  public Integer getVersion() {
    return version;
  }

  public void setVersion(Integer version) {
    this.version = version;
  }

  @Override
  public String toString() {
    return String.format(
        "[idPersonne=%s, prenom=%s, nom=%s, poids=%s, taille=%s, rue=%s,  ville=%s, codePostal=%s, pays=
%s]",
        idPersonne, prenom, nom, poids, taille, rue, ville, codePostal, pays);
  }

  @Override
  public int hashCode() {
    int prime = 31;
    int result = 1;
    result = result * prime + this.prenom.hashCode();
    result = result * prime + this.nom.hashCode();
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }
    if (obj == null || !(obj instanceof Personne)) {
      return false;
    }
    Personne otherPerson = (Personne) obj;
    return (otherPerson.getPrenom().equals(getPrenom()) && otherPerson.getNom().equals(getNom()));
  }
```

```
    @Override
    public int compareTo(Personne otherPerson) {
        return (this.getPrenom() + this.getNom()).compareTo(otherPerson.getPrenom() + otherPerson.getNom());
    }

    public double getImc() {
        double imc = 0;
        if (taille != 0) {
            imc = poids / Math.pow(taille / 100, 2);
        }
        return imc;
    }
}
```

Créer le fichier **maven-dao-personnes-jpa-hibernate-session-factory/src/main/java/com/cours/main/MainApp.java** dont le contenu sera :

```
package com.cours.main;

import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.cours.dao.HibernateUtil;
import com.cours.entities.Personne;

public class MainApp {

   private static final Log log = LogFactory.getLog(MainApp.class);

   public static void main(String[] args) {
      testConfigurationJpaHibernate();
      findAllPersonnes();
      Personne personCRUD = findPersonneById(5);
      personCRUD = new Personne("SessionFactory--Edouard", "SessionFactory--
Green", 100.0, 170.0, "rue du paradis",
            "Rouen", "76000", "France");
      personCRUD = createPersonne(personCRUD);
      personCRUD.setPrenom("SessionFactory--Edouard Bis");
      personCRUD.setNom("SessionFactory--Green Bis");
      personCRUD = updatePersonne(personCRUD);
      deletePersonne(personCRUD);
   }

   public static void testConfigurationJpaHibernate() {
      SessionFactory sessionFactory = null;
      Session session = null;
      Transaction transaction = null;
      try {
         // Get Session
         sessionFactory = HibernateUtil.getSessionFactory();
         log.debug("CREATION SessionFactory AVEC SUCCES");
         session = sessionFactory.openSession();
         log.debug("CREATION Session AVEC SUCCES");
         log.debug("AVANT BEGIN TRANSACTION");
         transaction = session.beginTransaction();
         log.debug("APRES BEGIN TRANSACTION");
         transaction.commit();
         log.debug("APRES COMMIT TRANSACTION");
```

```
    } catch (Exception e) {
        HibernateUtil.rollBack(transaction);
        log.error("Une erreur s'est produite lors de l'execution la methode, Exception : " + e.getMessage()
                + " , Exception : " + e);
    } finally {
        // libération ressources
        log.debug("DEBUT FINALEMENT APRES LES CLOSES");
        HibernateUtil.closeSessionFactoryResources(sessionFactory);
        log.debug("FIN FINALEMENT APRES LES CLOSES");
    }
}

public static void findAllPersonnes() {
    List<Personne> personnes = null;
    SessionFactory sessionFactory = null;
    Session session = null;
    try {
        sessionFactory = HibernateUtil.getSessionFactory();
        session = sessionFactory.openSession();
        personnes = session.createCriteria(Personne.class).list();
        log.debug("personnes : " + personnes);
    } catch (Exception e) {
        log.error("Une erreur s'est produite lors de l'execution la methode, Exception : " + e.getMessage()
                + " , Exception : " + e);
    } finally {
        HibernateUtil.closeSessionFactoryResources(sessionFactory);
    }
}

public static Personne findPersonneById(Integer idPersonne) {
    Personne personReturn = null;
    List<Personne> personnes = null;
    SessionFactory sessionFactory = null;
    Session session = null;
    try {
        sessionFactory = HibernateUtil.getSessionFactory();
        session = sessionFactory.openSession();
        personReturn = (Personne) session.get(Personne.class, idPersonne);
        log.debug("personnes : " + personnes);
    } catch (Exception e) {
        log.error("Une erreur s'est produite lors de l'execution la methode, Exception : " + e.getMessage()
                + " , Exception : " + e);
    } finally {
        HibernateUtil.closeSessionFactoryResources(sessionFactory);
    }
    return personReturn;
}

public static Personne createPersonne(Personne person) {
```

```
    SessionFactory sessionFactory = null;
    Session session = null;
    Transaction transaction = null;
    try {
        sessionFactory = HibernateUtil.getSessionFactory();
        session = sessionFactory.openSession();
        transaction = session.beginTransaction();
        session.saveOrUpdate(person);
        transaction.commit();
    } catch (Exception e) {
        HibernateUtil.rollBack(transaction);
        log.error("Une erreur s'est produite lors de l'execution la methode, Exception : " + e.getMessage()
                + " , Exception : " + e);
    } finally {
        HibernateUtil.closeSessionFactoryResources(sessionFactory);
    }
    log.debug("person : " + person);
    return person;
}

public static Personne updatePersonne(Personne person) {
    SessionFactory sessionFactory = null;
    Session session = null;
    Transaction transaction = null;
    try {
        sessionFactory = HibernateUtil.getSessionFactory();
        session = sessionFactory.openSession();
        transaction = session.beginTransaction();
        session.saveOrUpdate(person);
        transaction.commit();
    } catch (Exception e) {
        HibernateUtil.rollBack(transaction);
        log.error("Une erreur s'est produite lors de l'execution la methode, Exception : " + e.getMessage()
                + " , Exception : " + e);
    } finally {
        HibernateUtil.closeSessionFactoryResources(sessionFactory);
    }
    log.debug("person : " + person);
    return person;
}

public static Personne deletePersonne(Personne person) {
    SessionFactory sessionFactory = null;
    Session session = null;
    Transaction transaction = null;
    try {
        sessionFactory = HibernateUtil.getSessionFactory();
        session = sessionFactory.openSession();
        transaction = session.beginTransaction();
```

```
            session.delete(person);
            transaction.commit();
        } catch (Exception e) {
            HibernateUtil.rollBack(transaction);
            log.error("Une erreur s'est produite lors de l'execution la methode, Exception : " + e.getMessage()
                    + " , Exception : " + e);
        } finally {
            HibernateUtil.closeSessionFactoryResources(sessionFactory);
        }
        log.debug("person : " + person);
        person = findPersonneById(person.getIdPersonne());
        log.debug("person : " + person);
        return person;
    }
}
```