COURS DE JAVA – Java JPA Hibernate
El Hadji Gaye - AlizNet

Avec Netbeans.

Auteur	El Hadji Gaye
Pour	Ecole
Date	11/12/2017
Objet	Java JPA Hibernate

I) **Vocabulaire** 3
II) **Généralités sur les ORM** 6
III) **Mise en place de l'implémentation JPA/Hibernate** 7

I) Vocabulaire

- **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 donnée 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) 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. On peut citer **Hibernate**, **Toplink**, **EclipseLink**, **OpenJpa** etc.…

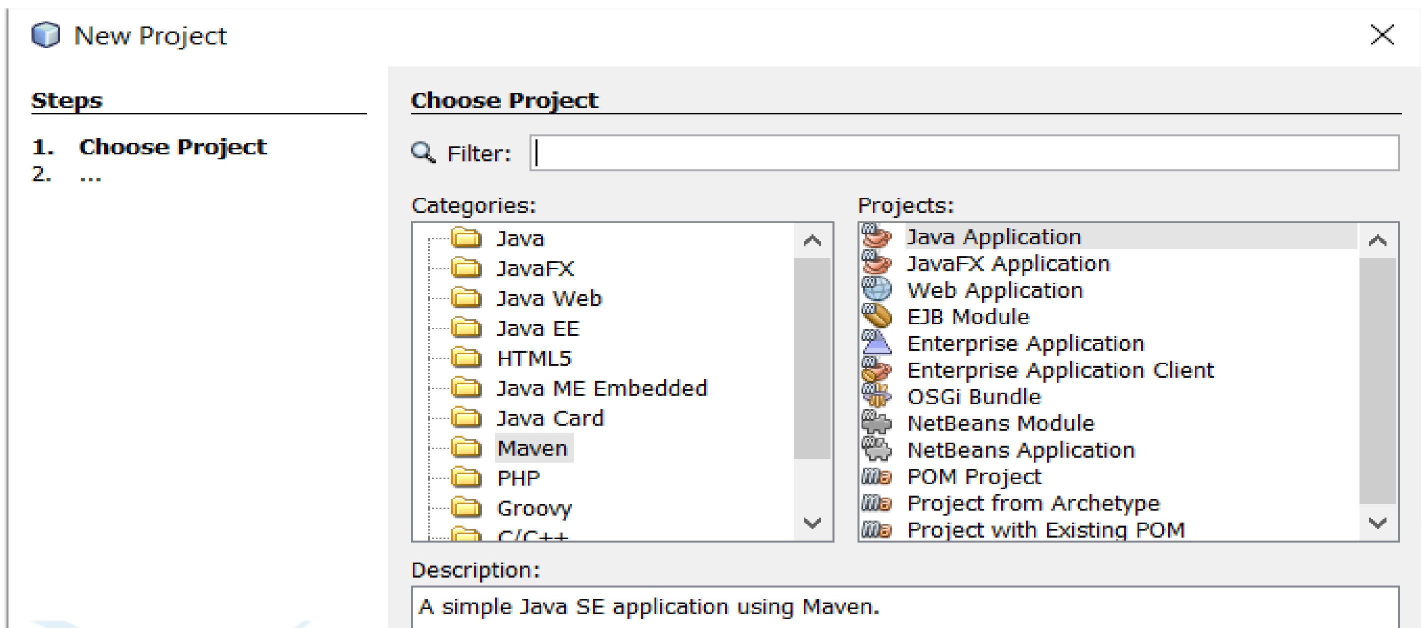
Le standard JPA utilise le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire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

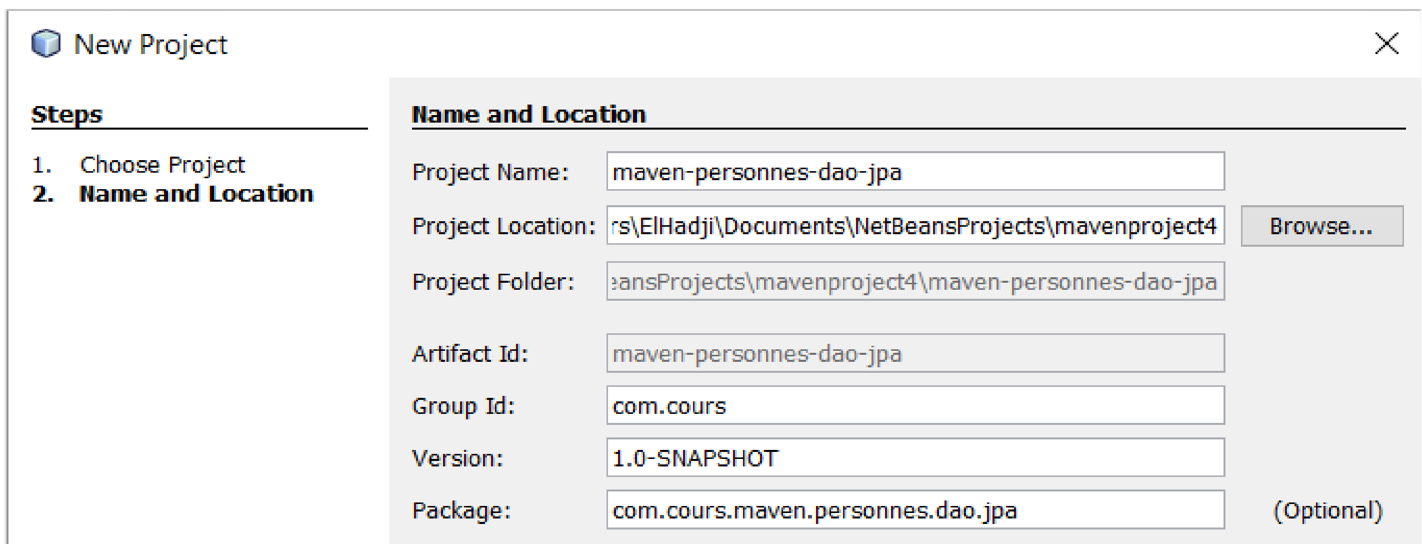
Nous allons voir dans le cadre de ce cours l'implémentation **Hibernate** de la spécification J.P.A.

III) Mise en place de l'implémentation JPA/Hibernate

Nous allons donc créer un projet Maven « **maven-personnes-dao-jpa** » qui sera notre projet de test de l'ORM **JPA/Hibernate**.

« New Project » → « Maven » → « Java Application » → « **maven-personnes-dao-jpa** » → « Finish ».





1. **Création de la base de donnée** « **base_personnes_jpa** » : créer la base MySQL [base_personnes_jpa] avec l'outil de votre choix. La base sera par la suite la propriété de l'utilisateur « **application** » et du mot de passe « **passw0rd** ».

Nous allons utiliser la base de données MYSQL avec une base qui se nommera « **base_personnes_jpa** ».

Pour ajouter un nouveau utilisateur :

- Allez dans la console d'administration de **PhpMyAdmin**, puis cliquer sur « **Utilisateur** », puis sur « **Ajouter un utilisateur** ».
- Mettre « **application** » dans « Nom d'utilisateur », « **passw0rd** » dans « Mot de passe » et « **localhost** » dans « Client », Cocher « **Tout cocher** » dans « Privilèges globaux » et cliquer sur « **Exécuter** ».
- Réitérer l'opération précédente en mettant pour le client « **Tout Client** » c'est-à-dire la valeur **%**, puis la valeur « **127.0.0.1** ».

← Serveur: mysql wampserver

Bases de données | SQL | État | Utilisateurs | Exporter | Importer | Paramètres

Survol des utilisateurs

	Utilisateur	Client	Mot de passe	Privilèges globaux	«Grant»	Action	
☐	N'importe quel	%	--	USAGE	Non	Changer les privilèges	Exporter
☐	N'importe quel	localhost	Non	USAGE	Non	Changer les privilèges	Exporter
☐	application	%	Oui	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	application	localhost	Oui	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	applicationUser	%	Oui	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	applicationUser	127.0.0.1	Oui	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	applicationUser	localhost	Oui	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	root	127.0.0.1	Non	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	root	::1	Non	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter
☐	root	localhost	Non	ALL PRIVILEGES	Oui	Changer les privilèges	Exporter

☐ Tout cocher *Pour la sélection :* Exporter

Ajouter un utilisateur

Effacer les utilisateurs sélectionnés

(Effacer tous les privilèges de ces utilisateurs, puis les effacer.)
☐ Supprimer les bases de données portant le même nom que les utilisateurs.

Bases de données | SQL | État | Utilisateurs | Exporter | Importer | Paramètres | Réplication | plu

Ajouter un utilisateur

Information pour la connexion

Nom d'utilisateur : Entrez une valeur: application

Client : Local localhost

Mot de passe : Entrez une valeur: ••••••••

Entrer à nouveau : ••••••••

Générer un mot de passe: Générer

Base de données pour cet utilisateur

☐ Créer une base portant son nom et donner à cet utilisateur tous les privilèges sur cette base.

☐ Donner les privilèges passepartout (utilisateur_%).

Privilèges globaux ☑ Tout cocher

Veuillez noter que les noms de privilèges sont exprimés en anglais

Données

☑ SELECT
☑ INSERT
☑ UPDATE
☑ DELETE
☑ FILE

Structure

☑ CREATE
☑ ALTER
☑ INDEX
☑ DROP
☑ CREATE TEMPORARY TABLES
☑ SHOW VIEW
☑ CREATE ROUTINE
☑ ALTER ROUTINE
☑ EXECUTE
☑ CREATE VIEW
☑ EVENT
☑ TRIGGER

Administration

☑ GRANT
☑ SUPER
☑ PROCESS
☑ RELOAD
☑ SHUTDOWN
☑ SHOW DATABASES
☑ LOCK TABLES
☑ REFERENCES
☑ REPLICATION CLIENT
☑ REPLICATION SLAVE
☑ CREATE USER

Limites de ressources

Note: Une valeur de 0 (zero) enlève la limite.

MAX QUERIES PER HOUR 0
MAX UPDATES PER HOUR 0
MAX CONNECTIONS PER HOUR 0
MAX USER_CONNECTIONS 0

Exécuter

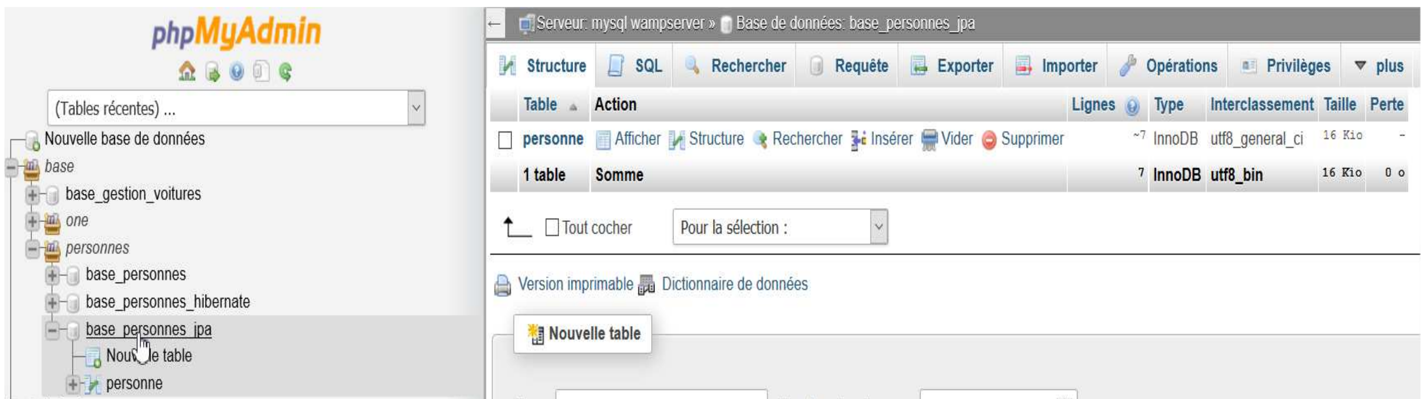
Nous allons jouer le script ci-dessous dans la base de données « **base_personnes_jpa** ».

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

 CREATE TABLE Personne (
        idPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
        prenom VARCHAR(100),
        nom VARCHAR(100),
        poids double,
        taille double,
        rue VARCHAR(100),
        codePostal VARCHAR(100),
        ville VARCHAR(100),
        pays VARCHAR(100),
        version int(15)
 )ENGINE=InnoDB DEFAULT CHARSET=utf8;

 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Julien', 'Marshall',55,160,'rue de Nantes','Laval','53000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Julien', 'Claire',85,175,'rue du Paradis','Paris','75000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Jacques', 'Dupont',62,145,'rue des Passeurs','Paris','75000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Dupont', 'Dupont',62,155,'rue des Passeurs','Paris','75000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Dupond', 'Dupond',62,169,'rue des Passeurs','Paris','75000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Charles', 'Hallyday',100,189,'rue des Feugrais','Rouen','76000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Serge', 'Lama',87,200,'rue des Heureux','Nantes','44000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Vincent', 'Thomas',64,178,'rue de Paris','Rennes','35000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Eric', 'Dummat',78,195,'rue de Versaille','Paris','75000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Nicolas', 'Samuel',112,199,'rue de Saint Louis','Laval','53000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Rémy', 'Guerry',96,186,'rue des Sages','Lyon','69000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Nicolas', 'Drapeau',87,165,'rue Mitterrand','Limoges','87000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Gaelle', 'Letourneau',75,179,'rue Jean François','Rouen','76000','France',0);
 INSERT INTO Personne(prenom,nom,poids,taille,rue,ville,codePostal,pays,version) VALUES
('Anne', 'Claire',85,194,'rue du Paradis','Paris','75000','France',0);
```

Allez dans la console d'administration de **PhpMyAdmin**, puis créer la base de données « **base_personnes_jpa** » et ensuite inserer les valeurs ci-dessus dans l'onglet **SQL**.

Cliquer à gauche sur la nouvelle base de données « **base_personnes_jpa** », puis sur SQL, puis copier coller le contenu du script SQL ci-dessus et enfin cliquer sur **Exécuter**.

En cliquant sur la base « **base_personnes_jpa** », on voit la table « **personne** ».



En cliquant sur « **Afficher** » on obtient :

Afficher | Structure | SQL | Rechercher | Insérer | Exporter | Importer | Privilèges | Opér

Affichage des lignes 0 - 6 (total de 7, Traitement en 0.0014 sec)

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

Profilage [En ligne] [Modifier] [Expliquer SQL] [Créer

Nombre de lignes : 25

Trier sur l'index: Aucune

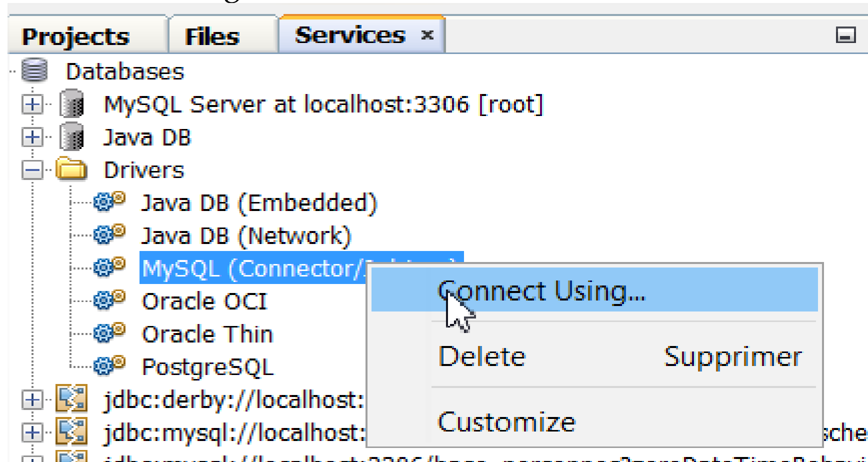
+ Options

←T→			idPersonne	prenom	nom	poids	taille	rue	codePostal	ville	pays	version
Modifier	Copier	Effacer	1	Julien	Marshall	55	160	rue de Nantes	53000	Laval	France	0
Modifier	Copier	Effacer	2	Julien	Claire	85	175	rue du Paradis	75000	Paris	France	0
Modifier	Copier	Effacer	3	Jacques	Dupont	62	145	rue des Passeurs	75000	Paris	France	0
Modifier	Copier	Effacer	4	Dupont	Dupont	62	155	rue des Passeurs	75000	Paris	France	0
Modifier	Copier	Effacer	5	Dupond	Dupond	62	169	rue des Passeurs	75000	Paris	France	0

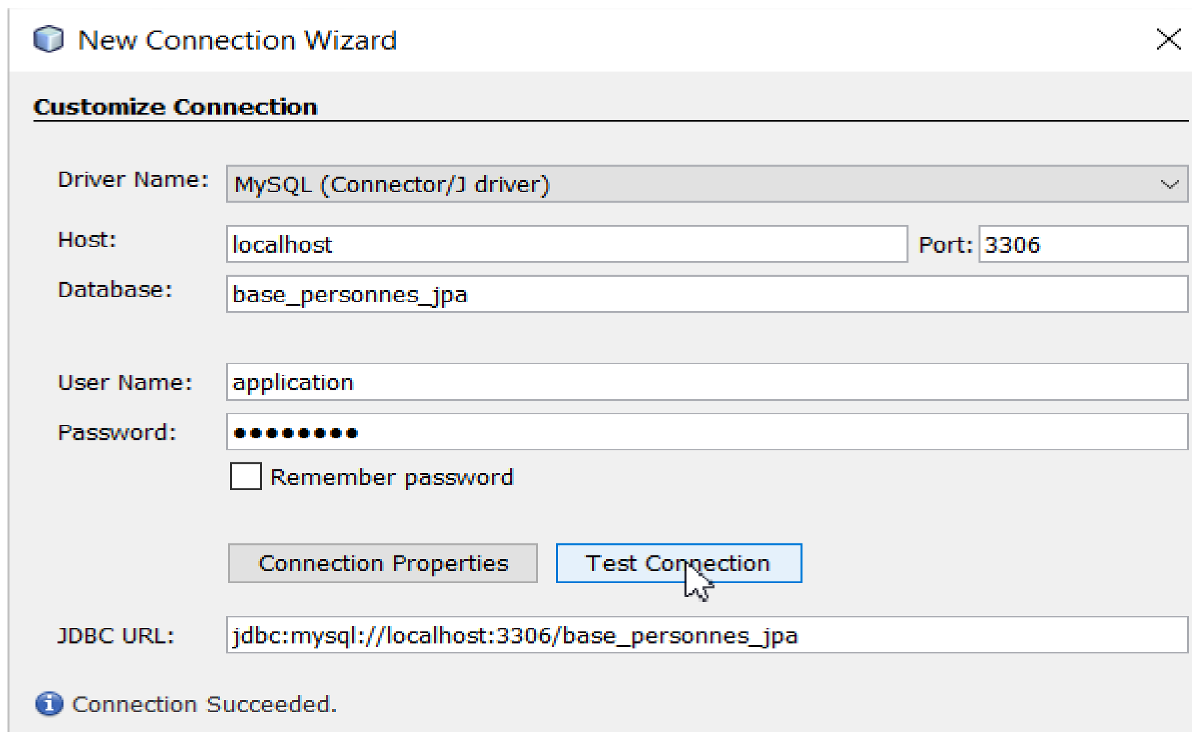
2. **Création connexion NetBeans** : créer une connexion Netbeans vers la base de données MySQL [base_personnes_jpa]

Regardons maintenant du coté de l'onglet « Service » pour créer la connexion à la base de données « **base_personnes_jpa** ».

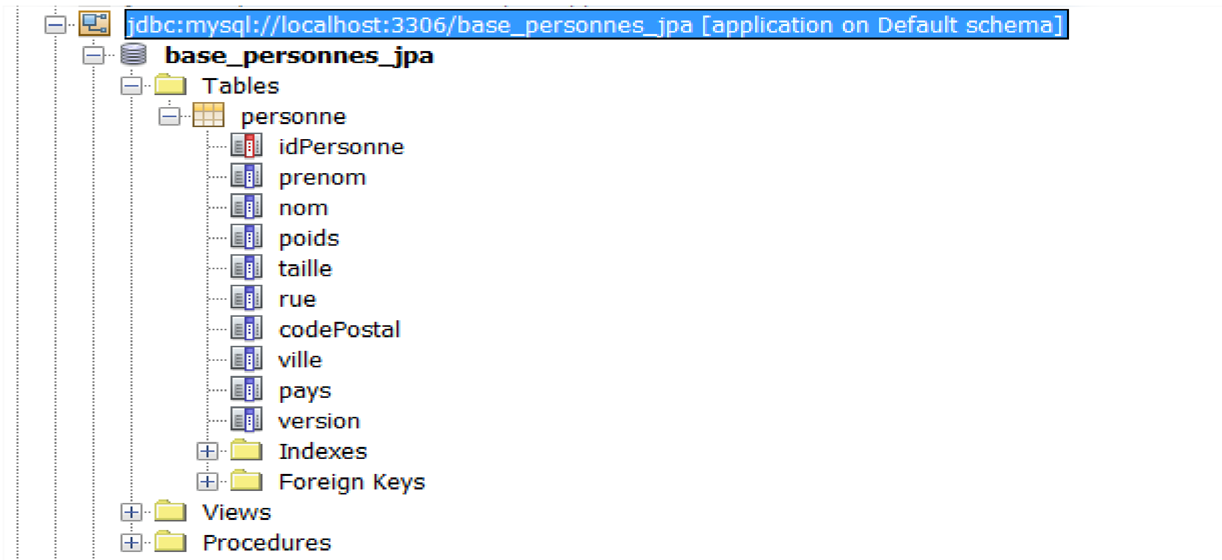
« Connect Using »→ « Test Connection » → « Finish »



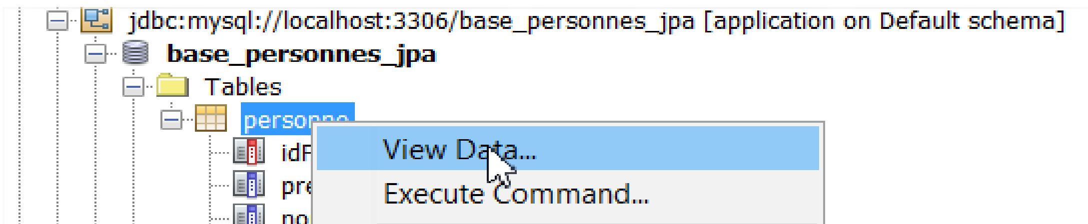
Remplir les caractéristiques de la base de données (user=**application** et mot de passe : **passw0rd**).



Après avoir créé la connexion, cliquez sur la base de données puis la table personne.



Le clic sur « view Data » permet de voir les données de la table « Personne ».



Connection: jdbc:mysql://localhost:3306/base_personnes_jpa [application on Default ...

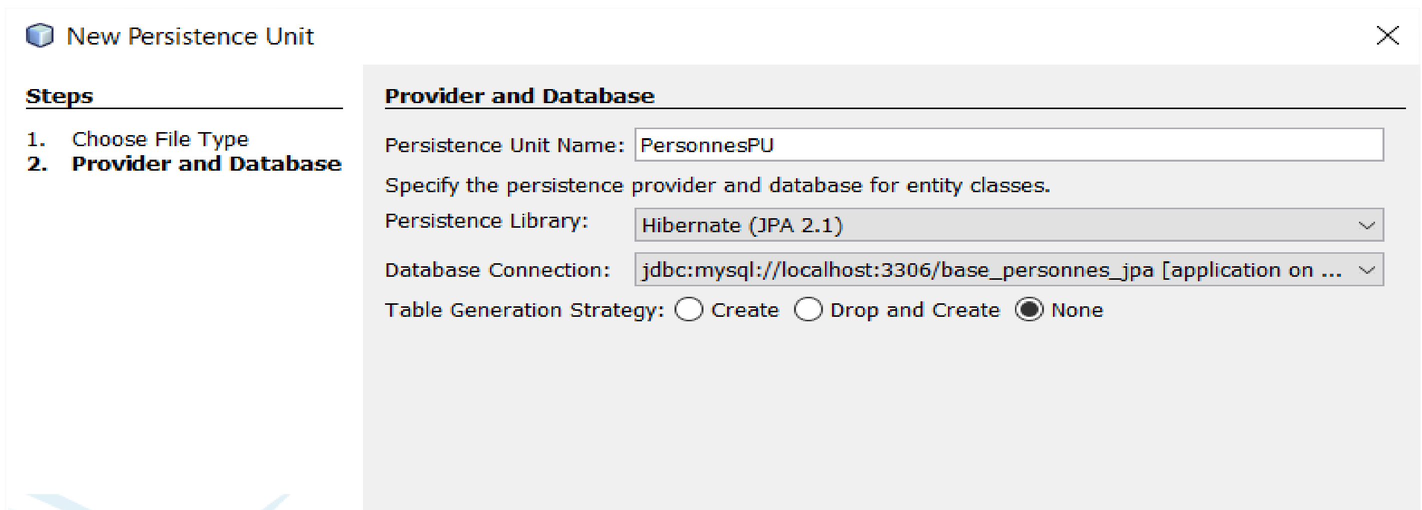
```
1   select * from personne;
2
```

select * from personne ×

Page Size: 20 Total Rows: 14 Page: 1 of 1

#	idPersonne	prenom	nom	poids
1	1	Julien	Marshall	55.0
2	2	Julien	Claire	85.0
3	3	Jacques	Dupont	62.0
4	4	Dupont	Dupont	62.0
5	5	Dupond	Dupond	62.0
6	6	Charles	Hallyday	100.0
7	7	Serge	Lama	87.0
8	8	Vincent	Thomas	64.0
9	9	Eric	Dummat	78.0
10	10	Nicolas	Samuel	112.0
11	11	Rémy	Guerry	96.0
12	12	Nicolas	Drapeau	87.0

3. **Création de l'unité de persistance** : Créer l'unité de persistance du JPA : « Clic droit »→ « New »→ « Persistence Unit ou Other → Persistence → Persistence Unit → « PersonnesPU »→ « Hibernate (JPA 2.1) »→ « Database connection » → « Selectionner la base de données « **base_personnes_jpa** » → « None »→« Finish ».



Le fichier **persistance.xml** qui servira à configurer la JPA sera :

```
<?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/persistenc
  <persistence-unit name="PersonnesPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.dao.entities.Personne</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/base_personnes_jpa"/>
      <property name="javax.persistence.jdbc.user" value="application"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="passw0rd"/>
    </properties>
  </persistence-unit>
</persistence>
```

En version copiable :

```
<?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.ejb.HibernatePersistence</provider>
    <class>com.dao.entities.Personne</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/base_personnes_jpa"/>
      <property name="javax.persistence.jdbc.user" value="application"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="passw0rd"/>
    </properties>
  </persistence-unit>
</persistence>
```

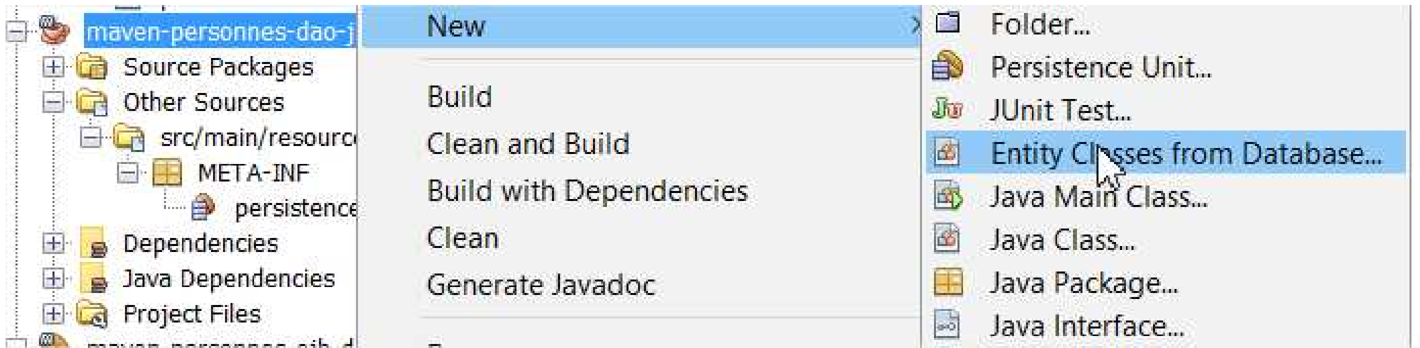
Ligne 2 : la balise racine du fichier XML est **persistence**.
Ligne 3 : **persistence-unit** sert à définir une unité de persistance. Il peut y avoir plusieurs unités de persistance. Chacune d'elles a un nom (attribut name) et un type de transactions (attribut transaction-type). L'application aura accès 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.ejb.HibernatePersistence**], 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.

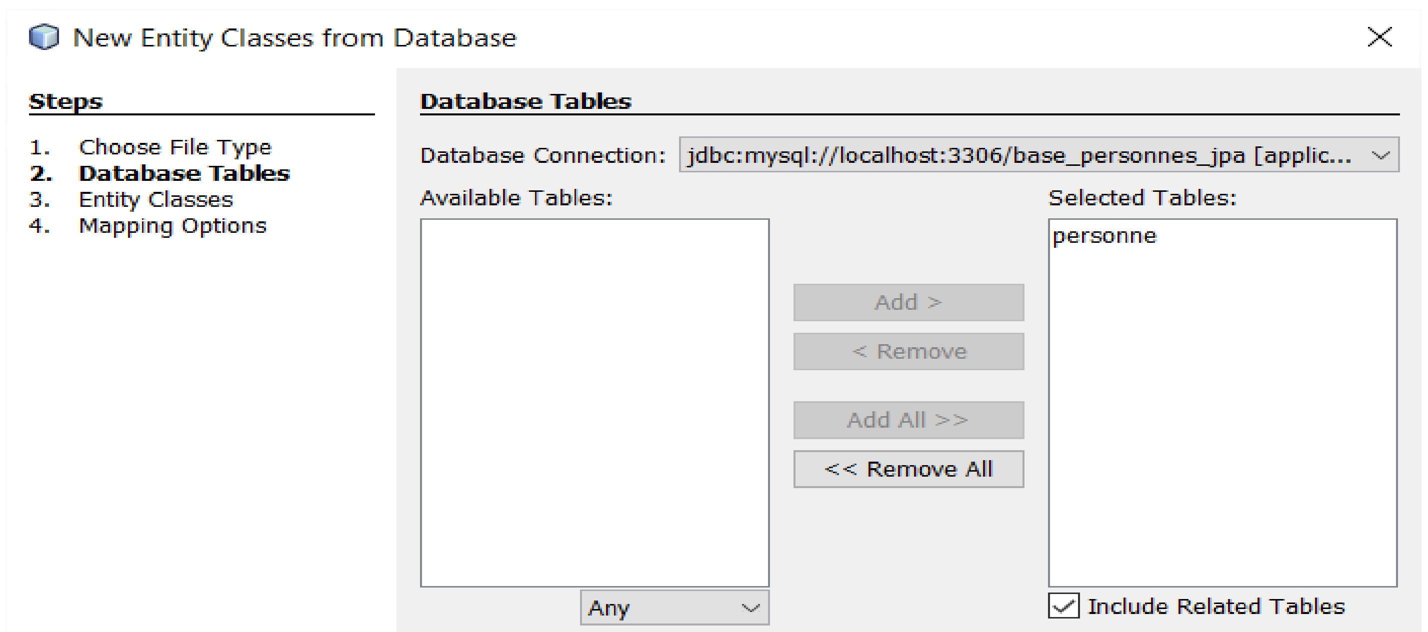
Ligne 5 : la balise **properties** introduit des propriétés propres au provider particulier choisi. Ainsi selon qu'on a choisi **Hibernate, EclipseLink, Toplink, Kodo**, ... on aura des propriétés différentes. Celles qui suivent sont propres à **Hibernate**.
Ligne 6 : l'url de la base de données utilisée
Ligne 7 : l'utilisateur de la connexion au SGBD.
Ligne 8 : le mot de passe de la connexion au SGBD.
Ligne 9 : la classe du pilote JDBC du SGBD, ici MySQL.

4. **Génération de la classe « com.dao.entities.Personne »** :

On peut maintenant générer la classe entité « Personne » objet image de la table « **com.dao.entities.Personne** » de la base « **base_personnes_jpa** ».
« Clic droit »→ « New »→ « Entity Classes from Database » oubien « Clic droit »→ « New »→ « Other »→ « Persistence »→ « Entity Classes from Database ».



Mettre la table « **Personne** » dans « **Selected Tables** » puis appuyez sur « Next ».



Mettre le package « **com.dao.entities** » puis appuyez sur « Next ».

New Entity Classes from Database

Steps

1. Choose File Type
2. Database Tables
3. **Entity Classes**
4. Mapping Options

Entity Classes

Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
personne	Personne	New

Project: maven-personnes-dao-jpa

Location: Source Packages

Package: com.dao.entities

- [x] Generate Named Query Annotations for Persistent Fields
- [x] Generate JAXB Annotations
- [ ] Generate MappedSuperclasses instead of Entities

Selectionner les collection de type « **java.util.List** » et l'associaion Fetch « default » puis « Finish ».

New Entity Classes from Database

Steps

1. Choose File Type
2. Database Tables
3. Entity Classes
4. **Mapping Options**

Mapping Options

Specify the default mapping options.

Association Fetch: default

Collection Type: java.util.List

- [ ] Fully Qualified Database Table Names
- [ ] Attributes for Regenerating Tables
- [x] Use Column Names in Relationships
- [ ] Use Defaults if Possible
- [ ] Generate Fields for Unresolved Relationships

La classe **Personne** générée sera :

```java

package com.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 ElHadji
 */
@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;
    }
```

```
97
98      public Double getPoids() {
99          return poids;
100     }
101
102     public void setPoids(Double poids) {
103         this.poids = poids;
104     }
105
106     public Double getTaille() {
107         return taille;
108     }
109
110     public void setTaille(Double taille) {
111         this.taille = taille;
112     }
113
114     public String getRue() {
115         return rue;
116     }
117
118     public void setRue(String rue) {
119         this.rue = rue;
120     }
121
122     public String getCodePostal() {
123         return codePostal;
124     }
125
126     public void setCodePostal(String codePostal) {
127         this.codePostal = codePostal;
128     }
129
130     public String getVille() {
131         return ville;
132     }
133
134     public void setVille(String ville) {
135         this.ville = ville;
136     }
137
138     public String getPays() {
139         return pays;
140     }
141
142     public void setPays(String pays) {
143         this.pays = pays;
144     }
145
146     public Integer getVersion() {
147         return version;
148     }
149
150     public void setVersion(Integer version) {
151         this.version = version;
152     }
153
154     @Override
155     public int hashCode() {
156         int hash = 0;
157         hash += (idPersonne != null ? idPersonne.hashCode() : 0);
158         return hash;
159     }
160
161     @Override
162     public boolean equals(Object object) {
163         // TODO: Warning - this method won't work in the case the id fields are not set
164         if (!(object instanceof Personne)) {
165             return false;
166         }
167         Personne other = (Personne) object;
168         if ((this.idPersonne == null && other.idPersonne != null) || (this.idPersonne != null && !this.idPersonne.equals(other.idPersonne))) {
169             return false;
170         }
171         return true;
172     }
173
174     @Override
175     public String toString() {
176         return "com.dao.entities.Personne[ idPersonne=" + idPersonne + " ]";
177     }
178
179 }
180
```

En version copiable :

```
package com.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;

@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")
```

```java
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(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;
}
```

```java
    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) {
        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.entities.Personne[ idPersonne=" + idPersonne + " ]";
    }
}
```


EL Hadji Gaye

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
<?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</groupId>
  <artifactId>maven-personnes-dao-jpa</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
    <hibernate.version>4.0.1.Final</hibernate.version>
    <mysql.version>5.1.41</mysql.version>
  </properties>
  <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>
     <!-- MySql dependencies -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysql.version}</version>
    </dependency>
  </dependencies>
</project>
```

5. **Création de la classe «** **com.main.MainJpa** » avec le contenu :

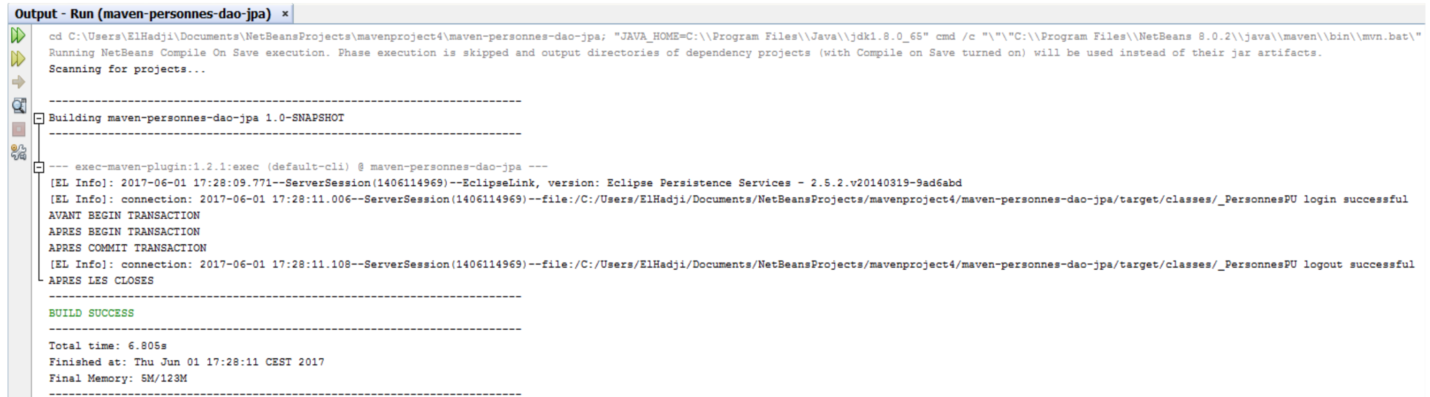
```
1   package com.main;
2
3   import com.dao.entities.Personne;
4   import java.util.List;
5   import javax.persistence.EntityManager;
6   import javax.persistence.EntityManagerFactory;
7   import javax.persistence.Persistence;
8
9   public class MainJpa {
10      private static final String persistenceUnit = "PersonnesPU";
11      public static void main(String[] args) {
12          String methodName = "main";
13          EntityManagerFactory emf = Persistence.createEntityManagerFactory(persistenceUnit);
14          EntityManager em = emf.createEntityManager();
15          System.out.println("AVANT BEGIN TRANSACTION");
16          em.getTransaction().begin();
17          System.out.println("APRES BEGIN TRANSACTION");
18          em.getTransaction().commit();
19          System.out.println("APRES COMMIT TRANSACTION");
20          // libération ressources
21          em.close();
22          emf.close();
23          System.out.println("APRES LES CLOSES");
24      }
25  }
```

En version copiable :

```
package com.main;

import com.dao.entities.Personne;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class MainJpa {
  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.

6. **Rajout des methodes** « **findAllPersonnes** », « **findPersonneById** » , « **createPersonne** », « **updatePersonne** » et « **removePersonne** » dans la classe « **com.main.MainJpa** ».

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 methode **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 methode **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 methode **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 methode **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 methode 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());
}
```

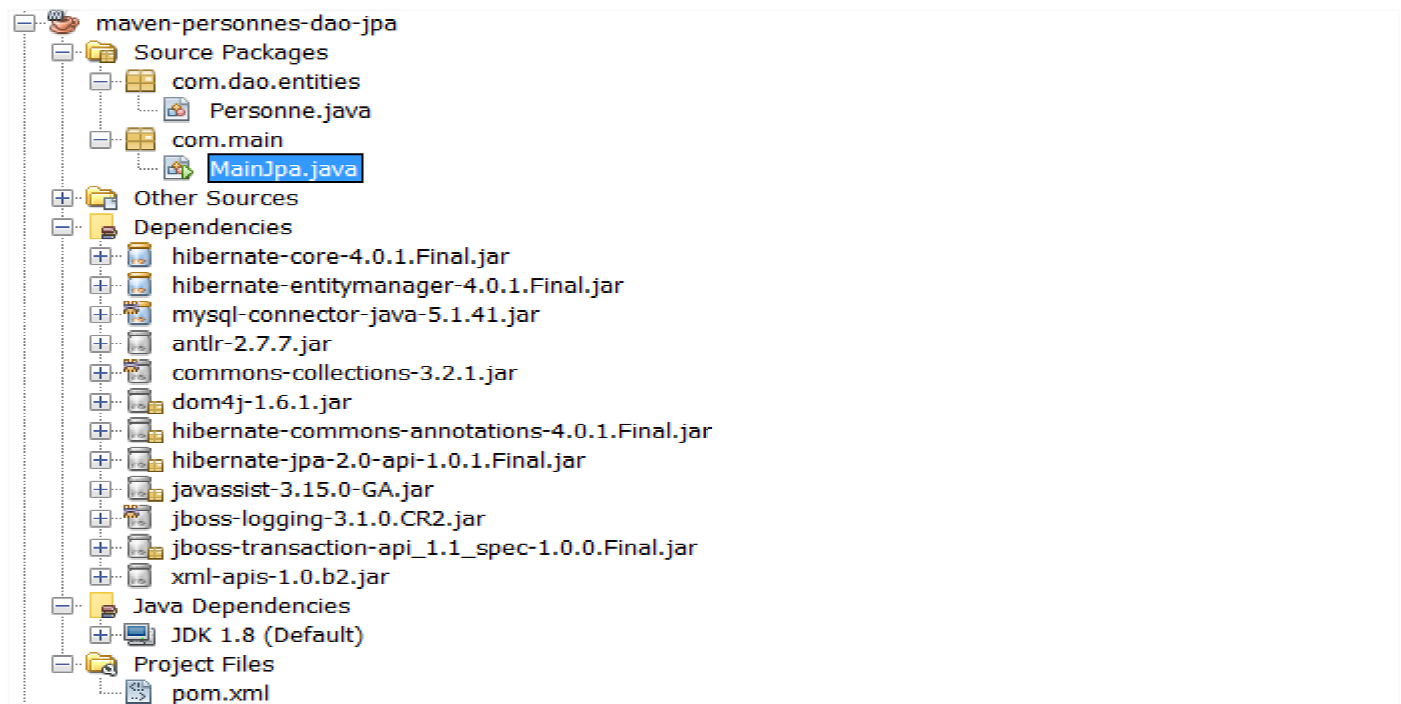
La methode main devient :

```
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());
}
```

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());
}
```

Le projet « **maven-personnes-dao-jpa** » deviendra donc :



A l’exécution de la methode 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>
```