Chapitre 4
Manipulation des données

1. Préparation

Afin de continuer sur ce chapitre, il faut avoir à disposition le fichier persistence.xml et les différentes entités créées dans le projet commencé à la section Premier lancement de NetBeans - Création du projet du chapitre Environnement de développement et continué tout au long du chapitre Préparation d'un projet. Pour cela, le fichier persistence.xml et les entités sont à remplacer par ceux trouvés dans l'exemple PROJET_ENI_MAVEN.zip téléchargeable sur le site des Éditions ENI.

Une fois les fichiers mis en place, il faut modifier les informations de connexion dans le fichier persistence.xml pour correspondre à vos paramètres de connexion à la base de données comme expliqué à la section Paramétrage de l'ORM - Les propriétés du fichier de persistance du chapitre Préparation d'un projet :

```
<property name="javax.persistence.jdbc.url" value="..."/>
<property name="javax.persistence.jdbc.user" value="..."/>
<property name="javax.persistence.jdbc.driver" value="..."/>
<property name="javax.persistence.jdbc.password" value="..."/>
```

Afin de voir les requêtes envoyées par l'ORM au système de gestion de base de données, il faut ajouter une propriété dans le fichier de persistance qui est propre à chaque implémentation, il n'y a pas de norme JPA pour ce paramètre.

Par exemple, pour Hibernate la propriété est :

```
<property name = "hibernate.show_sql" value = "true" />
```

2. Établissement de la connexion

Maintenant que le mapping des données a été réalisé, il n'y a plus qu'à se connecter à la base de données pour pouvoir les manipuler. C'est-à-dire les créer, les lire, les modifier ou les supprimer. Pour cela, il a été vu au chapitre Concept des ORM, que l'application doit obligatoirement interroger les données via un `EntityManager`, que cet `EntityManager` doit être fourni par un `EntityManagerFactory` qui lui-même connaît le mapping objet-relationnel (l'unité de persistance).

2.1 EntityManagerFactory

Pour créer l'`EntityManagerFactory`, il faut le nom de l'unité de persistance qui se trouve dans le fichier persistence.xml.

Par exemple, dans le fichier fourni, le nom est `projetEni` et se trouve à la ligne suivante :

```
<persistence-unit name="projetEni" transaction-type="RESOURCE_LOCAL">
```

L'`EntityManagerFactory` ne peut pas être instancié simplement car cela voudrait dire qu'il faut instancier l'`EntityManagerFactory` de l'implémentation de JPA, ce qui équivaudrait à cloisonner l'application et perdre le niveau d'abstraction que JPA permet.

Il a été vu que l'unité de persistance peut être paramétrée soit en mode `RESOURCE_LOCAL` soit en mode `JTA`. En mode `JTA`, l'unité de persistance est gérée par le conteneur JEE, il n'y a donc pas d'`EntityManagerFactory` à paramétrer au niveau du code source. En mode `RESOURCE_LOCAL`, elle doit être gérée par l'application. Deux solutions sont possibles selon s'il y a un conteneur JEE ou non.

2.1.1 Avec conteneur JEE, en RESOURCE_LOCAL

Lorsqu'un conteneur JEE est disponible, il est recommandé de laisser le serveur gérer l'`EntityManager`. Pour cela, il faut l'injecter dans le code source en utilisant l'annotation `@PersistenceUnit` juste avant la déclaration de l'`EntityManagerFactory`.

L'exemple ci-dessous montre comment configurer l'`EntityManagerFactory` avec l'unité de persistance `"projetEni"` :

```
@PersistenceUnit(unitName = "projetEni")
private EntityManagerFactory emf;
```

Cette utilisation de JPA, qui demande un conteneur JEE, n'est pas davantage détaillée dans ce chapitre et est donnée à titre informatif.

2.1.2 Sans conteneur JEE, en RESOURCE_LOCAL

Lorsqu'il n'y a pas de conteneur JEE (ou qu'il n'est pas utilisé pour JPA), il faut récupérer une instance d'`EntityManagerFactory`. Pour cela, il faut utiliser la classe `javax.persistence.Persistence` fournie par JPA.

Par exemple, la récupération d'un `EntityManagerFactory` pour l'unité de persistance `"projetEni"` s'écrit de la manière suivante :

```
Persistence.createEntityManagerFactory("projetEni");
```

Il a été vu qu'un `EntityManagerFactory` doit être unique durant la vie de l'application.

Par exemple, créer une classe utilitaire contenant cette instance unique permet de répondre à cette contrainte. Pour cela, il faut créer une classe `JpaUtil` dans le package `com.eni.jpa.util` :

```
public class JpaUtil {

    private static EntityManagerFactory emf = null;

    private JpaUtil() {
    }

    public static EntityManagerFactory getEmf() {
        if(emf == null){
            emf =
Persistence.createEntityManagerFactory("projetEni");
        }
        return emf;
    }
}
```

Comme les ressources sont gérées par l'application, il faut penser à les libérer avant de fermer l'application afin de fermer les connexions à la base de données. Pour cela, l'`EntityManagerFactory` possède la méthode `close()` qui permet de libérer les ressources.

Par exemple, comme une classe utilitaire `JpaUtil` a été créée, il suffit d'ajouter cette méthode et d'affecter `null` à la variable si jamais cette méthode est appelée en cours d'application pour pourvoir reconstruire l'`EntityManagerFactory`.

```
/**
 * Classe utilitaire pour JPA
 */
public class JpaUtil {

    /**
     * Singleton de l'EntityManagerFactory de l'application
     */
    private static EntityManagerFactory emf = null;

    /**
     * Permet de récupérer l'EntityManagerFactory de
     * l'application tout en le créant s'il n'existe pas
     *
     * @return l'EntityManagerFactory unique de l'application
     */
    public static EntityManagerFactory getEmf() {
        if(emf == null){
            emf = Persistence.createEntityManagerFactory("jpaTest");
        }
        return emf;
    }

    /**
     * Libère les ressources et détruit l'EntityManagerFactory
     * si jamais il faut le recréer.
     */
    public static void close(){
        if(emf!=null){
            emf.close();
            emf=null;
        }
    }

}
```

Finalement, avec cette classe l'initialisation de l'`EntityManagerFactory` se fait bien une seule fois lors du premier appel à la méthode `JpaUtil.getEmf()`, il est possible de le récupérer à tout moment de l'application et de libérer les ressources soit pour le recréer, soit lors de la fermeture de l'application.

En créant une simple méthode `main`, il est possible de tester dans un premier temps que les paramètres sont corrects :

```
    public static void main(String[] args) {
        //1
        System.out.println("Création de l'emf");
        JpaUtil.getEmf();
        //2
        System.out.println("Emf créé, nouvelle récupération de l'emf");
        JpaUtil.getEmf();
        //3
        System.out.println("fermeture de l'emf");
        JpaUtil.close();
        //4
        System.out.println("emf fermé, recréation d'un autre emf");
        JpaUtil.getEmf();
        //5
        System.out.println("fermeture de l'emf");
        JpaUtil.close();
        //6
        System.out.println("emf fermé, arrêt de l'application");
    }
```

Ce qui donne dans la console, lors de l'exécution de la première étape, le chargement de JPA et donc son implémentation Hibernate avec les paramètres saisis dans le fichier de persistance.

```
Création de l'emf
oct. 25, 2016 4:08:57 PM org.hibernate.jpa.internal.util.LogHelper
logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
            name: projetEni
            ...]
oct. 25, 2016 4:08:57 PM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.2.3.Final}
oct. 25, 2016 4:08:57 PM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
oct. 25, 2016 4:08:57 PM org.hibernate.cfg.Environment
buildBytecodeProvider
INFO: HHH000021: Bytecode provider name : javassist
```

```
oct. 25, 2016 4:08:57 PM
org.hibernate.annotations.common.reflection.java.JavaReflectionManager
<clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
oct. 25, 2016 4:08:57 PM org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl configure
WARN: HHH10001002: Using Hibernate built-in connection pool (not for
production use!)
oct. 25, 2016 4:08:57 PM org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001005: using driver [com.mysql.jdbc.Driver] at URL
[jdbc:mysql://localhost:3306]
oct. 25, 2016 4:08:57 PM org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001001: Connection properties: {user=root, password=****}
oct. 25, 2016 4:08:57 PM org.hibernate.engine.jdbc.connections.internal.
DriverManagerConnectionProviderImpl buildCreator
INFO: HHH10001003: Autocommit mode: false
oct. 25, 2016 4:08:57 PM
org.hibernate.engine.jdbc.connections.internal.PooledConnections <init>
INFO: HHH000115: Hibernate connection pool size: 20 (min=1)
Tue Oct 25 16:08:57 CEST 2016 WARN: Establishing SSL connection without
server's identity verification is not recommended. According to MySQL
5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established
by default if explicit option isn't set. For compliance with existing
applications not using SSL the verifyServerCertificate property is set to
'false'. You need either to explicitly disable SSL by setting useSSL=false,
or set useSSL=true and provide truststore for server certificate
verification.
oct. 25, 2016 4:08:57 PM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
```

Pour la deuxième étape, il n'y a rien de particulier car l'`EntityManagerFactory` a déjà été créé.

```
Emf créé, nouvelle récuperation de l'emf
```

Pour la troisième étape, la connexion avec la base de données est bien fermée.

```
fermeture de l'emf
oct. 25, 2016 4:08:57 PM
org.hibernate.engine.jdbc.connections.internal.DriverManager
ConnectionProviderImpl stop
INFO: HHH10001008: Cleaning up connection pool
[jdbc:mysql://localhost:3306]
```

Pour la quatrième étape, comme l'`EntityManagerFactory` a été détruit juste avant, il est recréé.