Mapping objet-relationnel en Java : JPA et Hibernate

Bernard Hugueney

Table des matières

1 Principe

Lorsque l'on implémente un CRUD à partir de JDBC, l'implémentation des DAO est extrêmement répétitive car toutes les entités doivent permettre les même fonctionnalités de base :

— création de nouvelle entités destinées à être enregistrée dans la table associée à la classe
— construction d'objets partir des lignes de la table associée à la classe
— mise à jour des lignes de la table pour prendre en compte les modifications des objets lorsque les attributs de ceux-ci ont été modifiés
— suppression des lignes correspondant à des objets que l'on veut supprimer.

Les principales différences entre deux classes entités, sont :

— le nom de la table associée à la classe
— les noms et types des colonnes associées à chaque attributs

Dans le cas un peu plus complexe de l'implémentation de relations entre entités, on doit aussi prendre en compte le type de relation (1-1, 1-plusieurs, plusieurs-1, plusieurs-plusieurs, et si elles sont unidirectionnelles ou bidirectionnelles) et identifier la table d'association éventuelle ainsi que décider si le chargement des entités associées doit être paresseux (lazy) ou non.

Si l'on pouvait *déclarer* ces *paramètres* l'implémentation de la correspondance entre les classes entités et les tables pourrait être automatisée. Il sera ainsi possible de manipuler les données de la base en désignant les classes et attributs au lieu des tables et colonnes correspondantes grâce à un *Domain Specific Language* adapté : Java Persistence Query Language (*JPQL*).

Le fait d'utiliser JPQL plutôt que SQL nous permettra aussi, dans un deuxième temps, de factoriser une partie des implémentations de nos *DAO*.

2 JPA et Hibernate

La correspondance entre les tables des Systèmes de Bases de Données Relationnels et les classes de la Programmation Orientée Objet permet de définir un *Mapping Objet-Relationnel*. En Java, JPA (Java Persistence API) est la spécification standard et Hibernate en est l'implémentation la plus populaire (même si EclipseLink est l'implémentation de référence). On a le même rapport entre spécification et implémentation qu'entre des *interfaces* et les classes implémentant ces interface (ces classes peuvent aussi implémenter plus que ce qui est spécifié par les interfaces).

Afin de ne pas dépendre d'une implémentation spécifique, on pourra vouloir se restreindre à n'utiliser que ce qui est spécifié par *JPA* même si l'on utilise *Hibernate*.

JPA constitue **encore** une couche d'abstraction. En tant que telle, elle ne sera pas vraiment utile pour de petits projets n'ayant pas besion d'évoluer. Le couplage avec le typage statique de Java procède des mêmes avantages et inconvénients.

3 Mise en œuvre

3.1 Dépendances

On peut utiliser *JPA* et *Hibernate* dans le cadre de *frameworks* (e.g. Spring boot, voire JHipster), mais dans un premier temps, on utilisera seulement/directement *JPA* et *Hibernate* dans un projet *Maven*.

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.2</version>
</dependency>

<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>javax.persistence</artifactId>
    <version>2.2.0</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.0.CR2</version>
</dependency>
```

3.2 Configuration XML

Dans la structure standard d'un projet *Maven*, on ajoutera un répertoire `META-INF` dans le répertoire `src/main/java` et dans ce répertoire un fichier `persistence.xml` :

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
            http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
  version="2.1">
```

```xml
<persistence-unit name="demo-jpa-1" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
                <class>co.simplon.patrimoine.model.City</class>
                <class>co.simplon.patrimoine.model.Monument</class>
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />

        <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost/postgres" /> <!-- !!! -->
        <property name="javax.persistence.jdbc.user" value="****" /> <!-- !!! -->
        <property name="javax.persistence.jdbc.password" value="****" /> <!-- !!! -->

        <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQL95Dialect"/> <!-- DB Dialect -->
        <property name="hibernate.hbm2ddl.auto" value="update" /> <!-- create / create-drop / update -->
        <property name="hibernate.show_sql" value="true" /> <!-- Show SQL in console -->
        <property name="hibernate.format_sql" value="true" /> <!-- Show SQL formatted -->
    </properties>

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

Exercice Que penser des propriétés `javax.persistence.jdbc.url` et surtout `javax.persistence.jdbc.u` voire `javax.persistence.jdbc.password` ? Que proposez-vous ?

Les autres propriétés de configuration de JPA pourraient elles aussi être exprimée en XML, dans un fichier orm.xml. Mais comme elles sont liées aux classes entités, on préférera les exprimer sous la forme d'*annotations*.

3.3 Annotations

Dans les classes `co.simplon.patrimoine.model.City` et `co.simplon.patrimoine.model.Monument`, on utilisera les annotations suivantes :

- sur la classe :
  - javax.persistence.Entity
  - javax.persistence.Table
- sur les attributs :
  - javax.persistence.Id pour l'attribut correspondant à la clé primaire
  - javax.persistence.GeneratedValue toujours pour la clé primaire. Avec une valeur de strategy à GenerationType.SEQUENCE, dans le cas d'une clé primaire de type `SERIAL` sous postgresql, notamment pour des raisons de performance.
  - javax.persistence.Column pour chacun des attributs.

Sur les classes suivantes :

```java
public class City {
    private Long id;
    private String name;
    private Double latitude;
    private Double longitude;

    public City() {
    }
    public City(String name, double latitude, double longitude) {
        this(null, name, latitude, longitude);
    }
    public City(Long id, String name, double latitude, double longitude) {
        this.id= id;
        this.name= name;
        this.latitude= latitude;
        this.longitude= longitude;
    }
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
```

```
    }

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

    public Double getLongitude() {
        return this.longitude;
    }

    public void setLongitude(Double longitude) {
        this.longitude = longitude;
    }

    public Double getLatitude() {
        return this.latitude;
    }

    public void setLatitude(Double latitude) {
        this.latitude = latitude;
    }

    @Override
    public String toString() {
        return "City [id=" + id + ", name=" + name + ", latitude=" + latitude
            + ", longitude=" + longitude + "]";
    }
}
```

Cette classe doit être liée à une table nommée `CITIES` avec des colonnes :

ID clé primaire de type `SERIAL`

NAME

LATITUDE

LONGITUDE

Exercice Indiquer que la valeur d'une colonne ne doit pas être `NULL` et qu'une chaîne de caractères doit avoir une taille limitée à 255 caractères ?

4 Utilisation

4.1 Construction d'un EntityManager

Au lieu d'utiliser directement des objets de type java.sql.Connection, on interagit désormais avec la base de données à travers des objets de type javax.persistence.EntityManager. Pour construire un tel objet en prenant en compte les propriétés définies dans le fichier `persistence.xml`, on utilise une javax.persistence.EntityManagerFactory. Lorsque l'on récupère cet objet *factory*, on indique le nom de la `persistence-unit` définie dans le fichier `persistence.xml` ainsi qu'une éventuelle table d'association qui permet de redéfinir certaines valeurs à l'exécution, par exemple les informations confidentielles :

```
String persistenceUnitName= "demo-jpa-1"; // defined in persistence.xml
Map<String, String> env = System.getenv();
Map<String, Object> configOverrides = new HashMap<String, Object>();
for (String envName : env.keySet()) {
  if (envName.contains("DB_USER")) {
    configOverrides.put("javax.persistence.jdbc.user", env.get(envName));
  }
  if (envName.contains("DB_PASS")) {
    configOverrides.put("javax.persistence.jdbc.password", env.get(envName));
  }
  if (envName.contains("DB_URL")) {
    configOverrides.put("javax.persistence.jdbc.url", env.get(envName));
  }
}
EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory(persistenceUnitName
                                            ,configOverrides);
```

(Si besoin, ajuster la configuration d'Eclipse pour qu'il reconnaisse le contenu du fichier persistence.xml).

4.2 Utilisation d'EntityManager

On peut utiliser l'objet de type `EntityManager` pour insérer un nouvel objet dans la table avec un appel à la méthode persist.

Exercice vérifier la valeur de l'attribut `id` avant et après l'appel à `persist`.

4.2.1 Création

```
1  public City createCity() {
2      EntityManager em= factory.createEntityManager();
3      City city= new City("Atlantis", 0, 0.5);
4      city= create(em, city);
5      em.close();
6      return city;
7  }
8  public City create(EntityManager em, City city) {
9      em.getTransaction().begin();
10     em.persist(city);
11     em.getTransaction().commit();
12     return city;
13 }
```

Une fois l'instance de la classe entité passée en argument à `persist`, celle-ci devient gérée (*managed*) par l'`EntityManager`. Ensuite, toutes modifications des attributs de l'objet effectuée avant l'appel à `commit` de l'`EntityManager` sera automatiquement répercutée :

```
1  public City createCityAndUpdate() {
2          EntityManager em= factory.createEntityManager();
3          City city= new City("Paris", 0, 0.5);
4          em.getTransaction().begin();
5          em.persist(city);
6          city.setLatitude(1000.);
7          em.getTransaction().commit();// MAGIC HAPPENS HERE !
8          em.close();
9          return city;
10 }
```

Exercice Observer le résultat de la gestion automatique dans la base de donnée.

4.3 Lecture

On peut lire directement une entité à partir de l'`EntityManager` à partir de la valeur de la clé primaire :

```
1  public City readCity() {
2      EntityManager em= factory.createEntityManager();
3      City city= readCity(em, 4L);
4      em.close();
5      return city;
6  }
7  public City readCity(EntityManager em, Long id) {
8      return em.find(City.class, id);
9  }
```

Exercice Que se passe-t-il si l'on change un attribut de l'objet lu ? Et si l'on effectue une transaction ensuite ?

4.4 Mise à jour

Lorsqu'on s'attend à ce qu'un objet soit déjà présent dans la base (l'attribut correspondant à la clé primaire doit donc avoir une valeur), et que l'on veut, le cas échéant récupérer une référence sur un objet géré par la base sans confier la gestion de l'objet passé en argument à l'`EntityManager`, on utilise `merge` plutôt que `persist`.

```java
public City updateCity() {
    return update(new City(4L,"PaRiS", -1., -2.));
}
public City update(City city) {
    EntityManager em= factory.createEntityManager();
    em.getTransaction().begin();
    city = em.merge(city);
    em.getTransaction().commit();
    return city;
}
```

Exercice Constater si l'instance retournée par `merge` est gérée (*managed*).

4.5 Suppression

On peut vouloir supprimer un objet selon deux cas de figures :
— à partir de la valeur de la clé primaire
— à partir d'une instance de la classe entité
Exercice Implémenter les deux cas de figure à l'aide de la méthode remove de l'`EntityManager`. Dans le deuxième cas de figure, prendre en compte que l'instance passé en argument doit être *gérée* par l'`EntityManager`.

5 Partique

Implémenter les mêmes fonctionnalités pour une classe `Monument` :

```java
public class Monument {
    private Long id;
    private String name;

    /* TODO
        private City city;
    */
    public Monument(String name) {
        super();
        this.name = name;
    }
    public Monument() {
    }
    public Long getId() {
        return this.id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    /*
    public City getCity() {
        return city;
    }

    public void setCity(City city) {
        this.city = city;
    }
    */
    @Override
    public String toString() {
        return "Monument [id=" + id + ", name=" + name
            + ", city=" /* city */+ "]";
    }
}
```

6 Associations

6.1 1-N, N-1

On va vouloir modéliser une association entre :
— un monument et une ville
— une ville et des monuments
Au niveau des entités, on peut ajouter des attributs (et accesseurs qui vont avec) :
— dans la classe `Monument` :

```
1  private City city;
```

— dans la classe `City` :

```
1  private List<Monument> monuments = new ArrayList<Monument>();
```

Remarque : On peut utiliser d'autres types de Collection que List.

Au niveau de la base de données, il suffirait d'avoir une colonne `city` (ou `fk_city` suivant la convention de nommage) comme clé étrangère.

On peut indiquer cela avec les annotations suivantes :

```
1  @ManyToOne(fetch = FetchType.LAZY)
2  @JoinColumn(name = "city")
3  private City city;
```

et

```
1  @OneToMany(mappedBy = "city")
2  private List<Monument> monuments = new ArrayList<Monument>();
```

Exercices Quel est l'effet de `fetch = FetchType.Lazy` ?

Quel seraient les effets du codes suivant ?

```
1  @OneToMany(mappedBy = "city", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
2  private Set<Monument> monuments;
```

Exercice Modifier la méthode `createMonument` du programme principal pour créer un monument qui soit rattaché à une ville.

6.2 N-N

On va ajouter une classe `User` qui permettra de modéliser des utilisateurs de notre application. Chaque utilisateur peut avoir visité plusieurs monuments et chaque monument peut avoir été visité par plusieurs utilisateurs.

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  import javax.persistence.Column;
5  import javax.persistence.Entity;
6  import javax.persistence.GeneratedValue;
7  import javax.persistence.GenerationType;
8  import javax.persistence.Id;
9  import javax.persistence.JoinTable;
10 import javax.persistence.JoinColumn;
11 import javax.persistence.ManyToMany;
12 import javax.persistence.Table;
13
14 @Entity
15 @Table(name = "USERS")
16 public class User {
17
18         @Id
```

```
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "ID")
        private Long id;

        @Column(name = "NAME", nullable = false, length = 100)
        private String name;

        @ManyToMany
        @JoinTable(name= "USER_MONUMENT",
                   joinColumns = {@JoinColumn(name = "FK_USER", referencedColumnName= "ID" ) },
                   inverseJoinColumns = { @JoinColumn(name = "FK_MONUMENT", referencedColumnName= "ID") })
        private Set<Monument> monuments = new HashSet<Monument>();

        public User() {
        }
        public User(String name) {
                this.name= name;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name= name;
        }
        public void addMonument(Monument m){
                monuments.add(m);
                m.getUsers().add(this);
        }
        public Set<Monument> getMonuments(){
                return monuments;
        }
        public void setMonuments(Set<Monument> monuments) {
                this.monuments= monuments;
        }
        public String toString() {
                return "User :{ id= "+id+"\n name= "+name+"\n nb momunents"+ monuments.size()+"\n}";
        }

}
```

Et en ajoutant dans la classe `Monument` l'attribut annoté suivant (et ses accesseurs) :

```
@ManyToMany(mappedBy="monuments")
private Set<User> users = new HashSet<User>();
```

Exercice Implémenter un méthode `createUser` .

7 Data Access Objects

Implementer les DAOs selon les interfaces suivantes :

```
public interface MonumentDao {
    Monument createMonument(Monument monument);
    Monument getMonumentById(Long id);
    Monument updateMonument(Monument monument);
    void deleteMonumentById(Long id);
}
```

```
public interface CityDao {
    City createCity(City city);
    City getCityById(Long id);
    City updateCity(City city);
    void deleteCityById(Long id);
}
```

```
public interface UserDao {
    User createUser(User user);
    User getUserById(Long id);
    User updateUser(User user);
```

```
5       void deleteUserById(Long id);
6   }
```

Exercices — Factoriser les interfaces avec une interface *générique*.
— Factoriser les implémentations avec une classe de base *générique*.

Bien sûr, les méthodes `find`, `persist`, `merge` et `remove` ne suffisent pas à interagir avec la base de données. Il est possible d'utiliser l'EntityManager pour effectuer des requêtes SQL avec la méthode createNativeQuery. Cependant, on pourra tirer un parti plus avantageux des correspondances classes / tables, attributs / colonnes, objets / lignes en écrivant des requêtes manipulant des classes, attributs et objets plutôt que des tables, colonnes et tuples avec un nouveau *DSL* (*Domain Specific Language*).

8 Java Persistence Query Language

JPQL reprend exactement les principes de SQL et l'on peut passer une `String` de code `JPQL` en argument à la méthode createQuery de l'objet `EntityManager` de ma même façon qu'on utilisait par exemple la méthode execute d'un objet `java.sql.Statement`. Ainsi, pour lister tous les monuments en les triant dans l'ordre alphabétique, le code JPQL sera :

```
1   " SELECT m FROM Monument m ORDER BY m.nom "
```

8.1 Paramétrage

Il est bien sûr aussi possible de paramétrer les requêtes *JQPL*. On peut utiliser deux types de paramètres :

positionnels Ils sont indiqué dans la requête *JPQL* sous la forme `?1`, `?2` ...

```
1   "SELECT c FROM City AS c WHERE c.name=?1"
```

nommés Ils sont indiqués dans la requête *JPQL* sous la forme `:nom` :

```
1   "SELECT c FROM City AS c WHERE c.name=:nameParam"
```

Un appel à la méthode setParameter prenant un `int` en premier argument ou à setParameter prenant une chaîne de caractères (le nom **sans** le préfixe ' :') en premier argument permet d'assigner une valeur à un paramètre avant d'exécuter la requête.

Exercice Utiliser des requêtes JPQL, notamment en explorant les opérateurs sur les chaînes de caractères plutôt qu'une simple égalité.

8.2 Typage

Plutôt que de récupérer des références de type `Object`, on préférera récupérer directement selon leur vrai type les instances de nos entités. On peut utiliser pour cela des objets de type TypedQuery :

```
1   TypedQuery<City> query = em.createQuery("SELECT c FROM City AS c WHERE c.name=:nameParam"
2                                           , City.class);
3   query.setParameter("nameParam", "Paris");
4   for (City c : query.getResultList()) {
5       System.out.println(c);
6   }
```

8.3 Définition statique

Grâce au paramétrage de requêtes, la plupart des requêtes peuvent être fixées à la compilation. Cela permet d'utiliser des *requêtes nommées* (*NamedQueries*) définies par des annotations. Par exemple :

```
@NamedQueries({
                @NamedQuery(name = "City.findAll", query = " SELECT c FROM City c ORDER BY c.name "),
                @NamedQuery(name = "City.deleteById", query = " DELETE FROM City c WHERE c.id = :id") })
```

Il est d'usage de situer ces annotations au niveau de la classe Entité qu'elles concernent (par exemple après les annotations `@Entity` et `@Table`). De même qu'il est d'usage d'utiliser le nom de la classe comme préfixe dans les noms des *requêtes nommées*.

On peut ensuite utiliser ces requêtes nommées de la façon suivante :

```
public List<City> findAll(int first, int size) {
    return entityManager.createNamedQuery("City.findAll", City.class)
        .setFirstResult(first).setMaxResults(size).getResultList();
}
```

Exercices — Implémenter des méthodes `findAll` avec des *requêtes nommées* pour les classes `Monument` et `User`.

— Implémenter des méthodes `deleteById` avec des *requêtes nommées* pour les classes `City`, `Monument` et `User`.

9 Mise en œuvre

Implémenter un CRUD en utilisant *JPA* ! Remarquer qu'on a les mêmes problèmes de durées de vie/partage pour les objets de type `EntityManager` que pour les objets de type `Connection`.