Java Persistence API (JPA)

1 Introduction

La technologie JPA (**Java Persistence API**[1]) a pour objectif d'offrir un modèle d'ORM (**Object Relational Mapping**) indépendant d'un produit particulier (comme Hibernate, TopLink, etc.). Cette technologie est basée sur

- un jeu d'interfaces et de classes permettant de séparer l'utilisateur d'un service de persistance (votre application) et le fournisseur d'un service de persistance,
- un jeu d'annotations pour préciser la mise en correspondance entre classes Java et tables relationnelles,
- un fournisseur de persistance (par exemple Hibernate),
- un fichier XML `persistence.xml` décrivant les moyens de la persistance (fournisseur, *datasource*, etc.)

Cette technologie est utilisable dans les applications Web (conteneur Web), **ou** dans les EJB (serveur d'applications) **ou bien** dans les applications standards (Java Standard Edition). C'est ce dernier cas que nous allons étudier.

Quelques liens :

- API JPA 2.0 (JEE 8)[2]
- Tutorial JPA (JEE 7)[3]
- Une documentation sur JPQL[4]
- La page de Wikipedia[5]
- De très bons exemples[6]

2 Mise en place

2.1 Projet Eclipse

Pour la mise en pratique de JPA, nous allons utiliser le *framework* Hibernate[7] qui va nous servir de fournisseur de persistance. Plus précisement, suivez les étapes ci-dessous :

- Reprenez le projet Eclipse utilisé pour le TP JDBC (séance précédente).
- Ajoutez au fichier `pom.xml` les dépendances nécessaires (nous allons utiliser le *Starter DATA JPA* de Spring Boot pour automatiser l'inclusion des librairies nécessaires) :

```
<!-- pour utiliser Spring DATA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- pour utiliser Hibernate -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
</dependency>
```

1. https ://www.oracle.com/technetwork/java/javaee/tech/
2. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/package-summary.html
3. https://docs.oracle.com/javaee/7/tutorial/partpersist.htm
4. http://download.oracle.com/docs/cd/E13189_01/kodo/docs40/full/html/ejb3_overview_query.html
5. http://en.wikipedia.org/wiki/Java_Persistence_API
6. http://www.java2s.com/Tutorial/Java/0355__JPA/Catalog0355__JPA.htm
7. http://www.hibernate.org/

- Préparez deux packages : `myapp.jpa.model` pour les JavaBeans et `myapp.jpa.dao` pour le service DAO (**Data Access Object**).
- Créez le répertoire `src/main/resources/META-INF` et le fichier de configuration JPA ci-dessous :

src/main/resources/META-INF/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_2.xsd"
    version="2.2">
    <persistence-unit name="myBase" transaction-type="RESOURCE_LOCAL">
        <properties>
            <!-- partie JPA générique -->
            <property name="javax.persistence.jdbc.driver"
                value="org.hsqldb.jdbcDriver" />
            <property name="javax.persistence.jdbc.url"
                value="jdbc:hsqldb:file:databases/myBase" />
            <property name="javax.persistence.jdbc.user" value="SA" />
            <property name="javax.persistence.jdbc.password" value="" />

            <!-- partie spécifique Hibernate -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect" />
        </properties>
    </persistence-unit>

</persistence>
```

- **Remarque 1** : Pour s'adapter au SGBDR, Hibernate utilise une couche logicielle spécifique à chaque SGBDR. C'est le *dialect*[8].
- **Remarque 2** : La propriété `hibernate.hbm2ddl.auto` demande à Hibernate de générer automatiquement les tables nécessaires au bon fonctionnement de la couche JPA (à noter que la valeur `create-drop` est aussi intéressante)[9].

2.2 Une première entité

Pour construire notre exemple, nous allons créer une entité personne (classe `Person`) :

8. http ://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html/#database-dialect
9. http ://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html/#configurations-hbmddl

```java
package myapp.jpa.model;

import java.util.Date;

import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.PostUpdate;
import javax.persistence.PreUpdate;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Transient;
import javax.persistence.Version;

import lombok.Data;
import lombok.NoArgsConstructor;

@Entity(name = "Person")
@Data
@NoArgsConstructor
public class Person {

    @Id()
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Basic(optional = false)
    private String firstName;

    @Basic()
    @Temporal(TemporalType.DATE)
    private Date birthDay;

    @Version()
    private long version = 0;

    @Transient
    public static long updateCounter = 0;

    public Person(String firstName, Date birthDay) {
        super();
        this.firstName = firstName;
        this.birthDay = birthDay;
    }

    @PreUpdate
    public void beforeUpdate() {
        System.err.println("PreUpdate␣of␣" + this);
    }

    @PostUpdate
    public void afterUpdate() {
        System.err.println("PostUpdate␣of␣" + this);
        updateCounter++;
    }

}
```

Explications : Il y a beaucoup de choses à expliquer sur cet exemple. Commençons par étudier les annotations :

`@Entity` Cette annotation indique que la classe est un *EJB Entity* qui va représenter les données de la base

de données relationnelle. Vous pouvez fixer un nom (attribut `name`) qui, par défaut, est celui de la classe. [`JavaDoc` [10]]

`@Id` Cette annotation précise la propriété utilisée comme clef primaire. Cette annotation est obligatoire pour un *EJB Entity*. [`JavaDoc` [11]]

`@GeneratedValue` Cette annotation spécifie la politique de construction automatique de la clef primaire. [`JavaDoc` [12]]

`@Basic` C'est l'annotation la plus simple pour indiquer qu'une propriété est persistante (c'est-à-dire gérée par JPA). A noter que le chargement retardé d'une propriété est possible avec l'attribut `fetch=FetchType.LAZY`. [`JavaDoc` [13]]

`@Temporal` A utiliser pour le *mapping* des types liés au temps (`java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp` et `java.util.Calendar`). [`JavaDoc` [14]]

`@Transient` Indique que la propriété n'est pas persistante. [`JavaDoc` [15]]

`@Version` Indique la propriété à utiliser pour activer et gérer la version des données. Cette capacité est notamment utile pour implémenter une stratégie de concurrence optimiste. [`JavaDoc` [16]]

`@PreUpdate/@PostUpdate` Deux exemples de *callback*. Voilà la liste des évènements récupérables : `PostLoad` [17], `PostPersist` [18], `PostRemove` [19], `PostUpdate` [20], `PrePersist` [21], `PreRemove` [22].

2.3 Le service DAO

Continuons en créant la classe de service qui va se charger des opération d'E/S.

10. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Entity.html
11. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Id.html
12. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/GeneratedValue.html
13. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Basic.html
14. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Temporal.html
15. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Transient.html
16. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Version.html
17. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PostLoad.html
18. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PostPersist.html
19. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PostRemove.html
20. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PostUpdate.html
21. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PrePersist.html
22. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/PreRemove.html

```java
package myapp.jpa.dao;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceException;

import org.springframework.stereotype.Service;

import myapp.jpa.model.Person;

@Service
public class JpaDao {

    private EntityManagerFactory factory = null;

    @PostConstruct
    public void init() {
        factory = Persistence.createEntityManagerFactory("myBase");
    }

    @PreDestroy
    public void close() {
        if (factory != null) {
            factory.close();
        }
    }

    /*
     * Ajouter une personne
     */
    public Person addPerson(Person p) {
        EntityManager em = null;
        try {
            em = factory.createEntityManager();
            em.getTransaction().begin();
            em.persist(p);
            em.getTransaction().commit();
            return p;
        } finally {
            closeEntityManager(em);
        }
    }

    /*
     * Charger une personne
     */
    public Person findPerson(long id) {
        EntityManager em = null;
        try {
            em = factory.createEntityManager();
            em.getTransaction().begin();
            Person p = em.find(Person.class, id);
            em.getTransaction().commit();
            return p;
        } finally {
            closeEntityManager(em);
        }
    }

    /*
     * Fermeture d'un EM (avec rollback éventuellement)
     */
    private void closeEntityManager(EntityManager em) {
        if (em == null || !em.isOpen())
```

Explication 1 : Lors de l'initialisation, la classe `Persistence` [23] est utilisée pour analyser les paramètres de connection (fichier `persistence.xml`) et trouver l'unité de persistance passée en paramètre (*myBase* dans cet exemple). A l'issue de cette étape nous récupérons une instance de l'interface `EntityManagerFactory` [24]. Cette usine, qui est généralement un singleton, nous permettra, dans un deuxième temps, d'ouvrir des connections vers la base de données.

Explication 2 : Pour agir sur les entités (`addPerson`) nous devons récupérer à partir de l'usine une instance de l'interface `EntityManager` [25]. Celle-ci va permettre les opérations CRUD de persistance sur les entités (Create, Read, Update et Delete). Un `EntityManager` [26] ne supporte pas le *multi-threading*. Il doit donc être créé et détruit à chaque utilisation par un *thread*. Cette création est une opération peu coûteuse qui peut se reproduire un grand nombre de fois (contrairement à l'usine).

2.4 Test du service DAO

Créez une classe de test unitaire en vous basant sur **Junit 5** :

```java
package myapp.jpa;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import myapp.jpa.dao.JpaDao;
import myapp.jpa.model.Person;

@SpringBootTest
public class TestJpaDao {

    @Autowired
    JpaDao dao;

    @Test
    public void addAndFindPerson() {
        // Création
        var p1 = new Person("Jean", null);
        p1 = dao.addPerson(p1);
        assertTrue(p1.getId() > 0);
        // relecture
        var p2 = dao.findPerson(p1.getId());
        assertEquals("Jean", p2.getFirstName());
        assertEquals(p1.getId(), p2.getId());
    }

}
```

`@Autowired` : La classe Dao est instanciée une seule fois car la création d'une `EntityManagerFactory` [27] est une opération très coûteuse. C'est typiquement une opération réalisée à l'initialisation de l'application.

Travail à faire :

- Exécutez ce test et vérifiez son bon fonctionnement (notamment en analysant les requêtes SQL utilisées par Hibernate et affichées sur la console).

23. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Persistence.html
24. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManagerFactory.html
25. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManager.html
26. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManager.html
27. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManagerFactory.html

- Vérifiez par un test unitaire que le nom ne peut pas être `null`.

2.5 Compléter le service DAO

Vous pouvez maintenant implémenter les méthodes ci-dessous en utilisant les méthodes de l'interface `EntityManager` [28] (notamment `merge` pour la mise à jour et `remove` pour la suppression). **Attention** : pour la suppression, vous devez au prélable charger l'entité à supprimer avec `em.find(...)`.

```
public void updatePerson(Person p) {
   ...
}

public void removePerson(long id) {
   ...
}
```

Travail à faire : Enrichir la classe de test unitaire pour créer une personne, la recharger, la modifier et finalement la détruire.

2.6 Configurer les noms SQL

Motivation : Pour l'instant, nous laissons JPA/Hibernate générer automatiquement les noms des tables et des colonnes à partir des noms de classes et de propriétés. Cette approche n'est pas possible si la base de données **existe déjà**.

L'annotation `@Table` [29] permet de préciser le nom de la table associée à une classe ainsi que d'ajouter des conditions d'unicité. Dans l'exemple ci-dessous, nous ajoutons une contrainte d'unicité sur le couple (prénom, date de naissance).

L'annotation `@Column` [30] permet préciser la correspondance entre colonne d'une table et propriété d'une classe et d'imposer des contraintes supplémentaires.

28. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManager.html
29. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Table.html
30. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Column.html

```
...

@Entity(name = "Person")

@Table(name = "TPerson",
   uniqueConstraints = {
      @UniqueConstraint(columnNames = {
         "first_name", "birth_day"
      })
   })

public class Person {

    ...

    // propriété à modifier (ajouter @Column)
    @Basic(optional = false, fetch = FetchType.EAGER)
    @Column(name = "first_name", length = 200)
    private String firstName;

    // propriété à ajouter
    @Basic(optional = true, fetch = FetchType.EAGER)
    @Column(name = "second_name", length = 100, nullable = true, unique = true)
    private String secondName;

    // propriété à modifier (ajouter @Column)
    @Basic()
    @Temporal(TemporalType.DATE)
    @Column(name = "birth_day")
    private Date birthDay;

    ...

}
```

Travail à faire :

- Ajoutez un test unitaire afin de valider la contrainte d'unicité sur la table.
- Ajoutez un test unitaire afin de vérifier la contrainte d'unicité du second nom.

3 Utiliser Spring avec JPA

Vous avez remarqué que les méthodes de la classe Dao contiennent beaucoup de code technique (création/fermeture d'un EM, ouverture/fermeture d'une transaction). Nous allons maintenant utiliser Spring pour nous aider à simplifier l'utilisation de JPA. Suivez les étapes ci-dessous :

- Préparez une classe de configuration pour Spring Boot :

```
package myapp.jpa.dao;

import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.Configuration;

import myapp.jpa.model.Person;

@Configuration
@EntityScan(basePackageClasses = Person.class)
public class SpringDaoConfig {

}
```

- Ajoutez à votre fichier de configuration Spring Boot (c'est-à-dire `application.properties`) les paramètres ci-dessous :

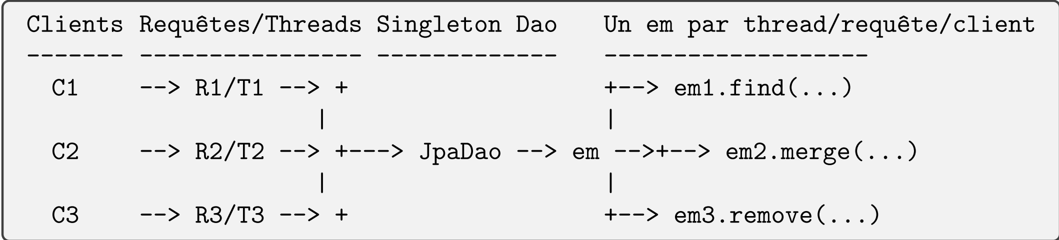
```
spring.datasource.url=jdbc:hsqldb:file:databases/myBase
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.driver-class-name=org.hsqldb.jdbc.JDBCDriver

spring.jpa.hibernate.ddl-auto = create-drop
spring.jpa.hibernate.dialect = org.hibernate.dialect.HSQLDialect
spring.jpa.hibernate.show_sql = true
spring.jpa.hibernate.format_sql = true
```

- Vous pouvez maintenant, dans la classe `JpaDao`, remplacer la création/fermeture des `EntityManager` [31] par l'injection d'un instance unique prévue pour fonctionner dans un environnement multi-threads :

```java
package myapp.jpa.dao;

...

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

@Repository
@Transactional
public class JpaDao {

    @PersistenceContext
    EntityManager em;

    public Person addPerson(Person p) {
        em.persist(p);
        return (p);
    }

    ....

}
```

Explication 1 : Les annotations `@Repository` et `@Transactional` permettent d'indiquer que la classe va manipuler des données et que chaque méthode doit être exécutée dans une transaction.

Explication 2 : L'annotation `@PersistenceContext` réclame l'injection d'un EM. Cette instance est capable de gérer en même temps plusieurs transactions associées à plusieurs threads s'exécutant en parallèle. Les transactions sont démarrées avant l'exécution des méthodes et stoppées après. L'entity manager injecté par Spring va agir comme un Proxy d'aiguillage qui va associer un EM réel au thread courant, c'est-à-dire à la requête courante, c'est-à-dire au client courant. Cette solution respecte le principe **un thread est associé à chaque requête et un EM à chaque thread**.



```
Clients Requêtes/Threads Singleton Dao    Un em par thread/requête/client
------- ---------------- -------------    -------------------
  C1    --> R1/T1 --> +                   +--> em1.find(...)
                      |                   |
  C2    --> R2/T2 --> +---> JpaDao --> em -->+--> em2.merge(...)
                      |                   |
  C3    --> R3/T3 --> +                   +--> em3.remove(...)
```

31. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManager.html

Travail à faire : Terminez l'adaptation des méthodes de la classe `JpaDao`.

Travail à faire : Pour tester la gestion des transactions, écrivez un test pour appeler la méthode `addPerson` sur la même personne dans deux threads qui s'exécutent en parallèle. Une insertion doit fonctionner et l'autre pas.

4 Les requêtes en JPA

JPA fournit le langage JPQL afin de construire des requêtes. Il est **important de noter** que JPQL s'exprime sur les classes et les propriétés et non pas sur les tables et les colonnes. En d'autres termes, JPQL travaille sur le modèle objet et pas sur le modèle relationnel. JPQL assure ainsi une **indépendance complète** de l'application vis-à-vis de la base de données et vis-à-vis du SGBDR utilisé.

4.1 Requêtes simples

Ajoutez à votre service DAO une méthode de listage des personnes :

```
public List<Person> findAllPersons() {
    String query = "SELECT p FROM Person p";
    TypedQuery<Person> q = em.createQuery(query, Person.class);
    return q.getResultList();
}
```

Travail à faire : En utilisant la JavaDoc de l'interface `Query` [32] et cette présentation [33] du langage JPQL (très proche de SQL), proposez une version permettant de sélectionner les personnes à partir de leur prénom (clause `like`). Pour ce faire, vous utiliserez des requêtes paramétrées (format `:nom-du-paramètre`) et la méthode `setParameter` de la requête typée pour fixer la valeur de paramètre.

```
public List<Person> findPersonsByFirstName(String pattern) {
    ...
}
```

4.2 Requêtes nommées

Vous pouvez aussi utiliser l'annotation `NamedQuery` [34] pour placer les requêtes dans les classes d'entité et y faire référence.

Travail à faire : Créez une requête particulière par l'annotation `NamedQuery` [35] dans la classe `Person` et utilisez-la grace à la méthode `createNamedQuery` (version typée JPA 2) de l'interface `EntityManager` [36]. C'est une solution simple pour délocaliser les requêtes dans les classes entité et simplifier les DAO.

4.3 Création d'objets

Nous avons souvent besoin de récupérer une partie des propriétés d'une entité. Par exemple le numéro et le prénom de chaque personne pour dresser une liste. Pour ce faire, nous devons créer un javaBean `myapp.jpa.model.FirstName` qui représente les informations voulues et utiliser la requête suivante :

```
SELECT new myapp.jpa.model.FirstName(p.id,p.firstName)
FROM Person p
```

32. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/TypedQuery.html
33. http://download.oracle.com/docs/cd/E13189_01/kodo/docs40/full/html/ejb3_overview_query.html
34. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/NamedQuery.html
35. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/NamedQuery.html
36. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/EntityManager.html

Travail à faire : Créez le nouveau JavaBean `FirstName` ci-dessous et ajoutez une méthode qui renvoie la liste des prénoms et testez-la.

```
package myapp.jpa.model;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class FirstName {

    private long id;
    private String firstName;

}
```

5 Les classes embarquées

Les classes embarquées permettent de coder plus facilement des relations un-un comme par exemple, le fait que chaque personne doit avoir une et une seule adresse. Dans ce cas il est plus simple de placer l'adresse dans la table qui code les personnes sans être obligé de créer une nouvelle table.

Définissons la classe `Address` et indiquons qu'elle peut être embarquée avec l'annotation `Embeddable` [37] :

```
package myapp.jpa.model;

import javax.persistence.Embeddable;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Embeddable
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Address {

    private String street;
    private String city;
    private String country;

}
```

Nous pouvons maintenant embarquer cette adresse dans la classe `Person` en utilisant l'annotation `Embedded` [38]. Nous retrouverons dans la table `TPerson` les colonnes nécessaires pour coder la rue, la ville et le pays.

```
    ...

    @Embedded
    private Address address;

    ...
```

Complément : Si nous voulions associer deux adresses à une seule personne, alors nous devrions changer le nom des colonnes qui codent la rue, la ville et le pays de la deuxième adresse. Pour ce faire, nous pouvons utiliser

37. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Embeddable.html
38. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Embedded.html

l'annotation `AttributeOverrides` [39] (voir `Embedded` [40]).

39. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/AttributeOverrides.html
40. https://javaee.github.io/javaee-spec/javadocs/javax/persistence/Embedded.html