JPA : Java Persistence API

P.Mathieu

IUT de Lille
http://www.iut-a.univ-lille.fr
prenom.nom@univ-lille.fr

Plan

Université de Lille

Principe général

Présentation

Université de Lille

- JPA : Java Persistence API
- JPA est une norme, une spécification qui impose à un ORM un fonctionnement précis
- JPA fournit les bases d'un framework respectant un `Design Pattern DAO`
- Définit un mapping Objet-Relationnel assurant la persistance des objets métier
- Fournit quelques requêtes génériques (`find`, `persist`, `remove`,...)
- Fournit le langage JPQL (Java Persistance Query Language)
- Fonctionne à partir d'annotations : `Entity, Id, ...`
- JPA est défini dans le package `jakarta.persistence`

Principe général

Implémentations

Université de Lille

Plusieurs ORM implémentent JPA

- EclipseLink (implémentation de référence)
- Hibernate (la plus connue)
- OpenJPA
- TopLink
- DataNucleus
- OrmLite, Jdbi, JEasyOrm, ...

Deux jars sont necessaires : l'un pour JPA, l'autre pour son implémentation

Principe général

Configuration MAVEN

Université de Lille

```
<project>
 .....
<dependencies>

    <!-- JPA -->
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>org.eclipse.persistence.jpa</artifactId>
      <version>4.0.3</version>
      </dependency>
      <!-- EclipseLink -->
    <dependency>
      <groupId>org.eclipse.persistence</groupId>
      <artifactId>eclipselink</artifactId>
      <version>4.0.3</version>
    </dependency>

</dependencies>
 ....
</project>
```

Principe général

Etapes principales

Université de Lille

Le principe général est très simple :

1. On définit le système de persistance dans un fichier XML `resources/META-INF/persistence.xml`
2. On crée les POJO des entités avec les bonnes annotations
3. Dans les programmes, on utilise un gestionnaire d'entités (`EntityManager`) qui gère la persistance et permet de manipuler les objets via le CRUD

On ne s'occupe plus de la BDD mais uniquement des objets

Principe général

Une unité de persistance minimale

Université de Lille

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.0" xmlns="http://xmlns.jcp.org/xml/ns/persistence" ... >
  <persistence-unit name="testjpa" transaction-type="RESOURCE_LOCAL">
    <class>fr.but3.Client</class>
    ...
    <class>fr.but3.Fournisseur</class>
    <properties>
        <property name="jakarta.persistence.jdbc.url"
                  value="jdbc:postgresql://lamachine/labase"/>
        <property name="jakarta.persistence.jdbc.user" value="..."/>
        <property name="jakarta.persistence.jdbc.password" value="..."/>
        <property name="jakarta.persistence.jdbc.driver" value="..."/>
    </properties>
  </persistence-unit>
</persistence>
```

Principe général

Arborescence JPA + MAVEN

Université de Lille

```
projet
|-- pom.xml
|-- run.sh
|-- src
    |-- main
        |-- java
        |    |-- fr
        |        |-- but3
        |            |-- App.java
        |-- resources
            |-- META-INF
                |-- persistence.xml
```

Le `META-INF/persistence.xml` doit etre placé dans `resources`

Principe général

Arborescence WEB + JPA + MAVEN

Université de Lille

```
projet
|-- pom.xml
|-- run.sh
|-- src
    |-- main
        |-- java
        |    |-- fr
        |        |-- but3
        |            |-- Servlet1.java
        |-- resources
        |    |-- META-INF
        |         |-- persistence.xml
        |-- webapp
             |--META-INF
             |     |-- context.xml
             |-- WEB-INF
             |     |-- web.xml
             |-- page1.jsp
```

Le `META-INF/persistence.xml` doit être placé dans `resources`
Le `META-INF/context.xml` doit être placé dans `webapp`

Principe général

Un POJO annoté au minimum

Université de Lille

```
@Entity
public class Client implements Serializable {
    @Id
    private Integer id;
    private String nom;
    private String prenom;

    // accesseurs
    ....
}
```

- Ces 2 annotations sont les seules obligatoires
- Il existe de nombreuses autres annotations (`@Table`, `@Column`, ..)

Principe général

Des requêtes par défaut

Université de Lille

`EntityManager` contient 4 méthodes génériques pour manipuler les objets

- `persist` – permet de sauver (faire persister) l'objet
```
em.persist(client);
```
- `find` – permet de rechercher un POJO sur sa clé
```
Client  client = em.find(Client.class, 17);
```
- `contains` permet de savoir si l'em manage cet objet (S'il l'a en memoire)
```
boolean b = em.contains(client);
```
- `remove` permet de détruire un objet (il doit avoir été récupéré)
```
em.remove(client);
```

Principe général

Le CRUD

Université de Lille

Toutes les opérations de m.a.j. se font entre un `em.getTransaction().begin()` et un `em.getTransaction().commit()`

- Création : `em.persist(o)`
- Recherche : `em.find(Classe,clé)`
- MàJ : si l'objet est persistant, il suffit de le modifier
- Effacement : `em.remove(o)`

Principe général

L'utilisation de l'ensemble

Université de Lille

```
public class MonProg
{
    public static void main(String[] args)
    {
        // Création d'un EntityManager
        EntityManagerFactory emf =
              Persistence.createEntityManagerFactory("testjpa");
        EntityManager em = emf.createEntityManager();

        // Création d'un nouveau client
        Client client = new Client();
        client.setId(1);
        client.setNom("Mathieu");
        client.setPrenom("Philippe");
        em.getTransaction().begin();
        em.persist(client);
        em.getTransaction().commit();
        emf.close();
    }
```

Principe général

Lien avec le DDL de la base de données

Université de Lille

Parmi les propriétés du `persistence.xml` la propriété `jakarta.persistence.schema-generation.database.action` permet de spécifier l'action à réaliser sur le schéma de base à chaque lancement du pgm.

- `none` (default)
- `create` (création si le schema n'existe pas)
- `drop-and-create` (détruit l'ancien schéma et recrée)
- `drop`

```
<property name="jakarta.persistence.schema-generation.database.action" value="drop-and-create"/>
```

En phase de mise au point `drop-and-create` est particulièrement utile !

Principe général

A l'usage

Université de Lille

- Dans les classes on ne parle que "Objet"... jamais de BDD !
- C'est JPA qui crée les tables
- C'est JPA qui initialise les données dedans
- C'est JPA qui s'occupe de SQL etc ...
- C'est JPA qui gère les différents caches et l'optimisation

Plan

Université de Lille

Paramétrages du `persistence.xml`

Génération de scripts SQL

Université de Lille

Sauvegarde des scripts DDL SQL utilisés par JPA :

- `jakarta.persistence.schema-generation.scripts.action`
  types de scripts à générer lors de la création : none (default), create, drop-and-create and drop.
- `jakarta.persistence.schema-generation.scripts.create-target`
  le nom du fichier généré pour les ordres de création
- `jakarta.persistence.schema-generation.scripts.drop-target`
  le nom du fichier généré pour les ordres de suppression

Propriétés très utiles pour les vérifications !

Paramétrages du `persistence.xml`

Chargement de données

Université de Lille

- `jakarta.persistence.sql-load-script-source`
  fichier de données à importer lors du lancement (que des Insert ! !)

  Avec `drop-and-create` et un fichier de données, il est alors possible de paramétrer son projet pour qu'à chaque lancement
  1. Il détruise les anciennes tables
  2. Il reconstruise automatique toutes les nouvelles tables
  3. Il remplisse ces nouvelles tables avec des données

- Augmenter le niveau de trace permet de voir les ordres SQL passés par JPA :

```
<property name="eclipselink.logging.level" value="FINE"/>
```

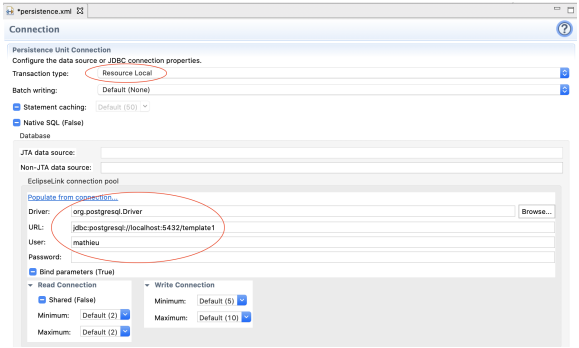
Une autre manière de faire.

Au lieu de demander à JPA de créer les tables en fonction des entités, on peut lui demander de lancer des scripts SQL-DDL de création et de destruction de tables.

- on paramètre la propriété `database.action` à `none`
- `jakarta.persistence.schema-generation.create-script-source`
  le nom du fichier SQL à utiliser pour la création de la base
- `jakarta.persistence.schema-generation.drop-script-source`
  le nom du fichier SQL à utiliser pour la suppression de la base

Paramétrages du `persistence.xml`

sous Eclipse



Plan

Université de Lille

Les annotations

De très nombreuses annotations

Université de Lille

voir la javadoc de javax.persistence ou www.objectdb.com

```
@Entity
@Id

@Table(name="xxx")
@ColumnName(name="xxx")
@GeneratedValue(strategy=GenerationType.xxx)

@OneToMany(MappedBy="xxx")
@ManyToOne
@JoinColumn(name="xxx", referencedColumnName="yyy")

@Embeddable // pour une classe de définition d'une clé multi-attributs

@NamedQuery
```

Les annotations

Un POJO un peu plus annoté ...

Université de Lille

```
@Entity
@Table(name = "LeClient")
@NamedQuery(name="Client.findAll", query="SELECT c FROM Client c")
public class Client implements Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;
  @Column(name = "lenom", nullable=false, length=20)
  private String nom;
  private String prenom;

  // accesseurs
  ....
}
```

Plan

Université de Lille

Les requêtes complémentaires

Définir ses propres requêtes

Université de Lille

Trois méthodes de définition des requêtes complémentaires :

- `createNamedQuery(String)`
  - S'appuie sur une requête JPQL définie dans le POJO
  - à utiliser pour des requêtes standard et réutilisables (findByNom, findAll,...)
- `createQuery(String)`
  - Crée une requête online à partir d'une description JPQL
  - à utiliser pour des requêtes créées dynamiquement (dans des boucles par ex)
- `createNativeQuery(String)`
  - Crée une requête à partir d'une description SQL dépendant du SGBD sous-jacent
  - à utiliser pour des requêtes complexe, non supportées par JPQL

Dans les 3 cas, les méthodes `getResultList()` , `getSingleResult()`, `getMaxResults()`, `executeUpdate()` exécutent la requête

Les requêtes complémentaires

Exemple `createNamedQuery(String)`

Université de Lille

Requêtes réutilisables définies dans le POJO en JPQL

```
@Entity
@NamedQuery(name="Client.findAll", query="SELECT c FROM Client c")
public class Client implements Serializable
{...}
```

et dans le code ....

```
List<Client> result =
     em.createNamedQuery("Client.findAll").getResultList();

for (Client c:result)
    System.out.println(c);
```

Les requêtes complémentaires

Exemple `createQuery(String)`

Université de Lille

Requêtes créées dynamiquement dans le programme en JPQL

```
List<Client> result = em.createQuery(
    "Select c from Client c where c.age<30").getResultList();

for (Client c:result)
    System.out.println(c);
```

Les requêtes complémentaires

Exemple `createNativeQuery(String)`

Université de Lille

Requêtes créées dynamiquement dans le programme en langage natif

```
List<Client> result = em.createNativeQuery(
     "Select * from client",Client.class).getResultList();

for (Client c : result)
    System.out.println(c);
```

Les requêtes complémentaires

Passage de paramètres en JPQL

Université de Lille

```
Query query  = em.createQuery(
      "Select c from Client c where c.age < :age");
query.setParameter("age",18);

List<Client> result = query.getResultList();
for (Client c:result)
    System.out.println(c);
```

Les requêtes complémentaires

Exemple de mise à jour JPQL

Université de Lille

```
Query query = em.createQuery(
      "DELETE FROM Client c WHERE c.age < :age");
  int nb = query.setParameter(age, 18).executeUpdate();
```

Plan

Université de Lille

Les associations

Le principe

Université de Lille

- Le programme gère des objets qu'il faut faire persister dans le SGBD
- D'une manière générale tout ce qui annoté par `@Entity` donnera naissance à une table
- Néanmoins ces objets sont rarement indépendants : principe d'encapsulation ou d'héritage
  (*Equipe qui a un chef, Personne qui contient une liste d'adresses ; Auteur qui contient une liste de livres, ...*)

  Les annotations suivantes indiquent comment gérer les clés étrangères :
  `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`

Les associations

Lien hierarchique

Université de Lille

Prenons le cas d'Auteurs et leurs Livres
Concernant la classe `Livre`, chaque livre n'a qu'un Auteur

```
@ManyToOne
private Auteur auteur;
```

Récupérer une instance de Livre fournira automatiquement l'objet Auteur (mode `Eager` par défaut)

Par défaut JPA ajoute une colonne à la table pour la clé étrangère (nom de la propriété annotée + _ + nom de la clé primaire).
L'annotation `@JoinColumn` permet de changer le nom de la clé étrangère.

Les associations

Lien réciproque

Université de Lille

Concernant la classe `Auteur`, chaque auteur contient une collection de livres

```
@OneToMany
private List<Livre> livres;
```

Récupérer un auteur ne fournit pas directement la collection (mode `LAZY`)

- Soit utiliser l'accesseur
- soit le demander explicitement avec un `join fetch` dans la req
  `select a from Auteur a join fetch a.livres`
- soit déclarer la collection en mode Eager :
  `@OneToMany(mappedBy="auteur" , fetch=FetchType.EAGER)`

le paramètre le `mappedBy` traite le cas d'une relation bi-directionnelle.

Sans cela, il considèrera 2 relations mono-directionnelles et recréera une table de jointure.

Les associations

Ce qui donne éventuellement

Université de Lille

```java
@Entity
@NamedQuery(name="Auteur.findAll",
            query="SELECT a FROM Auteur a")
public class Auteur implements Serializable {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Integer idauteur;
   private String email;
   private String nom;
   private String prenom;

   @OneToMany(mappedBy="auteur",fetch=FetchType.LAZY)
   List<Livre> livres;
   ...
```

```java
@Entity
@NamedQuery(name="Livre.findAll",
            query="SELECT l FROM Livre l")
public class Livre implements Serializable {
   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private Integer idlivre;
   private String categorie;
   private String titre;

   @ManyToOne(cascade = CascadeType.ALL)
   @JoinColumn(name="idAuteur")
   private Auteur auteur;
    ...
```

EAGER charge la propriété immédiatement, LAZY charge à l'appel du "getter" correspondant

Par défaut : OneToMany : LAZY , ManyToOne : EAGER , ManyToMany : LAZY , OneToOne : EAGER

Les associations

En résumé

Université de Lille

- `1:1` mono-directionnel `@OneToOne` sur l'attribut lien
- `1:1` bi-directionnel `@OneToOne(mappedBy="attr-liée")` sur l'autre attribut lien
- `1:n` mono-directionnel `@OneToMany` sur la collection côté n
- `1:n` bi-directionnel `@OneToMany(mappedBy="attr-lié"` sur la collection côté 1 et `ManyToOne` sur l'attribut clé étrangère
- `n:m` mono-directionnel `@ManyToMany` d'un côté
- `n:m` bi-directionnel `@ManyToMany(mappedBy="")` d'un côté et `@ManyToMany` de l'autre

Les associations

A la création d'un projet

Université de Lille

Deux grandes philosophies co-existent :

1. Issue du monde "Objet"
   - Je fais ma structuration "objet" (UML)
   - Peu m'importe comment sont faites les tables
2. Issue du monde "BDD"
   - Je pars d'un schéma de BDD (EA)
   - Je fais en sorte que mes objets soient conformes aux tables

C'est en général plus facile de créer des tables que de faire les POJO !

Les associations

Une bonne démarche ...

Université de Lille

1. Etablir un MCD et un MLD adapté
2. Créer les POJO pour qu'ils correspondent exactement au MLD
3. Se mettre en mode `drop-and-create` lors des developpements

De cette manière, à chaque lancement du projet, les tables seront recréées exactement comme prévu.

Les associations

Génération automatique des Entity Beans

Université de Lille

Plusieurs outils permettent de créer automatiquement les entités à partir de la base

- Eclipse : sur `JPA Tools`, “`Generate entities from tables`”
- IntelliJ : “`Generated Persistence Mapping`” puis “`By Database Schema`”
- Maven possède aussi un plugin `maven-jpa-entity-generator-plugin`

Les associations

Un Plugin de génération pour Maven

Université de Lille

Dans `pom.xml`

```
<plugin>
  <groupId>com.smartnews</groupId>
  <artifactId>maven-jpa-entity-generator-plugin</artifactId>
  ....
```

Dans `src/main/resources/entityGenConfig.yml`

```
jdbcSettings:
  url: "jdbc:postgresql://localhost/template1"
  username: "user"
  password: "pass"
  driverClassName: "org.postgresql.Driver"

packageName: "fr.but3.recup"
```

but : `mvn jpa-entity-generator:generateAll`

Voir explications sur le dépot

Les associations

Débuter avec Eclipse

Université de Lille

- S'assurer d'avoir ses drivers BDD, JPA et ORM
- `File` , `New` , `JPA project`
  (ouvre automatiquement la perspective JPA)
- `Database Connection` , `New` , puis configurer la base
- Click droit sur la base et tester `ping`
- Double click sur `persistence.xml`, onglet `Connection`, mettre `resource Local`, puis `populate from connection`
- Click droit sur le nom de projet, `Properties`, `Java Build Path`, Ajouter les 3 drivers au classpath
- Click droit, `JPA Tools` , `Generate tables from entities`

A garder sous le coude !

Permet de faire des tests, de créer ses Entity et d'avoir un exemple de persisence.xml