Java Persistence API : persistance universelle

Java Persistent API est un framework de persistence reprenant les grandes lignes d'Hibernate et les généralisant. Celui-ci est intégré à Java Entreprise Edition 5. Une implémentation est donc disponible avec tous les serveurs du marché conforme à cette norme. Cet article vous apprendra les bases de JPA et permettra de mieux cerner le gain de cette API dans vos projets.

Sur l'auteur

NOËL PEREZ

Noel Perez est expert en Java et Système Multi Agent, après cinq ans chez OSLO où il dirigeait la R&D, soit une équipe d'une dizaine de personnes Il se recentre sur l'architecture et possède une forte expertise en Java, Hibernate, Oracle qu'il met à profit sur différents projets client portail, simulation d'usine, outil de modélisation

Niveau de difficulté

Tout d'abord, je vais présenter le contexte dans lequel se place ce texte. Aucun projet informatique moderne ne se conçoit de nos jours sans sauvegarde de données. Le plus souvent concrétisée par une base de données et la création d'objets associés capables de se charger ou de se décharger à partir de cette même base. Cette solution pose de nombreux problèmes .

- Besoin d'expert pour créer schéma et requêtes de la base,
- Difficulté à maintenir ce même schéma et ces requêtes,
- Dépendance vis à vis du choix initial qui peut nuire à la montée en charge d'une application.

Une première réponse fut apportée avec les couches DAO permettant de dissocier les personnes travaillant sur la base, des développeurs métiers. Mais le changement de SGBD nécessitait tout de même la réécriture de cette même couche. Afin d'améliorer cette portabilité, les frameworks de sauvegarde sont apparus · TopLink, JDO, Hibernate et autres permettant tous par configuration de changer de SGBD cible. Mais ceux-ci sont tous très particuliers et incompatibles d'où l'utilité d'une standardisation sous une même banière . JPA.

JPA n'est pas une implémentation, mais juste un modèle de persistance défini au travers d'une API que les frameworks précédemment cités vont implémenter Interface standardisée dans JEE1.5 et donc livrée avec tous les serveurs JEE respectant cette norme Jboss, Geronimo, Weblogic, Oracle, etc. Il fournit ainsi une indépendance vis à vis des SGBD mais aussi des serveurs d'exécution.

Maintenant que son utilité est avérée, voyons un peu comment cela se présente.

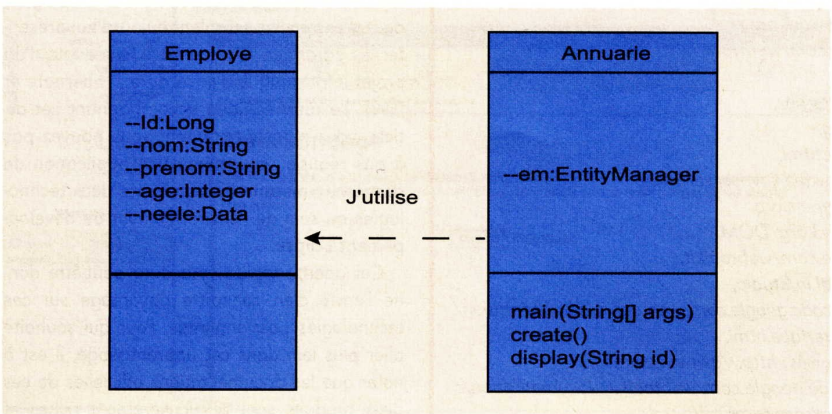

Figure 1. Diagramme de classe de l'application

Principe

Java Persistent API, comme précédemment dit, est un framework de persistence qui reprend les grandes lignes de ce que l'on a put connaître avec Hibernate et les généralises. La Figure 2 illustre ceci au travers des différentes couches qui constituent une application :

- En violet, les couches sous la responsabilité de l'utilisateur,
- En bleu, la couche JDBC permettant l'accés à la base de données dont historiquement l'interaction devait être gérée par le développeur et nécessitait des compétences importantes en SGBD pour créer tables et requêtes,
- Enfin en vert, les couches ajoutées qui permettent au développeur de ne plus se préoccuper de la base de données.

Comme vous pouvez le voir sur le schéma, le developpeur n'a plus besoin de connaissance particulière en SGBD mais juste autour de JPA. Tout le travail concernant la base de données se fera au travers de fichiers générés (schéma, requête, mapping), le développeur se contentant de paramétrer l'emplacement de sa base, son type et le couple login/pass pour être en mesure de l'exploiter.

Technique

Afin de mieux comprendre le fonctionnement de JPA, je vous propose de réaliser un simple annuaire dont les entrées seront sauvées dans une base Mysql au travers de JPA-Hibernate Ici vous évacuerez tout problème d'interface graphique, ainsi vous aurez un schéma de classe assez simple représenté sur la Figure 1

Vous trouvez des employés identifiables par un numéro unique · ID La liste de ces employés est modifiable et consultable au travers de l'annuaire. Vous comprenez donc que la classe Employe sera la classe persistante et que la classe Annuaire est là uniquement pour gérer le cycle de persistance et pour permettre l'interrogation de la liste

Nous nous placerons dans le cadre où la base de données est vierge et sera donc générée par JPA. Il est important de comprendre que vous pouvez aussi travailler à partir d'une base de données pré-existante et je ferai quelques allusions au fil du texte.

Installation

Avant d'entrer dans le vif du sujet, petit retour sur les prérequis d'un projet utilisant JPA et ce dont vous aurez besoin pour ce premier test

Tout d'abord, un JDK 5 au minimum afin d'avoir la gestion des annotations sur lesquelles vous allez vous appuyer. Faites attention à la version de votre IDE qui doit accepter ces mêmes annotations. Les dernières versions d'Eclipse associées avec Eclipse Web Tools Platform Project vous permettront d'être le plus productif, si vous travaillez sur portail Web. La version 3 3 intègre la gestion de JPA et sera donc un bon choix. Vous aurez besoin enfin de deux choses .

- Tout d'abord, l'API de JPA que vous pouvez télécharger à l'URL suivante *http://java.sun.com/javaee/downloads/index.jsp* (si vous travaillez à partir de JBoss, l'API se trouve dans le fichier *ejb3-persistence.jar* pour la version 4 2 2GA) et qu'il vous faudra intégrer au classpath du projet,
- Et d'une implementation de JPA Si vous travaillez sur une application Web, tout serveur JEE conforme 1.5 sera suffisant Sun vous fournit une implementation avec son SDK, mais je conseillerais quand même l'utilisation de JBoss Si vous voulez travailler sur une application stand-alone, vous devrez télécharger l'une des implémentations officielles (voir rubrique Sur Internet), dans ce cas Hibernate

Il est à noter que toutes les classes utiles de JPA appartiennent au package *javax.persistence*

Objet persistant

Dans ce cas, la classe *Employe* représente le seul objet à persister. Comme vous pouvez le voir au travers du Listing 1, les classes d'objets persistants sont de simples POJO auxquels des annotations contrôlant la persistance sont ajoutées.

Si on prend le temps d'analyser plus précisement le Listing, on s'aperçoit que les objets persistants reposent sur quelques règles :

- (1) tout objet doit être annoté `@entity` pour indiquer sa capacité à se sauver,
- (2) la classe doit être publique,

Remarque I

JPA est dans la norme JEE 1 5+, mais il ne concerne pas que les applications Web. Il peut tout à fait être intégré à une application JSE moyennant l'inclusion du jar JPA dans le classpath du projet et l'utilisation d'une des bibliothèques implémentant JPA (j'y reviendrai dans la rubrique installation)

Remarque II

Il est possible d'ajouter des annotations afin d'orienter le nom de la table créée sur la base pour stocker les employés `@Table(name="employe")` Ceci est le plus souvent utilisé quand on part d'une base existante, mais cela peut être rassurant si on sait vouloir changer d'implémentation de JPA ou de base Et oui, en fonction de l'implementation de JPA ou de la base utilisée les schèmas générés peuvent différer En définissant les tables, on augmente les chances de pouvoir changer de cible sans perdre les données sauvées par les précédents lancement de l'application.
Autre remarque importante, JPA est en accord avec le paradigme objet, puisqu'il va aussi gérer la persistance de données héritées

Terminologie

- *IDE* Environnement de développement intégré tel que Eclipse ou Netbeans,
- *JSE* Java standrard édition,
- *JEE* Java entreprise édition,
- *Pojo* Acronyme de Plain Old Java Object,
- *Annotation* Les Annotations permettent de marquer différents éléments du langage Java avec des attributs particuliers, afin d'ajouter des traitements avant ou en sus de la compilation grâce au nouveau JDK > 5 0

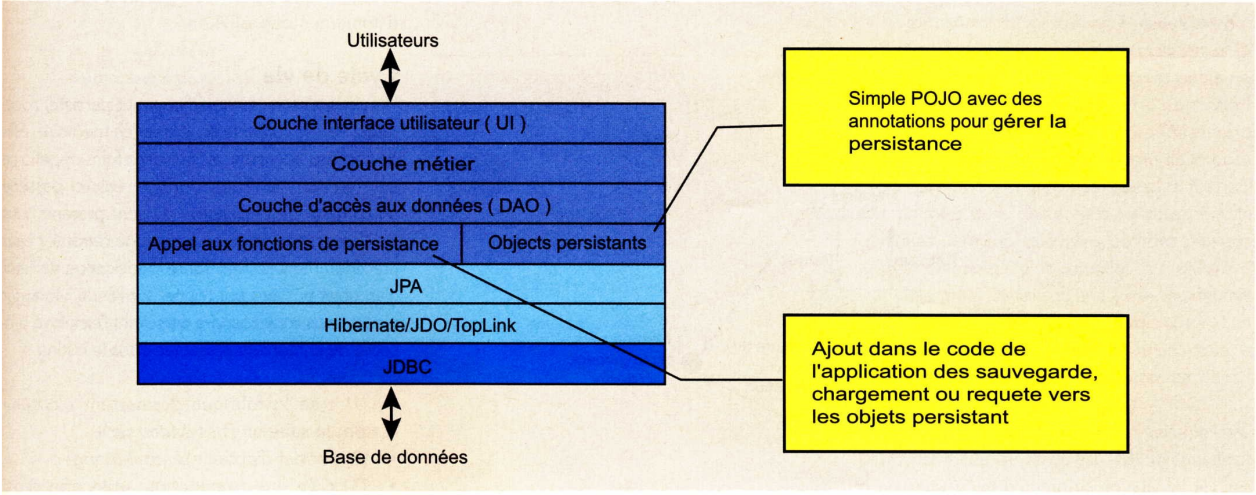

Figure 1. Couches applicatives

- (3) il faut déclarer un identifiant unique permettant de retrouver un objet persistant Cet identifiant peut-être : soit une propriété de la classe déclarée par l'annotation `@id`, soit un ensemble de propriétés et nécessite alors la création d'une classe particulière Cette classe devra être `serializable` et toutes ses propriétés publiques. Elle devra aussi implémenter les fonctions `equals(Objet)` et `hashCode()`,
- (4) dans ce cas, on va vouloir générer l'identifiant lors de la première sauvegarde. JPA met à notre disposition des stratégies de génération de clé avec cette annotation,
- (5) il est nécessaire d'implementer la fonction `hashcode`,
- (6) la fonction `equals` est elle aussi nécessaire.

Listing 1. Classe persistante Employe

```
import javax.persistence.*;
@entity (1)
public class Employe(2) {
   @Id (3)
   @GeneratedValue(strategie=GenerationType.AUTO) (4)
   private Long id;
   private String nom;
   private String prenom;
   private Integer age;
   private Date neele;
   public void setId(Long _id){id = _id;}
   public Long getId(){return id;}
   public void setNom(String _nom){nom = _nom;}
   public String getNom(){return nom;}
   ...
   public int hashcode(){return id;}(5)
   public boolean equals(Object o) (6) {
      if(o instanceof Employe && ((Employe)o.getId().equals(this.getId())) {
         return true;
      } return false;
} }
```

Listing 2. Fichier de configuration JPA : persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" 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_1_0.xsd">
   <persistence-unit name="jpa" transaction-type="RESOURCE_LOCAL">
      <provider>org.hibernate.ejb.HibernatePersistence</provider> (1)
      <jta-data-source>jpa</jta-data-source> (2)
      <class>Employe</class> (3)
      <properties> (4)
         <property name="hibernate.dialect"
               value="org.hibernate.dialect.MySQLDialect"/> (5)
         <property name="hibernate.show_sql" value="true"/> (6)
         <property name="hibernate.hbm2ddl.auto" value="update"/> (6)
      </properties>
   </persistence-unit>
</persistence>
```

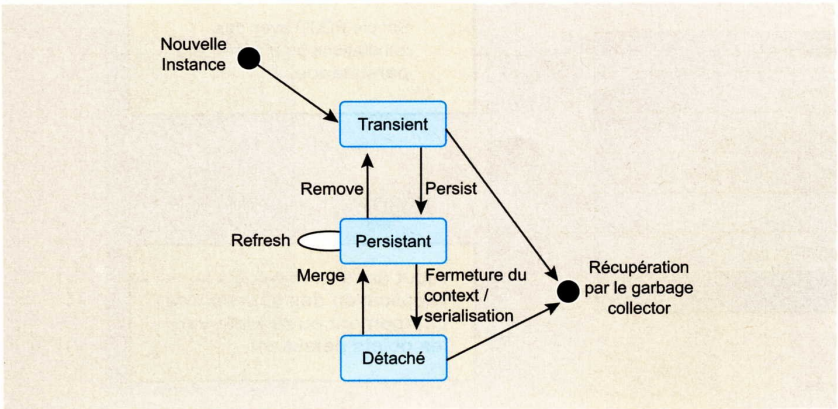

Figure 3. Cycle de vie d'un objet persistant

Base de données

Même si JPA est là pour rendre transparente l'utilisation d'une base de données, il vous faudra néanmoins configurer JPA afin de lui indiquer quelle base de données utiliser et quelle implémentation de JPA vous souhaitez utiliser Tout ceci se configure à travers le fichier *persistence.xml* qui se présente sous la forme suivante présentée dans le Listing 2. Voici l'explication des éléments du Listing 2 :

- (1) vous allez commencer par fournir l'implémentation de JPA à utiliser, ici Hibernate,
- (2) vous indiquez le datasource (nom de la définition de la base de données faite dans le serveur JEE). Vous pouvez si nécessaire indiquer l'ensemble des paramètres de connexion à la base (*login/pass/serveur/port* .),
- (3) chaque classe persistante doit être indiquée ici afin d'être prise en compte,
- (4) la zone properties correspond à une zone où sont indiquées les propriétés propres à l'implémentation de JPA choisie,
- (5) indique à Hibernate le type de la base de données cible,
- (6) indique le mode de création de la base. Dans ce cas, on indique à Hibernate de générer la base de données si besoin (table, type des champs, etc.).

Remarquez que ce fichier sera le seul à modifier que ce soit pour changer de base cible ou d'implémentation JPA.

Cycle de vie

Un objet n'est pas immédiatement persisté, l'opérateur `new` se contente de le créer en mémoire. Afin de gérer sa persistance, vous allez faire appel à ce que l'on nomme l'*EntityManager*, celui-ci possède un contexte de persistance où sont présents tous les objets à sauver et qui permet de retrouver ceux précédemment sauvés. Toute modification de l'état d'un objet passera par l'appel sur l'EntityManager ainsi si vous voulez rendre persistant l'employé procédez de la manière présentée dans le Listing 3 :

- (1) crée la fabrique permettant d'obtenir par la suite un *EntityManager*,
- (2) permet d'obtenir l'*EntityManager*,
- (3) crée une transaction, utile afin d'assurer l'atomicité de plusieurs opérations

comme l'inscription d'objets dépendants. Si l'un des objets ne peut être inséré, toutes les opérations précédentes sont annulées,
- (4) déclare le nouvel employé comme persistant (Figure 3),
- (5) inscrit les modifications sur la base de données,
- (6) ferme l'*EntityManager* et vous sauvez son contexte si modification depuis dernier commit

L'EntityManager permet de gérer l'ensemble du cycle de vie des objets persistants au travers d'une suite de fonctions (Figure 3)

Comme vous pouvez le voir un objet est transient à sa naissance, c'est-à-dire qu'il n'existe que dans la mémoire de la JVM Suite à l'appel de la fonction 'persist' il va devenir persistant et sera sauvé en base de données. Par la suite, il pourra être détaché et synchronisé avec la base de données en fonction des besoins

Java Persistence Query Language

Afin d'utiliser les objets déjà en base, il nous faut un moyen de les retrouver. C'est dans ce but que JPQL a été créé pour JPA. Il permet de rechercher des objets dans la base, de les mettre à jour, etc. Il s'agit d'une abstraction par dessus SQL qui manipule des objets au lieu de tables et colonnes, ainsi on transformera la requête suivante :

```
Select * FROM EMPLOYE where NAME='PERE'
```

en

```
select p from Employe p where p.nom='PERE'
```

Vous allez retrouver en JPQL les instructions, `select`, `update` et `delete` ainsi que les `order by`, `group by` et `having`.

CallBack d'évenements

Un autre point important de JPA réside dans la gestion des traitements sur les changements d'états des objets, autrement appelé les *callback*.

Les callbacks sont en fait des annotations à ajouter sur une fonction de l'objet persistant et qui déterminera les moments ou la fonction sera lancée.

On retrouve ainsi les *callbacks* tout au long du cycle de vie des objets persistants :

- lorsque l'objet est rendu persistant suite à l'appel de la fonction `persist` : `@PrePersist`, `@PostPersist`,
- au moment de la suppression : `@PreRemove`, `@PostRemove`,
- lors du chargement d'un objet : `@PostLoad`,
- ou encore pendant sa mise à jour : `@PreUpdate`, `@PostUpdate`.

Les callbacks dont le nom commence par *pre* sont appelés avant l'action demandée sur l'objet. Ainsi une méthode notée `@PrePersist` sera appelée automatiquement juste avant de rendre un objet persistant. De la même façon le callback commençant par 'post' sera, lui appelé une fois l'objet persisté en base. Nous allons ainsi pouvoir avec la fonction annotée `@PrePersist` vérifier la cohérence des données et avec une autre fonction annotée `@PostPersist` confirmer que l'objet a bien été sauvé.

Cette fonctionnalité est donc très utile pour ajouter des tests assurant la cohérence des données, lancer des alertes, etc.

Conclusion

Il est important de noter que les frameworks de persistance sont souvent difficiles à adapter pour traiter de très gros volumes de données et nécessitent de bien évaluer le paramétrage et le besoin avant de lancer un projet sous peine de voir les performances de son application fortement dégradées par la couche de persistance. Il est important de bien paramétrer le fichier *persistence.xml* et tout particulièrement la balise *properties* qui est liée à l'implémentation choisie.

Dans le cas présent, vous pourrez avec l'utilisation d'un pool de connection ou d'un cache grandement améliorer les performances mais ceci relève du paramétrage d'Hibernate.

Je terminerai en précisant qu'il est toujours plus simple de créer les objets persistants et générer la base que l'inverse. Certains outils permettent la génération des POJO à partir de la base mais ceux-ci s'avèrent peu pratiques à l'utilisation, car trop proche du schéma de la base.

JPA, Hibernate et autres sont de plus des outils très adaptés dans des projets où vous persistez quelques données, que vous accédez plus tard (portail Web) mais plus compliqués à utiliser, s'il s'agit de gérer le fait qu'une base de données contiennent tout un pan de la structure d'une application que vous remontez d'un coup en mémoire pour travailler dessus comme dans le cadre d'une simulation.

J'espère maintenant que vous avez une vision assez complète de ce qu'est JPA. Je vous invite à poursuivre votre initiation en suivant les liens que je joints à l'article

Sur Internet

- Lien officiel sur JPA *http://java.sun.com/javaee/technologies/persistence.jsp*
- Exemple d'utilisation de JPA dans Glassfish *https://glassfish.dev.java.net/javaee5/persistence/persistence-example.html*,
- Une des implémentations de JPA parmis les plus connues et libres *http://www.hibernate.org*,
- Implémentation professionnelle de JPA gérée par Oracle *http://www.oracle.com/technology/products/ias/toplink/jpa/index.html*,
- Implémentation qui monte et qui est utilisée sur Géronimo le serveur JEE d'apache - *http://openjpa.apache.org*

Listing 3. Classe annuaire

```
import javax.persistence.*;
public class Annuaire {
   private EntityManager em;
   public Annuaire() {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa"); (1)
            em = emf.createEntityManager(); (2)
            emf.close();
   } public void create() {
      EntityTransaction tx = em.getTransaction(); (3)
      tx.begin();
      em.persist(new Employe()); (4)
      em.persist(new Employe());
      tx.commit(); (5)
   } public void display(String nom) {          // décris dans la section sur JPQL
   } public void dispose() {
      em.close(); (6)
   } public static void main(String[] args) {
      Annuaire annu = new Annuaire();
      if("load".equals(args[0])) {
         create();
      } else if("search".equals(args[0])) {
         display(args[1]);
      } annu.dispose(); }}
```