JAVA Persistence API (JPA)

**Eric Sultan** [e.sultan@ajc-ingenierie.fr](mailto:e.sultan@ajc-ingenierie.fr) 0645104506

# Introduction & Définitions

* **API** (Application Programming Interface) / **ORM** (Object-Relational Mapping) destinée à la persistance d’objet (contrairement à transient)
* Stockage d’informations en Bases
* JPA génère les requêtes SQL pour nous
* JPA est une sous-spécification **d’EJB3 (Entreprise Java Beans)**
* Inclus dans JEE (Java Entreprise Edition)
* Hibernate & ClipsLinq ? sont des implémentations JPA
* JPA est une sorte de définition via interfaces, pas de code.
* Avec ça on code des POJO (Plain Old Java Object)
* Avantage de ces classes pour les tests = on limite les facteurs modifiants = on isole plus rapidement le problème = implémentations de couches
* **EJB1 à fuir, très mal foutu… EJB2 c’est chaud, EJB3 c’est calme.**
* L’idée principale de JPA c’est de **cacher le SQL**. On passe par le **JPQL (Java Persistence Query Language)**. Cela retourne directement les **objets**.
* **Classe maître = EntityManager**
* De cette classe on fait une transaction avec L’**EntityTransaction,** puis ça devient des **EntityBeans**

# Entity Beans

* API EntityManager sert à manipuler et créer des instances d’EntityBeans7
* **Application-managed** (nous-même) ou **Container-managed** (appli)
* EntityBean = objet java ordinaire avec des annotations spécifiques à JPA/Hibernate (**@Entity** avant la classe par exemple = je veux l’img de cette classe en BDD)
* Puis méthode de persistance pour enregistrer l’objet en base (**persistance**)
* Règles d’un JavaBeans = **constructeurs, getters & setters**
* Par défaut, tous les attributs d’une entité sont persistants, sinon on ajoute **@Transcient**

## Application-managed

* Souvent utilisée pour les app autonomes Java SE
* Code servant à obtenir et fermer explicitement l’EntityManager
* On le ferme impérativement dans le **finally**  du trycatch
* Peut être impliquée dans une **JTA (Java Transaction API)**: transaction distribuée= 2 Phases Commit : une grande transaction récupère les petites transactions et envoie d’abord un faux commit pour valider toutes les petites transactions, puis si ok envoie le vrai commit… Liaison entre plusieurs sources externes type BDD…
* Config de l’EntityManager dans un XML

## Container-managed

* Généralement utilisé dans un JEE 🡪 serveur d’application
* Récup l’EntityManager par injection de dépendances, ou le créer à partir d’une fabrique **EntityManagerFactory**

## Contexte de persistance

* Par défaut, le best : EMF 🡪 EM 🡪 EntityTransaction 🡪 persist(obj) 🡪 ET commit 🡪 EM close() ; (fin du cache)
* On peut étendre le cache = persistance étendue **à éviter absolument car erreur de conception… les query seront gardées en mem jusqu’à la prochaine connexion et les effectuera à ce moment = bordelik**
* On préfèrera le cache de second niveau = monter en mem toutes les infos qu’on va utiliser très souvent.
* On peut spécifier un cache par transaction

## EJB (Enterprise Java Beans)

* Permettent de résoudre certaines problématiques de discussion entre serveurs
* Protocol **RMI Remote Method Invocation**
* Pas très aimé des net&sysadmin car jouer avec les ports et tout ça c’est pas fun
* Problème du coup de contraintes, il faut impérativement auditer les contraintes techniques en amont de la réalisation, anticiper les problèmes
* **Protocol préféré http / https** le plus simple, le plus ouvert…

## Entités détachées

* Une entité gérée devient détachée lorsqu’un contexte de persistance est détruit.
* On peut toujours faire des modifs sur une entité détachée, elles ne seront pas enregistrées en base… Il faut rétablir une connexion
* Ou on peut la sérialiser
* Quand on est connecté on ne peut pas sérialiser, on est attaché au serveur

## Unité de persistance

* EM lie un ensemble de classes d’**EntityBeans** à une BDD, cet ensemble = Unité de Persistance
* Les UP sont décrites dans un xml placé dans un rep META-INF
* Une UP définie et/ou configure :
  + L’ensemble des classes d’EB concernées
  + Le fournisseur de persistance (Hibernate, autres…)
  + La source de données
  + Le mode de gestion transactionnelle (JTA ou RESOURCE\_LOCAL)

## Interagir avec un Entity Manager

* Un EB peut prendre 4 états par rapport à un EM
  + New : objet associé à aucun contexte et pas d’image BDD
  + Managed : objet inclus dans le contexte de persistance, possède une image BDD, modif gérées par EM
  + Detached : obket possède image BDD, mais échappe au contrôle de l’EM, contexte fermé ou vidé
  + Removed : objet encore inclus dans un contexte mais destiné à être retiré de la BDD
* CF liste des fonctions dans le PDF JPA
* **Flush()** vide le cache, en exécutant son contenu d’abord. Auto dans le commit.
* Pour relier deux entités entre elles il faut que les deux soient persistantes
* Lorsqu’une modif est faite sur une entité, il y’a **dirty checking**, flag des attributs modifiés et de l’entité elle-même. Elle sont analysées par le flush/commit
* Une entité liée à un EM, ne pourra jamais être reliée à un autre EM. Une entité ne sera plus utilisable après fermeture de l’EM
* Lors d’un merge, il retourne une **copie de l’objet qui est managé par un autre EM** dans l’EM a merger
* Lorsqu’on récupère une entité, elle est stockée dans le cache. Du coup si on demande le même objet, il va regarder d’abord dans le cache, puis pointer dessus (même adresse). **Valable aussi lors d’un merge**

## Ecriture des Entity Beans

* **JavaBeans marqués avec @Entity**
* **Attributs + Getters&Setters + constructeurs**
* Sa classe, ses propriétés et ses méthodes ne peuvent pas être **final**
* Doit avoir au moins un ID (clé primaire)
* Les champs multi-valués doivent être déclarés comme des interfaces (List, Map…)

# Créer un bon environnement de développement

## Variables d’environnement

* On préfèrera installer les différentes version du JDK dans un même répertoire de dev
* On créer une variable d’environnement Windows, ouvrir le panneau système puis :
  + Paramètres système avancés
  + Variables d’environnement
  + Nouvelle… : JAVA\_HOME + répertoire du jdk à utiliser 🡪 OK
  + Modifier **Path** en ajoutant au tout début : %JAVA\_HOME%\bin;

## Compilation, class path et dépendances

* Le compilateur javac charge direct les class du jdk et celles présentes dans le dossier de travail
* Il faut ensuite préciser toutes les classes externes utilisées (Eclipse le fait)
* Il faut également associer les dépendances à certains packages, genre Hibernate
* Et télécharger ceux qui ne sont pas présents sur la machine…
* == 3 Milliards de trucs à charger 🡪 on utilise un gestionnaire de projet **Maven**

## Maven

### Définition & Install

* Pom.xml à la racine du projet (son repertoire)
* GroupID = big projet principal
* Sous-projets avec les versions
* Pour un projet on indique les packages utilisés/les dépendances à installer, avec la version
* **On recherche les repo sur mvnrepository.com**
* Regarde s’il faut des dépendances transitives
* **On ne livre jamais une version en dev… on fait une release à une version + snapshot pour pouvoir retourner sur cette version si besoin de débug**
* Maven est un tout petit noyau, lanceur d’extensions. Et on lui greffe plein d’extensions qu’on veut utiliser.
* Hiérarchie/structure des répertoires prédef, mais on peut la modifier.
* **Installation :**
  + DL sur le site le Binary zip archive
  + Dézipper dans notre répertoire de dev (avec les versions de JDK)
  + Créer une variable d’environnement M2\_HOME + rep maven
  + Ajouter %M2\_HOME%\bin ; au PATH

### Fonctionnement

* Système de **Goals & Phases**
* Une phase appelle un plugin d’un goal…
* Mvn compile 🡪 validate, DL toutes les dépendances et tout ce qui va bien

### Eclipse

* Import/Maven/Existing Maven Project/Browse/Notre rep (contenant le pom.xml)
* Créer un persistence.xml (config liaison bdd) qu’on place dans src/main/resources/META-INF
* Test d’un EMF EM ET begin() commit() close().. voir si ça fonctionne
* TryCatcher les erreurs + fanally pour fermer les connexions

# ORM des Entity Beans

## Génération d’Id

* @Id Généré auto par les bases, de différentes façons : séquence partagée, séquence par table...
* On peut spécifier comment la valeur est générée avec @(type=…)

## Identifiant Composite

* Identifiant unique selon 1 ou plusieurs colonnes, deux manières :
  + @IdClass nul à chier
  + @Embeddable

## Classes embarquées

* Enregistrer toute une classe dans une colonne, ex : Table client avec son adresse (n°,rue.. Objet adresse Java)
* @Embedded0§eùrhcyciti (sur l’objet adresse dans client) & @Embeddable (sur la classe adresse)
* Cela créer les colonnes de adresse dans client

## Objets délocalisés/externalisés

* Un ou plusieurs attribut(s) objet d’une classe entité, peut être stocké dans une autre table plutôt que dans celle principale la classe.
* @SecondaryTable(s) au dessus de la classe (après @Entity)

**CF BANQUE TEST**

# Bonnes pratiques

* Toujours fixer les longueurs pour les col de chaines de char…
* Les @settings.. se mettent sur les getters
* Il est préférable d’utiliser Integer vs int, car Integer = objet avec « null » comme val par défaut et non 0. Dans une table c’est mieux d’avoir un NA qui spécifie que la val n’a pas été attribuée, plutôt qu’un 0.

# Relations entre tables

## @OneToOne

* Jointure directe simple Classe1.idFK = Table2.id
* Il faut renseigner les entités dans chaque classe
* Pour le mapping on a :
  + Un côté maitre, qui possède la FK (colonne associée) :

@OneToOne

@JoinColumn(s)(name=  « nom colone »)

* + Un côté esclave qui n’a pas de colonne spé : @OneToOne(mappedBy = « attribut FK du maitre »)

## @OneToMany, ManyToOne

* Connexion entre un objet et plusieurs autres
* Java :
  + Un côté avec la collection d’objets
  + L’autre avec l’attribut monovalué
* Mapping :
  + Au-dessus de la collection : @OneToMany(mappedBy= attribut monovalué)

## @ManyToMany

* Tables reliant id de deux autres tables
* @ManyToMany

@JoinTable…

* @ManyToMany(mappedBy=…)
* Méthode à bannir… Trop de données chargées ç chaque modif…
* **Créer plutôt une classe intermédiaire (qui correspondrait à la Table intermédiaire créée par le M2M) qui possède 2 one2many, vers les classes à relier**

## Attribut Cascade

* **Super dangereux, ne pas utiliser…**
* Cela précise l’action à effectuer sur les refs lors du persist
* Il est préférable de le gérer soi-même

# Stratégies de représentation

* 1 table pour 1 classe + toutes ses sous classes
* 1 table par classe
* 1 table attachée aux classes concrètes

## Table unique

* Plus utilisée, meilleur choix
* Beaucoup de valeurs NULL
* Préciser la colonne discriminante dans la super-classe @DiscriminatorColumn(role= ‘’nom’’, discriminatorType=DiscriminatorType.STRING)
* Préciser dans chaque sous classe la colonne discriminante

# Transactions

* Propriétés **ACID – Atomicité Consistance Intégrité Durabilité**
* setAutoCommit(false) sur la connexion JDBC, équivaut à tx.begin() en JPA
* API **JTA** = transaction multi-distribuée, multi-base, JPA directement lié si bien fait ?
* Utilise un serveur d’application, car connexion à différentes bases, non locales.
* **Une transaction est le temps d’un clique… max du max 100ms**
* **Les transactions sont faites le plus court possible**
* **Pour que deux transactions surviennent au même moment, et modifient les mêmes données, c’est complètement Négligeable (méthode optimiste)…**
* Cependant il ne faut pas laisser l’éventualité survenir, il existe deux types de gestion de transactions **Optimiste ou Pessimiste**

## Approche Optimiste

* Le contrôle se contente d’analyser les erreurs au moment de l’écriture des données.
* On informe lors de l’écriture si les données ont été modifiées.
* Meilleurs perf, car on ne lock aucune données
* Meilleur extensibilité (**scalabilité**), charge déterminé par les testeurs
* Exemple avec une gestion de **version,** en ajoutant @Version sur un attribut que l’on creer

## Approche Pessimiste