Développement d'applications Web transactionnelles

420-HND-VL

Cours 2 : JPA (Java Persistence Api)

- JPA
  - Introduction JPA
  - DAO générique et problématiques clés
- Annotation des attributs et des identifiants des entités
  - Cycle de vie des Entité JPA et API EntityManager
  - Aspect stayique : Mapping One-To-One et One-To-Many
  - Aspect stayique : Mapping Many-To-Many
  - Aspect dynamique : Chargement lazy, cascade
  - Aspect dynamique : Strategies de fetching et cascade
- ORM et Configuration JPA
- Librairie Jackson

JPA : Introduction

Toute application d'entreprise effectue des opérations de base de données en stockant et en récupérant de grandes quantités de données.

Malgré toutes les technologies disponibles pour la gestion du stockage, les développeurs d'applications ont généralement du mal à effectuer efficacement les opérations de base de données.

Pour les développeurs Java, en utilisant **JPA**, diminuent considérablement la charge d'interaction avec la base de données.
JPA constitue un pont entre nos **objets du domaine** (programme Java) et les **modèles relationnels** (programme de base de données).

Annotation des attributs et des identifiants des entités

```java
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Data
@Table(name = "logging_TBLE")
public class Logging implements Serializable {

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

    private String libelle;

    private Boolean active;

    @Version
    private int version;

}
```

Cycle de vie des Entité JPA et API EntityManager

Le cycle de vie de l'**EntityManager peut être géré de deux(2) manières:**

1- Gérer par l'utilisateur

2- Via **CDI** (**C**ontext and **D**ependency **I**njection)

Nous allons voir le cas 1. c-à-d cycle de vie de l'**EntityManager** gérer manuellement pour mieux comprendre le concept.

EntityManager

On obtient l' **EntityManager** à partir de l'**EntityManagerFactory** (qui est une fabrique d' **EntityManager**) via l'expression:

```
EntityManagerFactory emf =  Persistence.createEntityManagerFactory("persistenceUnitName");
EntityManager em = emf.createEntityManager();
```

Cycle de vie des Entité JPA et API EntityManager

Rappel: le paramètre “persistenceUnitName” est la valeur de l’attribut “name” de la balise <persistence-unit> provenant du fichier persistence.xml

Ainsi donc à chaque fois qu’on aura besoin de faire des opérations de transactions on fera ce traitement:

(1) EntityManagerFactory emf = Persistence.createEntityManagerFactory("persistenceUnitName");

Nous utiliserons le Design Pattern **Singleton** pour la création de l’ EntityManagerFactory le cas (1);

(2) EntityManager em = emf.createEntityManager();

(3) em.getTransaction().begin(); (pas nécessaire en lecture)

(4) Traitements…

Cycle de vie des Entité JPA et API EntityManager

(5) em.flush(); (à condition qu'un traitement de la même transaction est nécessaire; il synchronise notre base de données avec l'état actuel des objets contenus dans la mémoire nous permettant ainsi de libérer de la mémoire sans toutefois valider la transaction.)

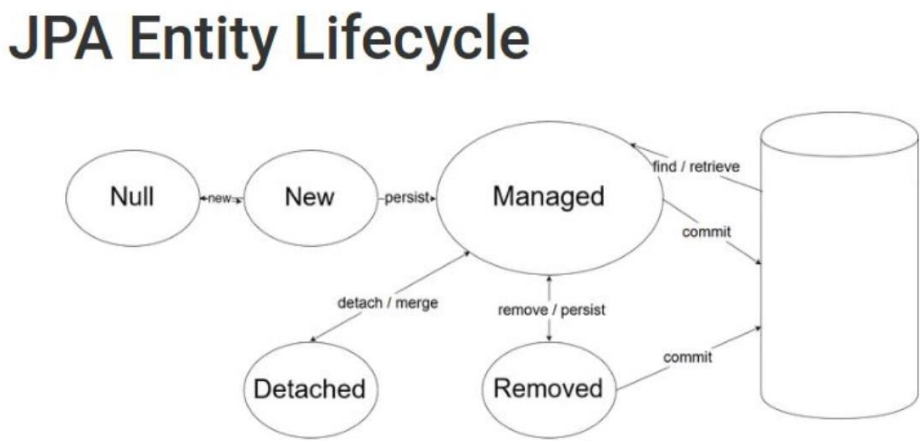
(6) em.getTransaction().commit(); (pas nécessaire en lecture il permet de valider la transaction en revanche cela met fin au EntityManager (em) plus besoin d'un em.close())

(7) em.close(); (utilisé quand il a eu une transaction débutée. très souvent non utilisé au profit de .commit)

Cycle de vie des Entité JPA et API EntityManager

Cycle de vie Entité JPA

L'EntityManager est le garant du cycle de vie des Entités JPA



Cycle de vie des Entité JPA et API EntityManager

L'**EntityManager** est le garant du cycle de vie des **Entités JPA**

Transient : private transient nom;le Champs n'existe pas encore dans la BD et ne sera pas pris en compte dans la BD ou MyObject myObject = new MyObject(); Objet existe uniquement dans le code mais non-manager par l'EntityManager et n'exist pas encore dans la BD on ne peut donc pas dire qu'il est "**Detached**"

Managed : l'étape où l'objet devient persistant et géré par EntityManager. Pour ce faire, nous devons appeler la méthode persist depuis une transaction. entityManager.getTransaction().begin();
entityManager.persist(**myObject**);
entityManager.getTransaction().commit();

Detached : Supprime l'objet de EntityManager, mais l'objet existe toujours dans la base de données. Certaines méthodes EntityManager sur un objet détaché entraîneront une exception IllegalArgumentException.
entityManager.detach(**myObject**);

Cycle de vie des Entité JPA et API EntityManager

Removed : Supprime l'objet de la base de données et tout comme persistent, cela doit également avoir lieu dans une transaction.

```
entityManager.getTransaction().begin();
entityManager.removed(myObject);
entityManager.getTransaction().commit();
```

NB : le transient appliqué sur un champ peut se faire aussi par annotation

```
@Transient
private String nom;
```

Aspect statique : Mapping One-To-One et One-To-Many

Avant de voir le Mapping **OneToOne** & **OneToMany** voyons d'abord quelques annotations utiles dans la déclaration d'une classe dit **Entité JPA**

la classe soit persistante

Obligatoire : - @java.persistance.Entity : placé au dessus de la classe à fin qu'elle soit pesistante

- @javax.persistence.Id : placé au dessus du champ identifient obligatoire pour que la la classe soit persistante

- La classe doit avoir un constructeur sans argument

Clé primaire : Une entité doit avoir un attribut qui correspond à la clé primaire de la table associée La valeur de cet attribut ne doit jamais être modifiée. Pour une clé composite, on utilise **@java.persistence.EmbeddedId** ou **@java.persistence.IdClass**

Le type de la clé primaire doit être d'un des types suivants:

1.type primitif Java 2. classe qui enveloppe un type primitif

3. java.lang.String 4. java.util.Date 5. java.sql.Date

Pour les clés primaire de type numérique on n'a la possibilité de générer leurs valeurs de façon automatique

Aspect statique : Mapping One-To-One et One-To-Many

@GeneratedValue indique que la clé sera automatiquement générée par le **SGBD**

l'annotation peut avoir un attribut **strategy** qui indique comment la clé sera générée

AUTO: le **SGBD** choisi (valeur par défaut)

SEQUENCE: il utilise une séquence SQL

IDENTITY: il utilise un générateur de type IDENTITY (auto increment dans MySQL par exemple)

TABLE: il utilise une table qui contient la prochaine valeur de l'identificateur

Aspect statique : Mapping One-To-One et One-To-Many

Autre Propriétés :

l'Annotation **@Column** et ces attributs

(**name**="nomDB"): nom de l'attribut

(**unique**=true): la valeur est-elle unique ? true/false

(**nullable**=false): accepte une valeur nulle ? true/false

(**insertable**=true): autorise ou non l'attribut à être mis à jour

(**table**=""): lorsque l'attribut est utilisé dans plusieurs tables

(**length**= 5): longueur max

(**scale**=4): nombre de chiffre après la virgule

(**precision**=3): précision pour le nombre des valeurs numériques

Aspect statique : Mapping One-To-One et One-To-Many

@Temporal

On peut ajouter l'annotation pour préciser le type d'un attribut

Ex: @Temporal(TemporalType.DATE)

private Date dateNaissance;

On a le choix entre DATE, TIME, TIMESTAMP

@Transient

Toutes les informations d'une classe n'ont pas besoin d'être persistantes par exemple, l'âge

Ex: @Transient

private Integer age;

Aspect statique : Mapping One-To-One et One-To-Many

Quelques méthodes de traitements avec EntityManager(em):

Soit : Object newObject = new Object();

em.persist(newObject); => pour enregistrer un objet dans la BD.

em.fin(Object.class, Pk); => pour la lecture d'un objet dans la BD.

em.merge(newObject); => pour la mise a jour cela sous-entend que l'objet existe déjà dans la BD.

em.remove(newObject); => pour la suppression cela sous-entend que l'objet existe déjà dans la BD.

em.createQuery("SELECT o FROM Object o").getResultList(); => pour lire tous les éléments de l'Objet dans la BD.

em.createNamedQuery(newObject);

em.createNativeQuery: à ce niveau JPA nous donne la latitude de pouvoir utiliser le language SQL

Aspect statique : Mapping One-To-One et One-To-Many

LES RELATION D'ASSOCIATIONS

Une association peut être uni- ou bi-directionnelle Cardinalités: **1-1**, **1-N**, **N-1**, **M-N**
Elles sont définies grâce à une annotation sur la propriété correspondante

Le **OneToOne (1-1) unidirectionnel:** caractérisé par**: l'annotation @OneToOne**
représente la clé étrangère dans la table proprietaire
dans le cas ci-dessous on aura le code suivant

Aspect statique : Mapping One-To-One et One-To-Many

OneToOne (1-1) unidirectionnel

```java
@Entity
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String commune;

    private String tel;

    public Adresse() {
    }
}
```

```java
@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "client")
public class Client implements Serializable {

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

    @Column(name="NOM", nullable = false)
    private String nom;

    @Column(name="PRENOM")
    private String prenom;

    @OneToOne
    @JoinColumn(name="ADRESSE_PK", nullable = false)
    private Adresse adresse;

    // getters and setters

}
```

Aspect statique : Mapping One-To-One et One-To-Many

Le **OneToMany (1-N)** Unidirectionnel**:**
Dans la DB nous aurons 2 tables(“maison” et “personne”) dont “maison” contient une colonne de clé étrangère de “personne” soit le code suivant:

Aspect statique : Mapping One-To-One et One-To-Many

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name="maison")
public class Maison implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String ville;
    @OneToMany
    private Collection<Personne> personne;
}
```

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "personne")
public class Personne implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String prenom;
    @ManyToOne(fetch = FetchType.LAZY)
    private Maison maison;
}
```

Aspect statique : Mapping One-To-One et One-To-Many

Le **OneToMany (1-N) bidirectionnel:**

Aspect statique : Mapping One-To-One et One-To-Many

Le **OneToMany (1-N) bidirectionnel:**

Une relation **1-N bidirectionnel** doit correspondre à une relation **N-1** dans la classe destination de la relation.

Comme pour le cas des relations 1:1, le caractère **bidirectionnel** d'une relation 1:p est marqué en définissant l'attribut **mappedBy** sur la relation.

JPA nous pose une contrainte ici : l'attribut **mappedBy** est défini pour l'annotation **@OneToMany**, mais pas pour l'annotation **@ManyToOne**. Or, comme nous l'avons vu dans le cas de l'annotation **@OneToOne**, **mappedBy** doit être précisé sur le côté esclave d'une relation. Dans le cas d'une relation **1-N bidirectionnel**, **JPA** ne nous laisse donc pas le choix de l'entité maître et de l'entité esclave. Voyons donc le code ci-dessou

Aspect statique : Mapping One-To-One et One-To-Many

Le **OneToMany (1-N)** Bidirectionnel**:**
Ici remarquez la présence de l'attribut mappedBy qui a pour valeur la propriété de portant l'annotation
OneToMany dans l'autre Classe.

Aspect statique : Mapping One-To-One et One-To-Many

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "personne")
public class Personne implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String prenom;
    @ManyToOne(fetch = FetchType.LAZY)
    private Maison maison;
}
```

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name="maison")
public class Maison implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String ville;
    @OneToMany(mappedBy = "maison", fetch = FetchType.LAZY)
    private Collection<Personne> personne;
}
```

Aspect statique : Mapping One-To-One et One-To-Many

NB: le **OneToMany** (Unidirectionnel et Bidirectionnel)

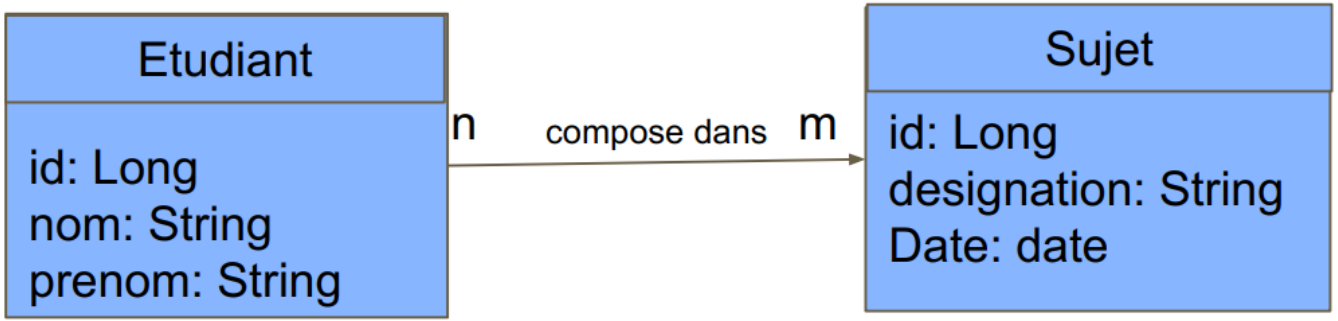
Dans ces deux cas, **JPA** créé juste 2 tables dont une comporte l'ID de l'autre; mais il est possible de créer une table de jointure entre les deux tables associées aux deux entités en y ajoutant l'annotation @JoinTable. Cette table de jointure porte une clé étrangère vers la clé primaire de la première table, et une clé étrangère vers la clé primaire de la deuxième table. Il est plus simple de faire le 1er choix.

Aspect statique : Mapping Many-To-Many

Le Mapping **Many-To-Many (N-M) unidirectionnel:** caractérisé par**: l'annotation @ManyToMany**

Une relation **N-M** est une relation de plusieurs vers plusieurs. La façon classique d'enregistrer ce modèle en base consiste à créer une table de jointure, et c'est ce que fait JPA.

Soit le diagramme de classe suivant:



Aspect statique : Mapping Many-To-Many

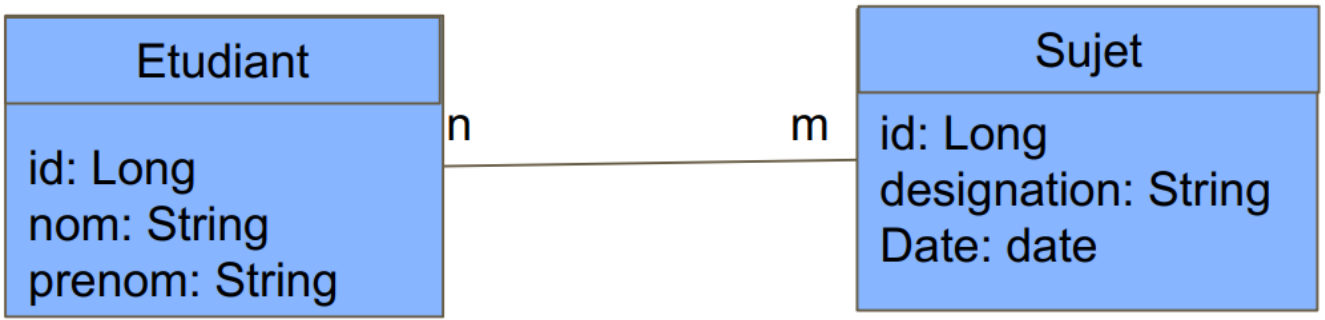
```java
@NoArgsConstructor
@Entity(name = "Sujet")
@Table(name = "sujet")
public class Sujet {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String designation;
    @Temporal(TemporalType.DATE)
    private Date dateSujet;
}
```

```java
@NoArgsConstructor
@Entity(name = "Etudiant")
@Table(name = "etudiant")
public class Etudiant {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String prenom;
    @ManyToMany
    private Collection<Sujet> sujets;
}
```

Aspect statique : Mapping Many-To-Many

Le Mapping **Many-To-Many (N-M) Bidirectionnel:** caractérisé par**: l'annotation @ManyToMany**

Dans le cas bidirectionnel, l'entité cible porte également une relation **@ManyToMany** vers l'entité maître. Cette relation doit comporter un attribut **mappedBy**, qui indique le nom de la relation correspondante dans l'entité maître.



Aspect statique : Mapping Many-To-Many

```java
@NoArgsConstructor
@Entity(name = "Etudiants")
@Table(name = "etudiants")
public class Etudiants {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
    private String prenom;
    @ManyToMany(fetch = FetchType.LAZY)
    private Collection<Sujets> sujets;
}
```

```java
@NoArgsConstructor
@Entity(name = "Sujets")
@Table(name = "sujets")
public class Sujets {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String designation;
    @Temporal(TemporalType.DATE)
    private Date dateSujet;
    @ManyToMany(mappedBy = "sujets", fetch = FetchType.LAZY)
    private List<Etudiant> etudiants;
}
```

Aspect dynamique : Chargement lazy, cascade

Dans la plupart des cas, le comportement cascade suffit à traiter les différentes actions à mener sur l'entité propriété

Mais il arrive que des effacements d'entité soient nécessaires sans qu'il y ait eu d'appel à la méthode remove() de l' EntityManager .

Avec l'exemple des entités **Client** et **Adresse**, effacer un **client** entraînera l'effacement de son **adresse**. Mais l'appel à **client**.**setAdresse**(null) doit aussi entraîner l'effacement de cet **adresse**, dans la mesure où cette
entité sera orpheline.

JPA 2.0 apporte une amélioration très intéressante sur JPA 1.0
D'où l'ajout de la propriété : orphanRemoval = true

Aspect dynamique : Chargement lazy, cascade

NB:
Le comportement *cascade* est précisé par l'attribut **cascade**, disponible sur les annotations :

@**OneToOne**,@**OneToMany** et @**ManyToMany**. La valeur de cet attribut est une énumération de type

CascadeType. En plus des valeurs **DETACH, MERGE, PERSIST, REMOVE, REFRESH**, cette énumération définit la
valeur **ALL**, qui correspond à toutes les valeurs à la fois

Aspect dynamique : Strategies de fetching et cascade

• **Chargement lazy et Chargement EAGER**
JPA nous permet de régler, relation par relation, la façon dont il doit se comporter :
• doit-il la charger par défaut ?
• doit-il la laisser vide, et la charger à la demande ?
On utilise pour cela l'attribut **fetch**, défini sur les annotations **@OneToOne, @OneToMany, @ManyToOne** et **@ManyToMany.**

Cet attribut peut prendre deux valeurs :
• **FetchType.LAZY** : indique que la relation doit être chargée à la demande ;
• **FetchType.EAGER** : indique que la relation doit être chargée en même temps que l'entité qui la porte.

Aspect dynamique : Strategies de fetching

- **Cascade**

Comme nous l'avons vu, l'**EntityManager** permet de mener à bien cinq opérations sur une entité :
DETACH,MERGE, PERSIST, REMOVE, REFRESH.
Le comportement *cascade* consiste à spécifier ce qui se passe pour une entité en relation d'une entité père (que cette relation soit monovaluée ou multivaluée), lorsque cette entité père subit une des opérations
définies ci-dessus.

Exple 1: le cascade={CascadeType.PERSIST, CascadeType.REMOVE} A chaque enregistrement ou suppression l'action est répercutée sur l'entité propriété

Exple 2 : le cascade= CascadeType.ALL Toute action est appliqué sur l'entité propriété

Aspect dynamique : Strategies de fetching

```
@OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
private Adresse adresse;
```

```
@OneToOne(cascade= CascadeType.ALL)
private Adresse adresse;
```

ORM et configuration JPA

Ainsi donc nous allons utiliser un **ORM** Java(**O**bject-**R**elational **M**apping) entendez par **M**apping **O**bject-**R**elational et l'**API Reflection** du **JDK** pour éviter la répétition de code.

A retenir:
JPA : **JAVA Persistence API** c'est un standard Java issue du groupe de travail d'experts **JSR** 220 (Java Specification Requests) on dira que c'est une un interface de programmation Java.
ORM: Object-**R**elational **M**apping; est un type de programme informatique qui se situe entre un programme applicatif et une base de données relationnelle afin de simplifier l'interaction Objet Java et BDR
EclipseLink : Est l'implémentation standard de l'interface JPA fournie par Oracle avec le serveur GlassFish

ORM et configuration JPA

Hibernate : **Tout comme EclipseLink** est une implémentations de l'interface JPA on dira donc que c'est un framework **ORM.** Il est fournie par **Red-Hat** .

Le fichier persistence.xml

Le fichier **persistence.xml** est un fichier de configuration standard dans JPA. Il doit être inclus dans le répertoire META-INF du fichier JAR contenant les beans entité. Le fichier persistence.xml doit définir une unité de persistance avec un nom unique dans le chargeur de classes à portée courante. L'attribut provider spécifie l'implémentation sous-jacente de l'EntityManager JPA.

ORM et configuration JPA

le persistence unit caractérisé par la balise: <persistence-unit name=”pu” transaction-type=”xxxx”>
l'attribut “**name**” est un nom unique dans le projet via lequel on obtient les informations de connection à la base de donnée utile pour l’**EntityManager**
Une **unité de persistance** définit l'ensemble de toutes les classes liées ou groupées par l'application et qui doivent être colocalisées dans leur mappage vers une base de données unique l’attribut **“transaction-type” définit le type de transactions soit:**
Transactions locales à une ressource “RESOURCE_LOCAL” (fournies par JDBC associées à une seule base de données)
Transactions JTA “JTA” (plus de fonctionnalités que les transactions JDBC elles peuvent travailler avec plusieurs bases de données)
JTA = (Java Transaction API)

ORM et configuration JPA

l'interface **EntityManager**: l' EntityManager est associée à un contexte de persistance
Dans le contexte de persistance, les instances d'entité et leur cycle de vie sont gérés
Utilité : L'interface **EntityManager** est utilisée pour créer et supprimer des instances d'entités persistantes, pour rechercher des entités à l'aide de leur clé primaire et pour interroger des entités.
On obtient un **EntityManager** à partir d'un **EntityManagerFactory** c'est une interface utilisée pour interagir avec l'usine du gestionnaire d'entités pour l'unité de persistance.
Comparaison des contextes de persistance **RESOURCE_LOCAL** et **JTA**
Avec *<persistence-unit transaction-type = "RESOURCE_LOCAL">* **VOUS** êtes responsable de la création et du suivi de EntityManager (PersistenceContext / Cache)

ORM et configuration JPA

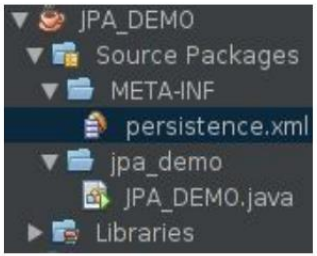
- Vous **devez** utiliser **EntityManagerFactory** pour obtenir un **EntityManager**
- L' instance **EntityManager** résultante **est** un PersistenceContext / Cache
- Un **EntityManagerFactory** peut être injecté par l'intermédiaire du **@PersistenceUnit** annotation uniquement (pas @PersistenceContext)
- Vous n'êtes **pas** autorisé à utiliser @PersistenceContext pour faire référence à une unité de type **RESOURCE_LOCAL**
- Vous **devez** utiliser l' API **EntityTransaction** pour commencer / valider **chaque** appel de votre **EntityManger**
- L'appel de **entityManagerFactory.createEntityManager ()** deux fois résulte en deux instances **EntityManager** séparées et donc **deux** PersistenceContexts / Caches séparés.
- Ce n'est **presque jamais** une bonne idée d'avoir plus d'une **instance** d'un **EntityManager** en cours d'utilisation (n'en créez pas une seconde à moins d'avoir détruit la première)

ORM et configuration JPA

Avec <*persistence-unit transaction-type* = "***JTA***"> le **CONTAINER** fera EntityManager (PersistenceContext / Cache) créer et suivre ...

- Vous **ne pouvez pas** utiliser **EntityManagerFactory** pour obtenir un EntityManager
- Vous ne pouvez obtenir qu'un **EntityManager** fourni par le **conteneur**
- Un **EntityManager** peut être injecté via l' annotation **@PersistenceContext** uniquement (pas @PersistenceUnit)
- Vous n'êtes **pas** autorisé à utiliser @PersistenceUnit pour faire référence à une unité de type JTA
- Le **EntityManager** donné par le conteneur est une **référence** au PersistenceContext / Cache associé à une Transaction JTA.
- Si aucune transaction JTA n'est en cours, EntityManager **ne peut pas être utilisé** car il n'y a pas PersistenceContext /Cache.
- Tout le monde avec une référence EntityManager à la **même unité** dans la **même transaction** aura automatiquement une référence au **même PersistenceContext / Cache**
- Le PersistenceContext / Cache est **rincé** et effacé à JTA **engager** le temps

ORM et configuration JPA




```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" 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">
  <persistence-unit name="JPA_DEMOPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpa_demo?
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.password" value="root"/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Librairie Jackson

La classe *Jackson ObjectMapper* (com.fasterxml.jackson.databind.ObjectMapper) est le moyen le plus simple d'analyser JSON avec Jackson. Le Jackson ObjectMapper peut analyser le JSON à partir d'une chaîne, d'un flux ou d'un fichier, et créer un objet Java ou un graphique d'objets représentant le JSON analysé. L'analyse de JSON en objets Java est également appelée *désérialisation des objets Java de JSON* .

Le *Jackson ObjectMapper* peut également créer du JSON à partir d'objets Java. La génération de JSON à partir d'objets Java est également appelée *sérialisation d'objets Java en JSON* .

Le mappeur d'objets Jackson peut analyser JSON en objets de classes développés par vous ou en objets du modèle d'arborescence JSON intégré.

À propos, la raison pour laquelle il est appelé *ObjectMapper* est qu'il mappe JSON en objets Java (désérialisation) ou des objets Java en JSON (sérialisation).

Sources

http://objis.com/
http://tutorials.jenkov.com/java-json/jackson-objectmapper.html