J.P.A.
(Java Persistence Api)
et
Hibernate

Table des matières

I - Mapping Objet-Relationnel (ORM) et JPA.....3
1. Problématique "O.R.M."....3
2. JPA (Java Persistance Api)....4

II - Architecture d'Hibernate (historique,évolution).....6
1. Hibernate: ancienne architecture & configuration....6
2. Hibernate en tant qu'implémenation de JPA 2....8

III - JPA : architecture & configuration.....9
1. Présentation de JPA (Java Persistence Api)....9
2. Unité de Persistance (META-INF/persistence.xml)....9
3. Configuration du mapping JPA via annotations....13

IV - EntityManager et entités persistantes.....16
1. Entity Manager et son contexte de persistance....16
2. Transaction JPA....18
3. Différents états - objet potentiellement persistant....19
4. Cycle de vie d'un objet JPA/Hibernate....19
5. Synchronisation automatique dans l'état persistant....20
6. objet persistant et architecture n-tiers....20

7. Principales méthodes JPA / EntityManager et Query....20
8. Contexte de persistance et proxy-ing....22

V - Langage de requêtes JPQL....24

VI - O.R.M. JPA (généralités)....28

1. Vue d'ensemble sur les entités, valeurs et relations....28
2. Identité d'une entité (clef primaire)....30
3. Propriétés d'une colonne....31
4. Relations (1-1, n-1, 1-n et n-n)....32

VII - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...)....33

2. Relations d'héritage & polymorphisme....43

VIII - JPA : aspects divers et avancés....47

1. Quelques spécificités de JPA....47
2. Cycle de persistance (pour @Entity) et annotations/callbacks associées pour "Listener"....49
3. Map et apports de JPA 2.0....49
4. Api "Criteria" de JPA 2....54

IX - Outils pour JPA , ...58

1. Approches top-down , down-top , ...58
2. Projet "JPA" d'eclipse....59

X - Configuration JPA pour TP....61

1. Projet appliSpringJPa (pour TP)....61
2. Premiers TPs sans SpringBoot (JPA seul)....64
3. Configuration de base de données par scripts....68
4. Config JPA avec SpringBoot et transactions....69

I - Mapping Objet-Relationnel (ORM) et JPA

1. Problématique "O.R.M."

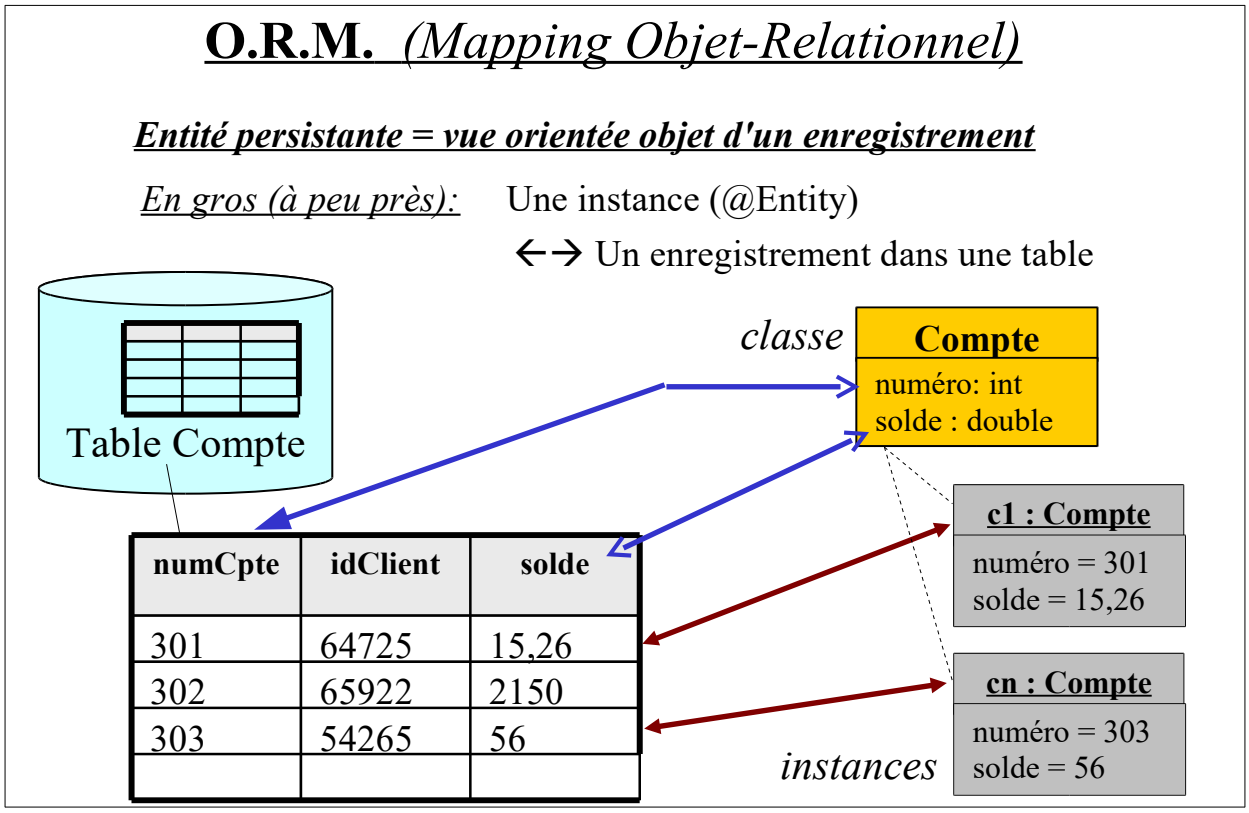
1.1. Objectif & contraintes:

L'objectif principal d'une technologie de **mapping objet/Relationnel** est d'établir une **correspondance relativement transparente et efficace** entre :

un **ensemble d'objets en mémoire**

et

un **ensemble d'enregistrements d'une base relationnelle.**



Une telle technologie doit permettre à **un programme orienté objet de ne voir que des objets** dont certains sont des **objets persistant**s. ***Le code SQL est en très grande partie caché*** car les objets persistants sont automatiquement pris en charge par la technologie "O.R.M."

Autrement dit , **le code SQL n'est plus (ou très peu) dans le code "java" mais est généré automatiquement à partir d'une configuration de mapping** (*fichiers XML* , annotations, ...) .

Contraintes : pour être **exploitable** , une **technologie "O.R.M."** se doit d'être :

- **simple** (à configurer et à utiliser) et **intuitive**
- **portable** (possibilité de l'intégrer facilement dans différents serveurs d'application)
- **efficace** (bonnes performances, fiable , ...)
- capable de gérer les **transactions** ou bien de participer à une transaction déjà initiée
- ...

1.2. Eléments techniques devant être bien gérés

Au delà des simples associations élémentaires du type
"**1 enregistrement d'une table <---> 1 instance d'une classe**" ,
une **technologie "O.R.M." sophistiquée** doit savoir gérer certaines **correspondances évoluées** :

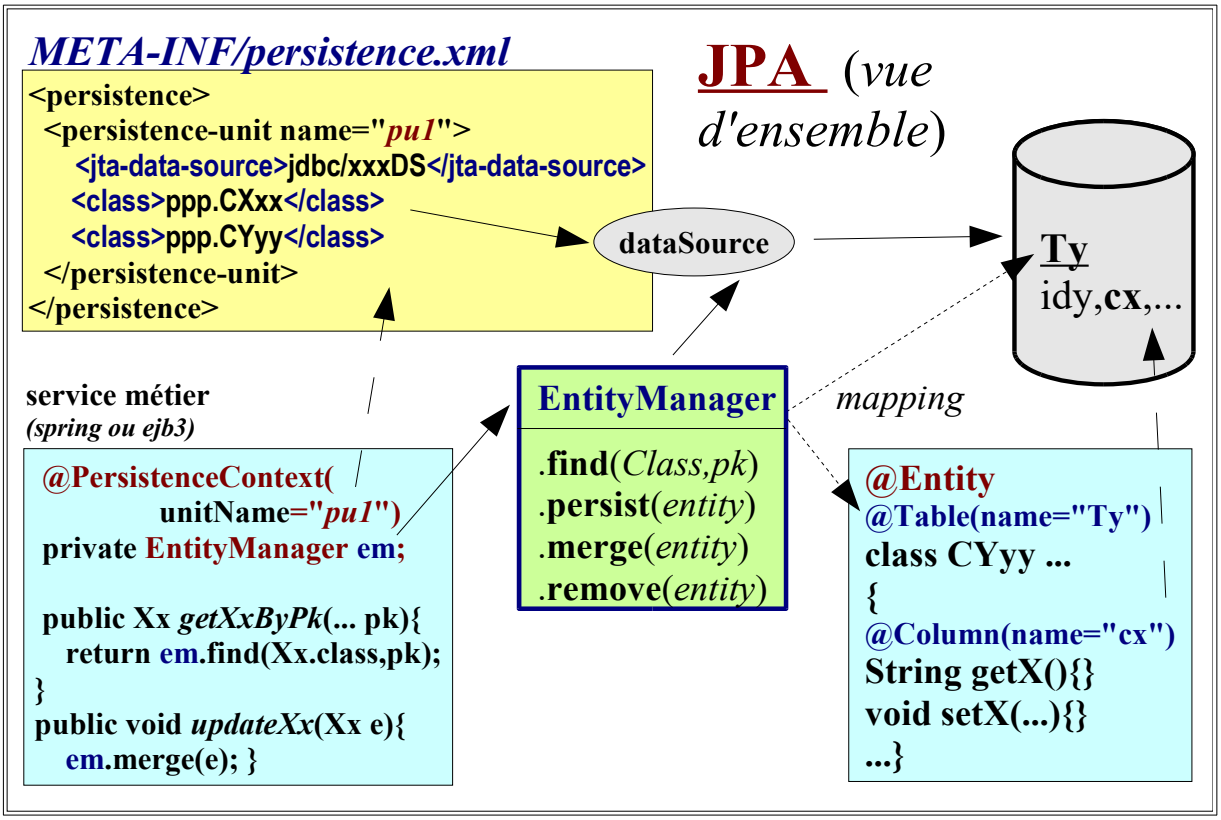
- **Associations "1-1" , "1-n" , "n-n"** (jointures entre tables <---> relations entres objets)
- **Objets "valeur"** (sous parties d'enregistrements , tables secondaires , ..)
- **Généralisation/Héritage/Polymorphisme** (<---> schéma relationnel ?)
- ...

Une technologie "O.R.M." doit sur ces différents points être **flexible et efficace** .

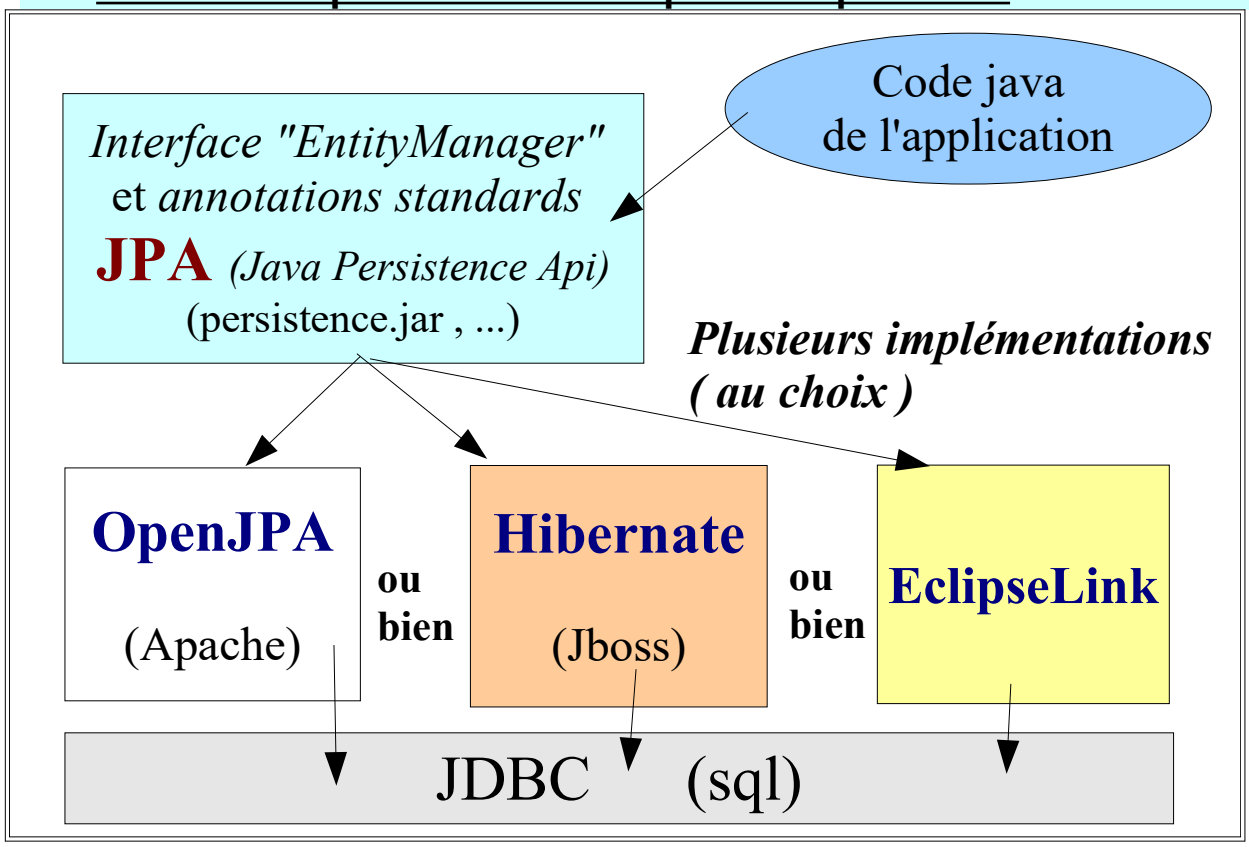
2. JPA (Java Persistance Api)

JPA (Java Persistence Api) et @Entity

- **API O.R.M. officielle** de **Java EE**
- **Configuration** basée sur des **annotations** *(avec configuration xml possible : orm.xml).*
- Utilisable dans les contextes suivants:
  - dans **module d'EJB3**
  - dans **module Spring**
  - de façon **autonome** (java sans serveur)
- Plusieurs **implémentations** internes disponibles (***OpenJPA*** , ***Hibernate*** depuis version 3.2 , ...)



2.1. Plusieurs implémentations disponibles pour JPA



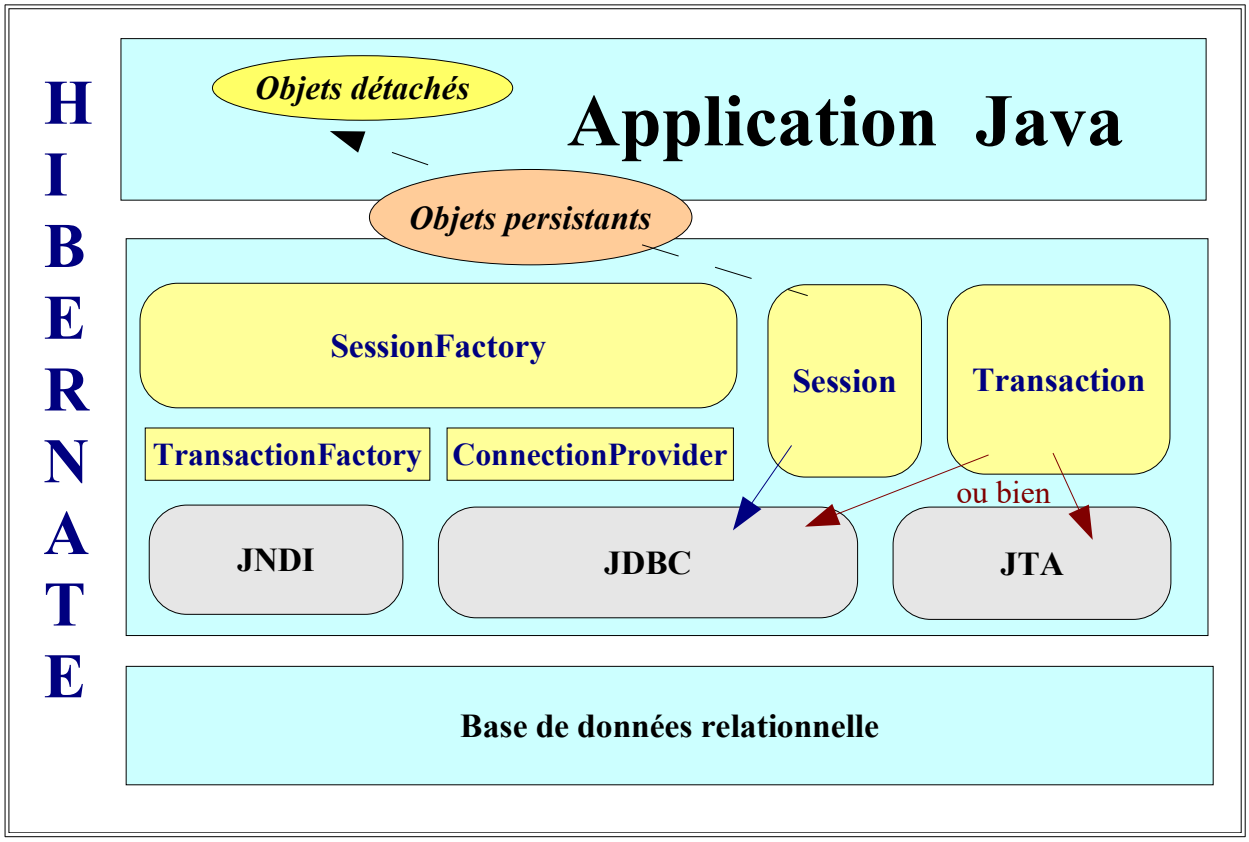
Selon le contexte, l'implémentation de JPA sera libre ou bien quasi imposée par le serveur d'applications (ex : JBoss Application Serveur utilise en interne Hibernate pour implémenter JPA).

II - Architecture d'Hibernate (historique,évolution)

1. Hibernate: ancienne architecture & configuration

1.1. Architecture historique d'hibernate (sans JPA)

NB : Vers environ 2002, 2003, 2004 , les premières versions d'hibernate avaient mises au point avant JPA et étaient structurées de cette façon :



NB: Les transactions (commit/rollback) sont supervisées à partir de l'objet abstrait "Transaction" d'Hibernate . Celui-ci peut soit s'appuyer sur des transactions simples JDBC (sur une seule base de données) ou bien sur des transactions complexes "JTA" gérées par le serveur d'application hôte (ex: WebSphere, Weblogic, Jboss, Geronimo, ...).

D'autre part, l'utilisation exacte de l'objet "Transaction" d'hibernate est assez variable en fonction du style de programmation utilisé (explicite/implicite , déclaratif , ...).

L'objet "Session" d'hibernate est de loin le plus important : il correspond à une sorte de "D.A.O." générique permettant de déclencher les opérations habituelles de persistance (CRUD : Create, Retreive, Update, Delete). L'objet "session" d'hibernate est soit "ouvert" ou "fermé" et ceci a une très grande incidence sur la connexion JDBC/SQL sous-jacente et sur l'état des objets (persistants ou détachés) .

1.2. Principaux objets de l'API Hibernate (sans JPA) :

SessionFactory	Un objet de ce type par base de données. cet objet représente un **cache de tous les mappings "objet/relationnel" liés à une certaine base de données**. Les traitements associés à cet objet sont **ré-entrants (multi-threading possible)**. Cet objet sert à fabriquer des objets "Session".
Session	Un objet de ce type représente une **session utilisateur** et est lié à un seul thread. Une session englobe un certain nombre d'**objets persistants** dont les valeurs sont mappées avec le contenu de certaines tables relationnelles. Chaque objet session incorpore une connexion JDBC et sert à récupérer ou fabriquer des objets de type "Transaction"
Transaction	Un objet de l'api hibernate représentant une **transaction que l'on peut explicitement délimiter** (begin , commit/rollback) . cet objet se base sur des transactions sous-jacentes de type JDBC, JTA ou ... .
Objets persistants	Objets java (JavaBean / POJO) comportant des valeurs persistantes et des traitements "métier" . Tant qu'ils sont liés à une session, ces objets ont des valeurs synchronisées/synchronisables avec la base de données. **Dès que la session est fermée, ces objets deviennent détachés et peuvent être directement réutilisés pour tout transfert ou affichage**.
...	

1.3. Anciens fichiers de configuration (selon variantes)

Utilisation d'Hibernate	*Configuration générale (base de données , ...)*	*Configuration fine du mapping objet relationnel*
Hibernate seul	**hibernate.cfg.xml**	xxx.**hbm.xml** , yyy.**hbm.xml**
Hibernate intégré dans Spring	Partie de **springConf.xml**	xxx.**hbm.xml**, yyy.**hbm.xml**
Hibernate intégré dans JPA (lui même intégré dans EJB3 ou Spring ou ...)	**META-INF/persistence.xml** (+ .... ou ....)	**annotations (@Entity ,@Table , @Column, ...) dans le code java des entités**

Structure ordinaire du code de l'application allant autour d'Hibernate:

DAO (ou Test) avec **Hibernate seul**	Ordres O.R.M. via "***session***" + **gestion explicite** des **transactions** + gestion explicite de la session
DAO ou Service avec ***"Spring+Hibernate"***	Ordres O.R.M. via ***hibernateTemplate*** (+ try/catch)
Test ou DAO avec **JPA (sans EJB3 ni Spring)**	Ordres O.R.M. + **gestion explicite** des **transactions** + gestion explicite "entityManager"
Service avec ***JPA*** (intégré dans ***Spring*** ou ***EB3***)	Ordres O.R.M. via "***entityManager***" (+try/catch)

2. Hibernate en tant qu'implémenation de JPA 2

- Avant 2005/2006 , Hibernate était souvent utilisé seul ou avec spring et JPA n'existait pas encore .
  A cette époque les annotations (@…) n'existaient pas encore et tout se configurait en xml (application.xml de spring + hibernate.cfg.xml + xxx.hbm.xml)

- **Dès 2005/2006 et EJB3/JPA1, Hibernate a été restructuré pour devenir une implémentation de JPA 1** .
  Les projets de cette époque sont très progressivement passé d'une configuration purement xml à une configuration "xml + annotations) : META-INF/persistence.xml + configuration spring xml + @Entity , @ManyToOne, …

  JPA 1.0 étant fonctionnellement moins riche que Hibernate , certains projets des années 2007, 2008 utilisaient encore à cette époque certaines parties d'Hibernate en direct (sans systématiquement se restreindre au standard JPA) .

- **A partir de 2009,2010 et l'avènement de JPA 2, presque toutes les fonctionnalités d'Hibernate sont devenues accessibles via JPA** . Il n'y a maintenant plus vraiement d'intérêt à utiliser Hibernate en direct sans passer par le standard JPA .

  Les versions JPA 2.1 de 2011 , JPA 2.2 de 2017 et JPA 3.0 de 2020 sont venues conforter cet état de fait .

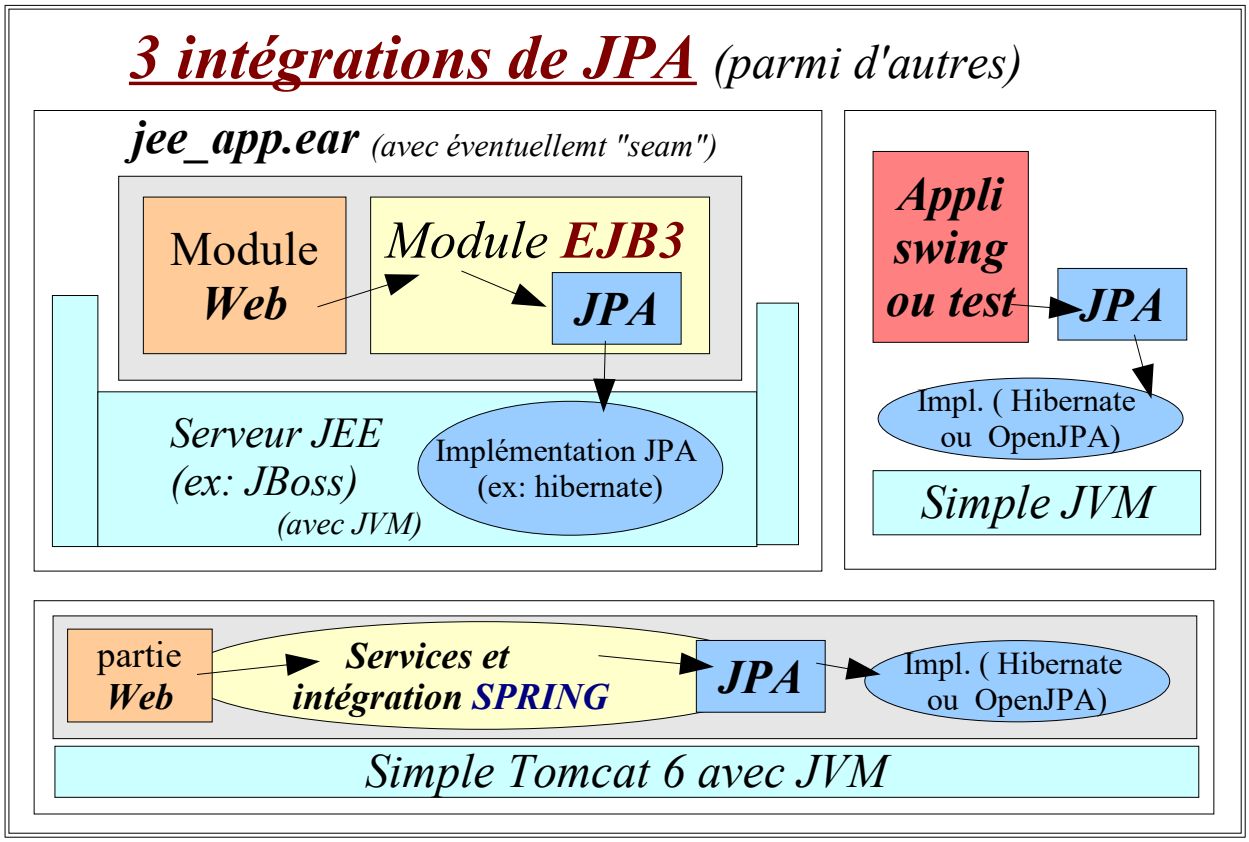
- Bien qu'utilisable "toute seule" (sans spring ni EJB et via une gestion explicite des transactions), **la technologie JPA/Hibernate est, au sein des vrais projets d'entreprises, quasi systématiquement utilisée de manière intégrée à la technologie EJB 3.x ou bien Spring de manière à bénéficier de la gestion automatique et implicite des transactions (selon le standard Java EE)** .

- A partir de 2015 environ, la configuration du framework spring a petit à petit évolué d'une configuration "XML + annotations" à une configuration "javaConfig + annotations + .properties/.yaml) et **la plupart des projets modernes sont basés maintenant sur une intégration de JPA/Hibernate dans SpringBoot + SpringDataJpa** .

III - JPA : architecture & configuration

1. Présentation de JPA (Java Persistence Api)

L'api JPA peut être utilisée de façon très variable (dans EJB3 , dans Spring ou seul)

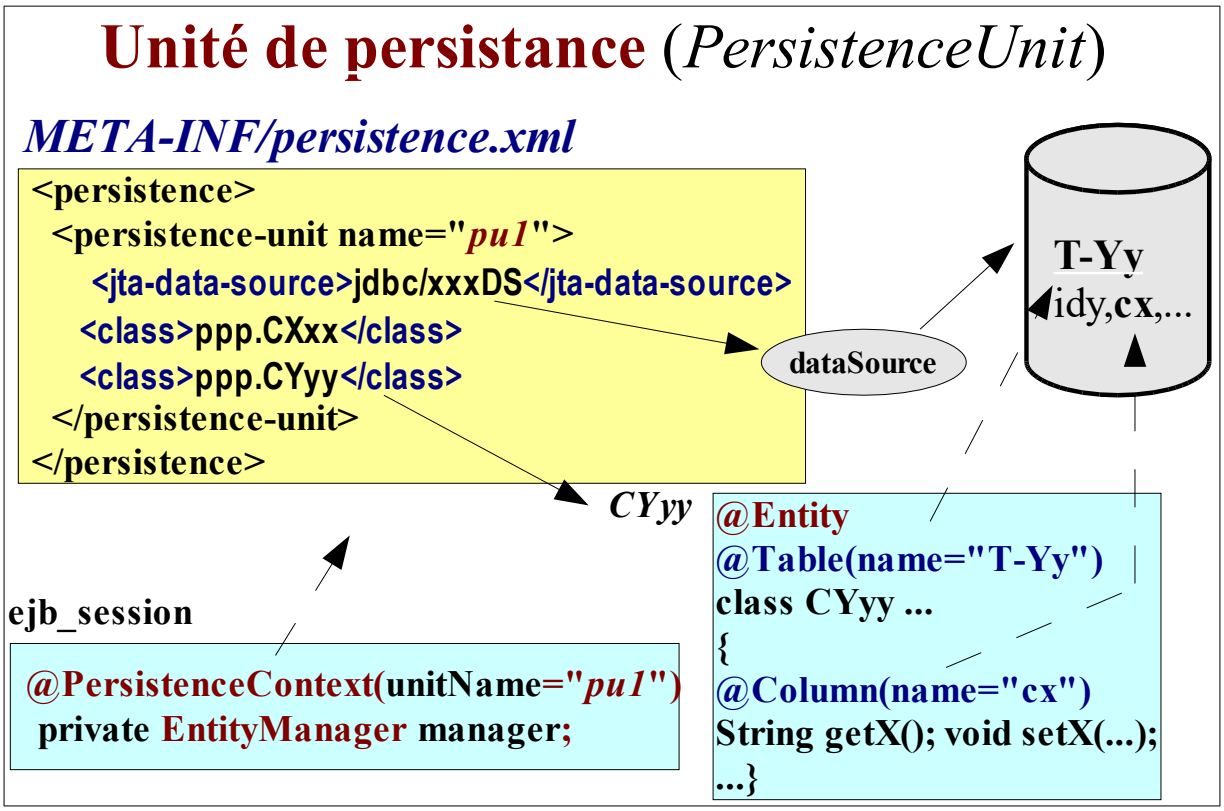


Une intégration "EJB3" s'appuie essentiellement sur ce qui est offert par le serveur d'applications (Jboss, WebSphere, ...). Par exemple JBoss 4.2.3 ou 5.1 comporte déjà en lui toutes les librairies (.jar) correspondant à l'API JPA (et son implémentation basée sur Hibernate).

Une intégration "Spring" s'appuie par contre généralement sur des librairies choisies (Hibernate ou OpenJPA ou ....) et qui sont quelquefois livrées et déployées avec l'application (dans WEB-INF/lib par exemple) .

Une classe de Test "Junit" exécutée avec une simple JVM peut très bien utiliser directement l'API JPA (avec ou sans Spring) en s'appuyant sur une des implémentations disponibles (Hibernate ou OpenJPA ou ...) .

2. Unité de Persistance (META-INF/persistence.xml)



META-INF/persistence.xml (exemple)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"  version="2.0"
      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_2_0.xsd" >
    <persistence-unit    name="OrderManagement">
       <description>....</description>
          <jta-data-source>java:/MyOrderDS</jta-data-source>
          <!-- <mapping-file>orm.xml</mapping-file> -->
          <class>com.widgets.Order</class>          <class>com.widgets.Customer</class>
          <properties>
             <!--  <property name="hibernate.hbm2ddl.auto" value="create-drop"/> -->
              <property  name="hibernate.dialect"  value="org.hibernate.dialect.MySQLDialect"/>
               <property  name="hibernate.show_sql"  value="true"/>
           </properties>
        </persistence-unit>
</persistence>
```

<persistence-unit> comporte un attribut optionnel "***transaction-type***" dont les valeurs possibles sont "***JTA***" ou "***RESOURCE_LOCAL***".
La valeur par défaut est "JTA" dans un env. EE (ex : EJB3) et "RESOURCE_LOCAL" dans un env. SE.

La valeur de **<jta-data-source>** ou **<non-jta-data-source>** correspond au ***nom JNDI global de la source de données SQL*** (pool de connexions permettant d'atteindre une base de données précise).
[Rappel : le serveur Jboss A.S. ajoute le préfixe "java:/" sur les noms JNDI des "dataSources"]

L'ensemble des classes (avec annotations @Entity) devant être prises en compte et prises en charge par le contexte de persistance se définit via une (ou plusieurs et complémentaires) indication(s) suivante(s):

- **Une liste explicite de classes** (balises ***<class>***)
- un ou plusieurs **fichier(s) xml de mapping "objet/relationnel"** (balises ***<mapping-file>***)
- ….

NB: Dans le cas particulier d'un module EJB3 ou Spring moderne, les classes comportant des annotations @Entity seront automatiquement détectées et il n'est donc pas absolument nécessaire de les déclarées dans le fichier META_INF/persistence.xml

NB: un fichier xml de type **orm.xml** (rare et facultatif) constitue soit une alternative vis à vis d'une configuration basée sur des annotations Java5 , soit un mécanisme de redéfinition.

Dans le cas où JPA est implémenté via la sous couche Hibernate , on peut éventuellement fixer la propriété ***hibernate.hbm2ddl.auto*** à l'une des valeurs suivantes :

* **create-drop** [créer les tables au démarrage de l'application et les supprime à l'arrêt]
* **update** [mise à jour des tables sans suppression]
* **none (ou le tout en commentaire)**

===> Les "create table" sont générés à partir des informations @Table , @Column , ….
===> Cette fonctionnalité peut être pratique en mode développement mais est très dangereuse en mode production !!!

Variante de META-INF/persistence.xml pour intégration dans Spring :

```
<persistence version="2.2" 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">
<persistence-unit name="myPersistenceUnit"
            transaction-type="RESOURCE_LOCAL">
 <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <!-- ...   <class>entity.persistance.jpa.XxxYyy</class> -->
 <properties>
   <property name="hibernate.dialect"  value="org.hibernate.dialect.MySQL8Dialect" />
        <!--    <property name="hibernate.hbm2ddl.auto" value="update" /> -->
  <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
  <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/mydb" />
  <property name="javax.persistence.jdbc.user" value="root" />
  <property name="javax.persistence.jdbc.password" value="root" />
</properties>
</persistence-unit>
</persistence>
```

NB: **<provider>**...HibernatePersistenceProvider**</provider>** permet de préciser la technologie choisie pour implémenter JPA (Hibernate ou eclipseLink ou OpenJpa) .

2.1. Evolution de la configuration JPA

Durant les années 2005-2015 , la technologie JPA 1 (puis JPA2) avait été principalement utilisée de manière intégrée à des applications "JEE/EJB3" ou bien "Spring-framework" .

- Dans le cadre (originel) d'une **application "JEE/EJB3"** , JPA est configuré via le fichier standard **META-INF/persistence.xml** et au sein de ce fichier la balise **<jta-data-source></jta-data-source>** référence un nom logique de "dataSource" (objet de configuration spécifique au serveur qui indique comment de connecter à une base relationnelle) .
  La configuration de ce dataSource est spécifique à un type et à une version du serveur JavaEE (ex : WebSphere-AS 7 ou Jboss-widfly 16 ou WebLogic ou Tomee ou …).
  Par exemple, le serveur ***Jboss-Wildly_ou_EAP*** configure un "dataSource" dans le fichier ***standalone.xml*** .

- Dans le cadre d'une application Spring (Spring-framework sans spring-boot) , la configuration JPA était **partiellement** stockée dans le fichier standard **META-INF/persistence.xml** et **avec un complément de configuration dans un fichier de configuration XML de Spring (ex : application.xml)**comportant la configuration "au format spring" d'un dataSource (pour se connecter à une base de données)

A partir de 2015, le framework "Spring" a beaucoup évolué *(configuration simplifiée sans xml , démarrage sans serveur , ...)* et la plupart des applications modernes sont maintenant basées sur **Spring-Boot** .
"**spring-boot-jpa**" se configure maintenant directement dans le fichier **application.properties** (ou ***.yml***) et dans ce contexte le fichier ~~**META-INF/persistence.xml**~~ n'est plus analysé/pris en compte.

Adaptation spring-boot :

application.properties

```
server.servlet.context-path=/appliSpringJpa
server.port=8080
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/mydb?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# si l'option spring.jpa.hibernate.ddl-auto=create est activée, toutes les
# tables nécessaires seront re-crées automatiquement à chaque démarrage
# à vide et en fonction de la structure des classes java (@Entity)
spring.jpa.hibernate.ddl-auto=create

#enable spring-data (generated dao implementation classes)
#spring.data.jpa.repositories.enabled=true
```

NB : une propriétée JPA/Hibenate est configurée via

<property name="javax.persistence.xyz" value="xyz-value" /> dans le fichier standard ***META-INF/persistence.xml***

et via **spring.jpa.properties.javax.persistence.xyz=xyz-value** dans ***application.properties de Spring-boot***

Options secondaires (disponibles en version 2.2 de JPA)

```
#tables créés automatiquement au démarrage et fichier sql déclenché automatiquement :
...javax.persistence.schema-generation.database.action=drop-and-create
...javax.persistence.sql-load-script-source=META-INF/data.sql


#fichiers sql générés (pour consultation) mais pas déclenchés:
...javax.persistence.schema-generation.create-source=metadata
...javax.persistence.schema-generation.scripts.action=drop-and-create
...javax.persistence.schema-generation.scripts.create-target=src/main/script/create.sql
...javax.persistence.schema-generation.scripts.drop-target=src/main/script/drop.sql
```

3. Configuration du mapping JPA via annotations

Le mapping de JPA se configure génralement en insérant quelques annotations JAVA dans le code JAVA des entités.

3.1. Exemple d'entité persitante (@Entity)

```
@Entity
@Table(name = "T_CUSTOMER")
public class Customer implements Serializable {

@Id
private Long number;

@Column(name = "customer_name")
private String name;
private Address address;

@OneToMany(...)
private List<Order> orders = new ArrayList<Order>();
```

```
@ManyToMany(...)
private List<PhoneNumber> phones = new ArrayList<PhoneNumber>();

public Customer() {} // No-arg constructor

public Long getNumber() {return number;}  // pk
public void setNumber(Long number) {this.number = number;}

public String getName() {return name;}
public void setName(String name) {this.name = name;}
public Address getAddress() {return address;}
public void setAddress(Address address) {this.address = address;}

public List<Order> getOrders() {return orders;}
public void setOrders(List<Order> orders) {
                        this.orders = orders;}
public List<PhoneNumber> getPhones() {return phones;}
public void setPhones(List<PhoneNumber> phones) {
                        this.phones = phones;}

// Business method to add a phone number to the customer
public void addPhone(PhoneNumber phone) {
this.getPhones().add(phone);
// Update the phone entity instance to refer to this customer
phone.addCustomer(this);}
}
```

Dans l'exemple précédent :

- l'annotation **@Entity** permet de marquer la classe Customer comme une classe d'objets persistants
- l'annotation **@Table(name="T_CUSTOMER")** permet d'associer la classe java "Customer" à la table relationnelle "T_CUSTOMER" .
- l'annotation **@Column(name="customer_name")** permet d'associer l'attribut "name" de la classe "Customer" à la colonne "customer_name" de la table relationnelle.
- l'annotation **@Id** permet de marquer le champ "number" comme **identifiant de l'entité (et comme clef primaire de la table)**.
- les autres annotations (**`@OneToMany`** , ...) seront approfondies dans un chapitre ultérieur.

NB: **Lorsqu'une annotation** (@Column ou ...) **n'est pas présente , les noms sont censés coïncider** (sachant que les minuscules et majuscules ont quelquefois de l'importance du coté SQL/relationnel selon le système d'exploitation hôte (ex: linux)).

Remarques importantes:

- Les annotations paramétrant le mapping objet/relationnel (@Id , @Column, @OneToMany, ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "get".

[au dessus des "private" --> plus lisible car dans le haut de la classe]
[au dessus des "get" --> plus clair et/ou plus fonctionnel en cas d'héritage]

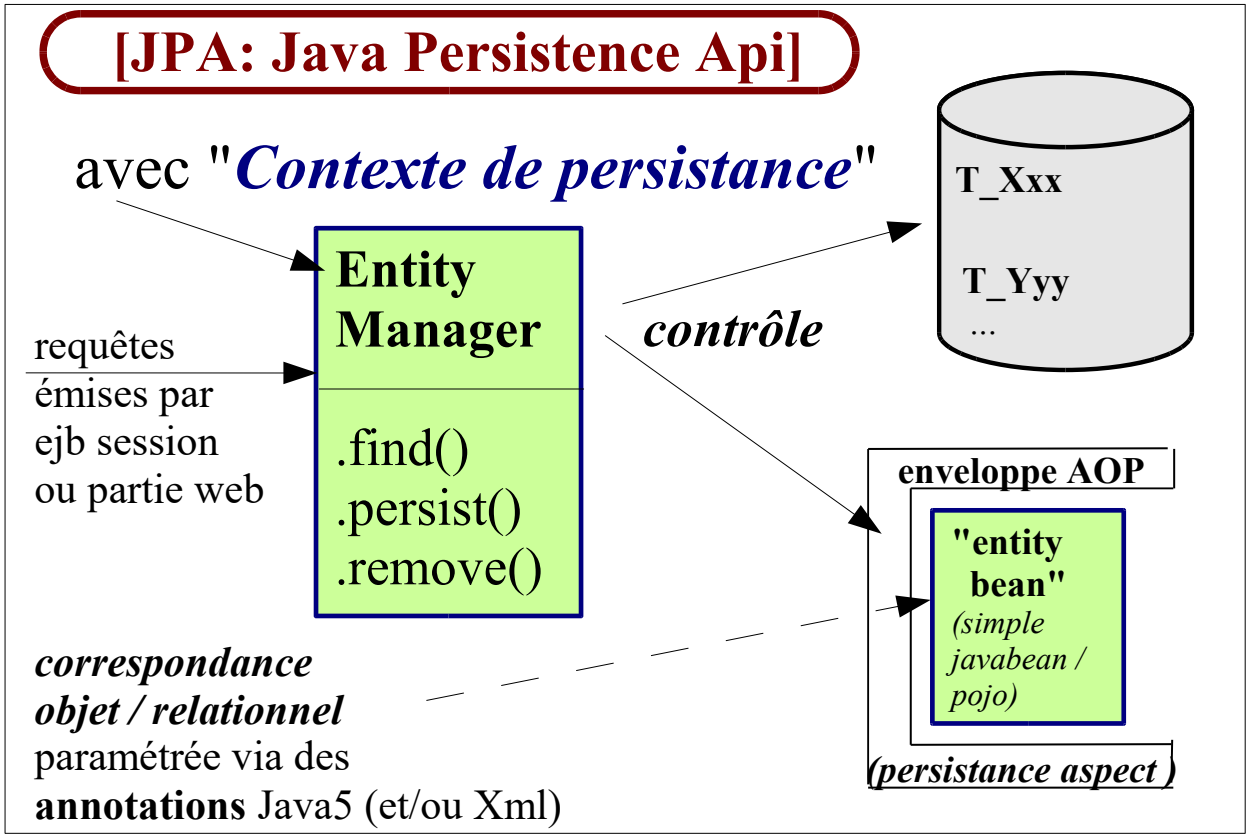
Attention: il faut absolument garder un style homogène (si quelques annotations au dessus des "privates" et quelques annotations au dessus des "getXxx()" --> ça ne fonctionne pas !!! sauf si annotation spéciale **@Access** de JPA 2 avec *AccessType.FIELD* ou *PROPERTY*)

- Les annotations paramétrant des injections de dépendances (@Resource , @PersistenceContext , ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "set".

- Même si une propriété (avec private +get/set) n'est marquée par aucune annotation , elle sera tout de même prise en compte lors du mapping objet-relationnel.
- Pour éventuellement (si nécessaire) désactiver le mapping d'une propriété d'une classe persistante, il faut utiliser l'annotation "**@Transien**t" (au dessus du "private" ou du "getter"de la propriété).
- **@Enumerated(*EnumType.STRING*)** permet se stocker la valeur d'une énumération sous forme de chaîne de caractères dans une colonne d'une table relationnelle.
- **@Temporal**(***TemporalType.DATE*** ou ***TemporalType.TIME*** ou ***TemporalType.TIMESTAMP***) permet de préciser la valeur significative d'une date java (java.util.Date) .
- ...

IV - EntityManager et entités persistantes

1. Entity Manager et son contexte de persistance



Un objet persistant (@Entity avec toutes ces annotations) doit être pris en charge par un **gestionnaire d'objets persistants** (implicitement associé à un "***contexte de persistance***") de façon à ce que la liaison avec la base de données soit bien gérée.

Ci après , figure l'interface prédéfinie "**EntityManager**" .

Celle-ci comporte tout un tas de **méthodes fondamentales** permettant de **déclencher/contrôler des opérations classiques de persistance (*recherches, mises à jour , suppressions , ...*)**.

```
package javax.persistence;

/*Interface used to interact with the persistence context. */
public interface EntityManager {

public void persist(Object entity); /* Make an instance managed and persistent.*/

public <T> T merge(T entity); /* Merge the state of the given entity
                                 into the current persistence context. */
public void remove(Object entity); /* Remove the entity instance. */

public <T> T find(Class<T> entityClass, Object primaryKey); /* Find by primary key. */
public <T> T getReference(Class<T> entityClass, Object primaryKey);
                                 /* Get an instance, whose state may be lazily fetched. */
public void flush(); /* Synchronize the persistence context to the underlying database.*/
public void setFlushMode(FlushModeType flushMode);
```

```
public FlushModeType getFlushMode();
public void lock(Object entity, LockModeType lockMode);
            /* Set the lock mode for an entity object contained in the persistence context.*/

public void refresh(Object entity); /* Refresh the state of the instance from the database */

public void clear(); /* Clear the persistence context */
public boolean contains(Object entity);
                    /* Check if the instance belongs to the current persistence context*/
public Query createQuery(String qlString);
                     /* Create an instance of Query for executing a JPQL statement */
public Query createNamedQuery(String name); /* in JPQL or native SQL */
public Query createNativeQuery(String sqlString);
public Query createNativeQuery(String sqlString, Class resultClass);
public Query createNativeQuery(String sqlString, String resultSetMapping);
public void joinTransaction(); /* join JTA active transaction */
public Object getDelegate(); /* implementation specific underlying provider */

public void close(); /*  Close an application-managed EntityManager */
public boolean isOpen();
public EntityTransaction getTransaction();
}
```

Un **contexte de persistance** (associé à *EntityManager*) peut:

- soit être **créé et géré par le conteneur** (ex: EJB , éventuellement WEB)
  puis injecté via l'annotation "**@PersistenceContext**" ou bien récupéré via un lookup JNDI .
- soit être explicitement créé via **createEntityManager**() de **EntityManagerFactory**().
  Son cycle de vie est alors à gérer (==> ***.close()*** à explicitement appeler)

2. Transaction JPA

L'interface "**EntityManager**" d'un **contexte de persistance** comporte une méthode ***getTransaction***() retournant de façon abstraite un objet permettant de contrôler (totalement ou partiellement) la transaction en cours:

```
public interface EntityTransaction {

public void begin(); /* Start a resource transaction. */
public void commit(); /* Commit the current transaction */
public void rollback(); /* Roll back the current transaction.*/

public void setRollbackOnly(); /* Mark the current transaction so that the only possible
outcome of the transaction is for the transaction to be rolled back.*/
public boolean getRollbackOnly();

public boolean isActive();
}
```

Remarque importante:

Ces transactions JPA peuvent en interne s'appuyer sur:

- ***des transactions simples et locales JDBC***
  ou bien
- ***des transactions complexes JTA (distribuées)***

Selon la configuration de ***META-INF/persistence.xml*** et la configuration du serveur d'application hôte (Jboss, WebSphere, Tomcat , ...).

D'autre part, en fonction du contexte d'intégration de JPA, les transactions seront explicitement ou implicitement gérées d'une des façons suivantes:

- **gestion déclarative** (et automatique/implicite) des **transactions** au sein des **EJB3**
- **gestion déclarative** (et automatique/implicite) des **transactions** au sein de **Spring**
- **gestion explicitement programmée des transactions** (sans framework d'intégration)

Exemple (avec gestion explicite):

```
try {
    entityManager.getTransaction().begin();
    Adresse nouvelleAdresse = new Adresse("rue elle","80000","Amiens");
    Client nouveauClient = new Client();
    nouveauClient.setNom("nom du nouveau client");
    nouveauClient.setAdressePrincipale(nouvelleAdresse);
    this.entityManager.persist(nouveauClient);
    num_cli = nouveauClient.getIdClient();
    System.out.println("id du nouveau client: " + num_cli);
    entityManager.getTransaction().commit();
    }
     catch (Exception e) {
       this.entityManager.getTransaction().rollback();
       e.printStackTrace();
     }
```

3. Différents états - objet potentiellement persistant

Etats objets	*Caractéristiques*
transient (proche état détaché)	Nouvel objet créé en mémoire, pris en charge par la JVM Java mais par encore contrôlé par une session Hibernate ou bien l'entityManager de JPA (le mapping objet/relationnel n'a jamais été activé). un tel objet n'a quelquefois par encore de clef primaire (elle sera souvent attribuée plus tard lors d'un appel à entityManager.persist() ou session.save())
persistant	objet actuellement sous le contrôle d'une session Hibernate ou de l'entityManager de JPA *(avant session.close() ou entityManager.close())*. Le mapping objet/relationnel est alors actif et une synchronisation (mémoire ==> base de données) est alors automatiquement déclenchée suite à une mise à jour (changement d'une valeur d'un attribut).
détaché	objet qui n'est plus sous le contrôle d'une session Hibernate ou de l'entityManager de JPA *(après session.close() ou entityManager.close())* . Les valeurs en mémoire sont conservées mais ne seront plus mises à jour (mapping objet/relationnel désactivé). Un objet détaché pourra éventuellement être ré-attaché à une session Hibernate (lors d'un update ou save_or_update() sur une session hibernate ou lors d'un appel à merge() sur l'entityManager de JPA) et ses éventuelles nouvelles valeurs pourront alors être synchronisées dans la base de données.

4. Cycle de vie d'un objet JPA/Hibernate

transient ==> **persistant** (==> ...)

(via **new**) (via ***session.save()*** ou ***entityManager.persist()***)

insert into ....

ou bien (après hibernation prolongée):

persistant ==> **détaché** (==> ***persistant***)

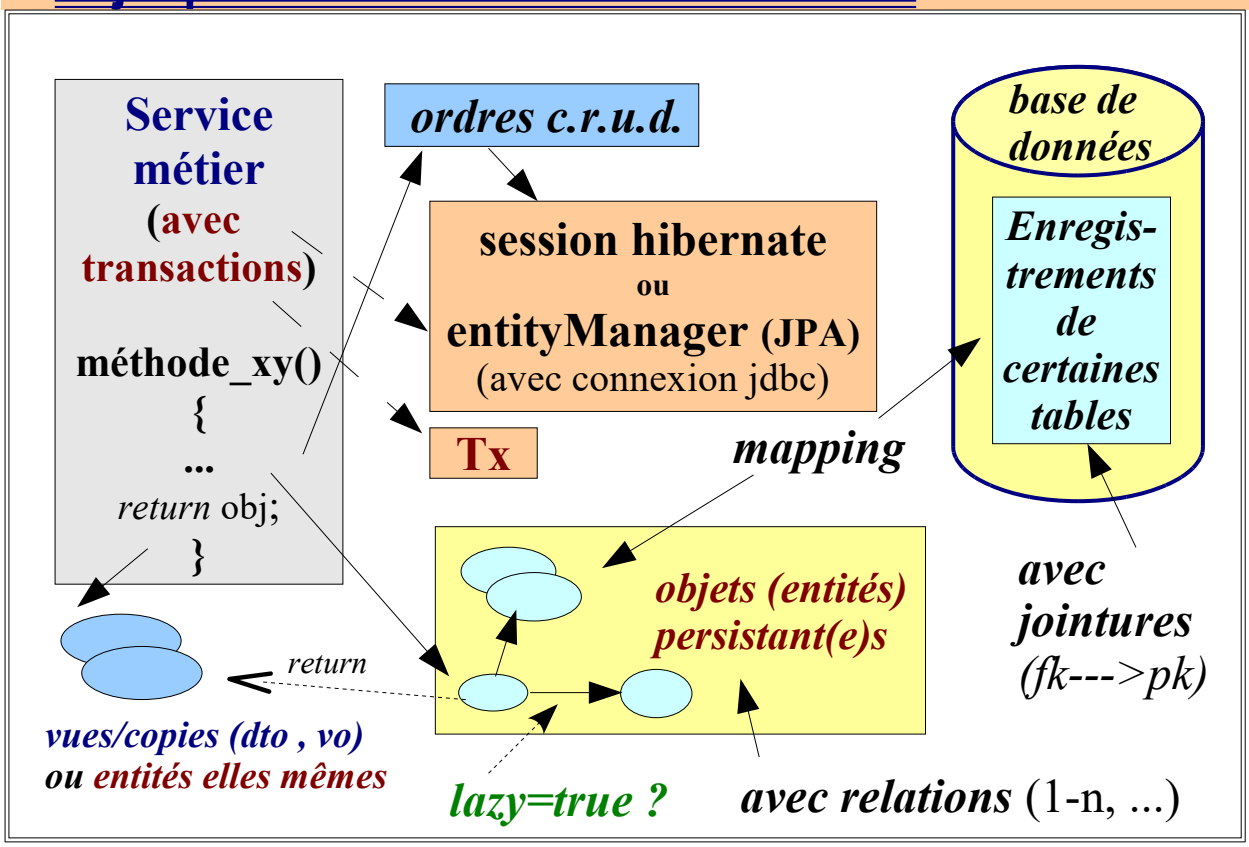
(via ***query***) (via session.close() ou entityManager.close()) (via ***session.update()*** , ... ou ***entityManager.merge()***)

select .... ***update .... set ...***

5. Synchronisation automatique dans l'état persistant

Synchronisation automatique dans l'état persistant (.update() ou .persist() non obligatoire) , mais .save() ou .merge() obligatoire pour un objet détaché.

6. objet persistant et architecture n-tiers



Remarques importantes:

- La technologie O.R.M. JPA ou hibernate sert essentiellement à remonter (et gérer) en mémoire un ensemble d'objets Java en tous points synchronisés avec les enregistrements d'une base de données relationnelle.
- Pour obtenir de bonnes performances (lazy=true) sans pour autant trop compliquer le code de l'application, il faut considérer les objets "entités persistantes" comme une structure d'ensemble orientée objet virtuellement liée à l'ensemble des données de la base (sachant que les mécanismes internes de JPA/Hibernate remontent les données en mémoire qu'en fonction des accès réellement effectués sur la structure objet [lazy=true]).
- La couche métier appelante (généralement développée avec Spring ou des EJB) doit souvent retourner des valeurs vers la couche présentation (IHM). Ces valeurs sont soit des références directes sur les entités persistantes elles mêmes ou bien des copies partielles sous formes de vues métiers (objets sérialisables / D.T.O. / V.O.) .
- Lorsque JPA/Hibernate est intégré dans Spring ou les EJB , les transactions sont automatiquement gérées par le conteneur et le code est alors significativement simplifié.

7. Principales méthodes JPA / EntityManager et Query

Principales méthodes de l'objet EntityManager:

méthodes	*traitements*
.**find**(class,pk)	Recherche en base et retourne un objet ayant les classe java et clef primaire indiquées
.**createQuery**(req_jpql)	Créer une requête JPQL et retourne cet objet
.**refresh**(obj)	Met à jour les valeurs d'un objet en mémoire en fonction de celles actuellement présentes dans la base de données (reload) .
.**persist**(obj_détaché) *ou bien assez inutilement* .***persist****(obj_déjà_persistant)*	Rend persistant un objet détaché : Sauvegarde les valeurs d'un nouvel objet dans la base de données (***insert into ...***) et retourne une exception de type `EntityExistsException` si l'entité existe déjà . La clef primaire est quelquefois automatiquement calculée lors de cette opération
.**merge**(obj_détaché) *ou bien assez inutilement* .***merge****(obj_déjà_dans_contexte_persistance)*	Sauvegarde ou bien met à jour (***update*** ...) les valeurs d'un objet dans la base de données. Cette méthode s'appelle .***merge***() car l'entité passé en argument peut quelquefois être utilisée pour remplacer les valeurs d'une ancienne version (avec la même clef primaire) déjà présente dans le contexte de persistance.
.**remove**(obj)	Supprime l'objet (delete ... from .... where .... dans la base de données)
.**flush**() déclenché indirectement via .***commit***()	Synchronise l'état de la base de données à partir des nouvelles valeurs des entités persistantes qui ont été modifiées en mémoire.

manager.**persist**() de **JPA** correspond à peu près au session.***save***() de ***Hibernate***

manager.**find**() de **JPA** correspond à peu près au session.***get***() de ***Hibernate***

manager.**remove**() de **JPA** correspond à peu près au session.***delete***() de ***Hibernate***

manager.**merge**() de **JPA** correspond à peu près au session.***save_or_update***() de ***Hibernate***

NB:

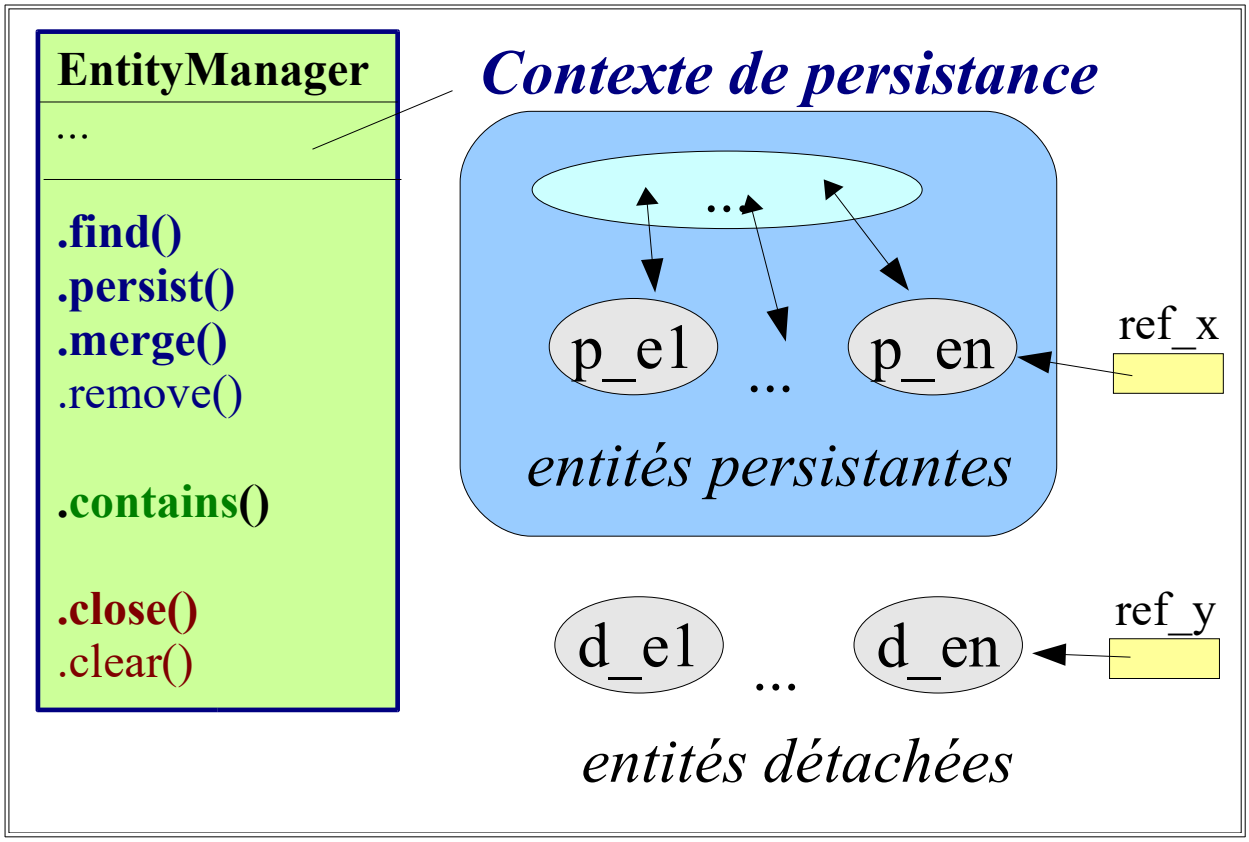
entityManager.getTransaction().**commit**() déclenche automatiquement *entityManager.****flush****()*

.**clear**() vide le contexte de persistance [tout passe à l'état détaché] sans effectuer de *.flush*()

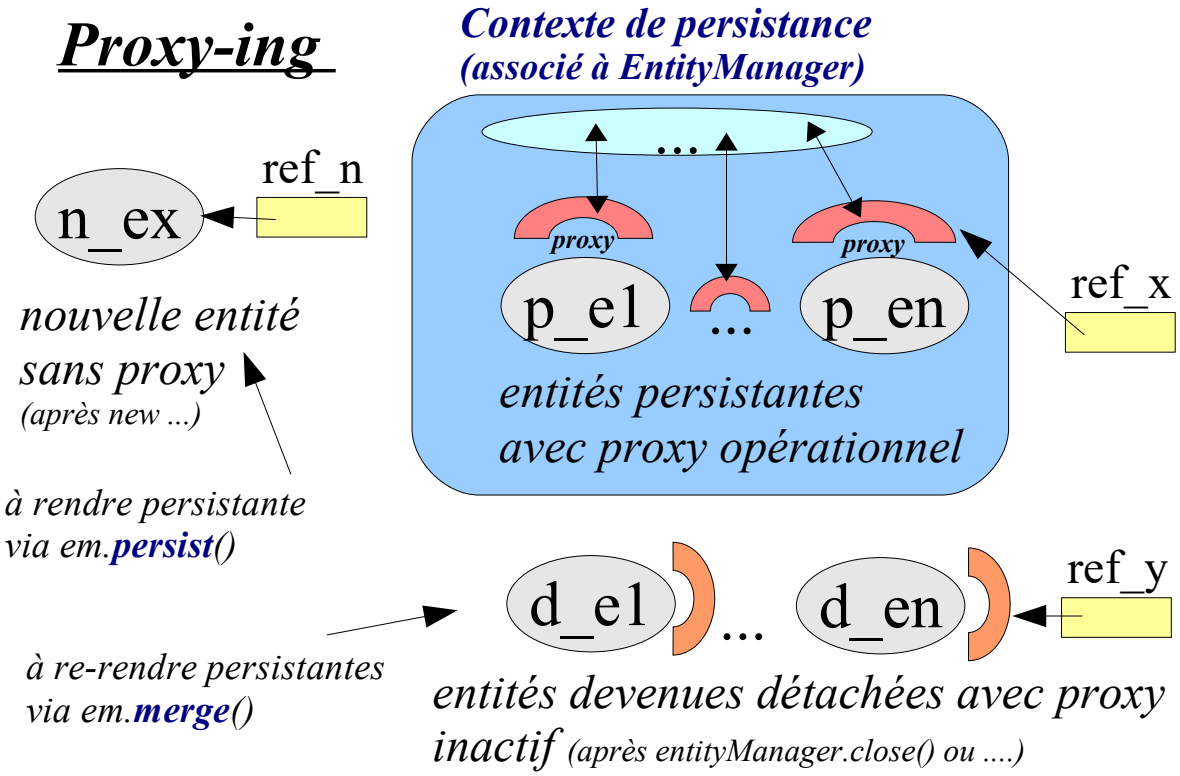
Principales méthodes de l'objet Query:

méthodes	*traitements*
.**setParameter**(paramName,paramValue)	met à jour un paramètre de la requête jpql
.**getSingleResult**()	récupère l'unique objet d'une requête simple.
.**getResultList**()	retourne sous forme d'objet "***java.util.List***" la liste des objets retournés en résultat de la requête JPQL.
.**executeUpdate**()	exécute une requête autre qu'un select
.***setMaxResults***()	fixe le nombre maxi d'éléments à récupérer

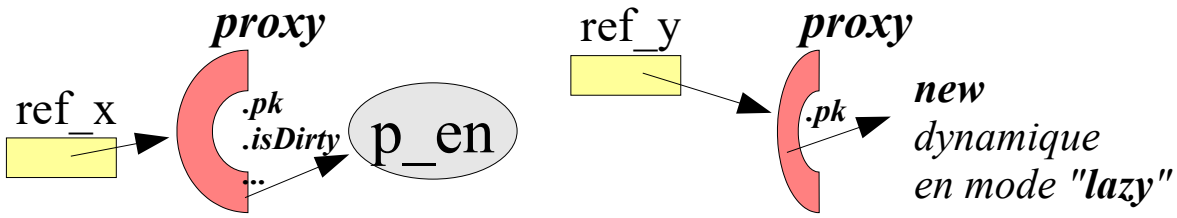
8. Contexte de persistance et proxy-ing



Proxy-ing



Proxy-ing et persistance



Le "**proxy**" *(généré par cglib ou autre)* sert entre autres à
- ***marquer les entités comme devant être mises à jour*** en base *(isDirty=true)* lors du prochain "***.flush()***"
- remonter les entités en mémoire de façon différée (***lazy***).

ref_persistant = entityManager.***merge***(ref_détaché);
ref_proxy_sans_entité = entityManager.***getReference***(class,pk);

V - Langage de requêtes JPQL

1.1. Présentation générale de HQL et JPQL

JPQL signifie ***JPA Query Language***
HQL ou **HbQL** signifie ***Hibernate Query Language***

Ces 2 langages correspondent à des ***minis langages dérivés de SQL*** et qui permettent d'**exprimer des requêtes** dans le cadre un **mapping objet/relationnel** géré par **un contexte de persistance JPA** ou par une **session Hibernate**.

1.2. Principaux éléments de syntaxe de JPQL et HQL

Le langage JPQL (ou HQL) n'est pas sensible à la casse pour sa partie SQL ("select" est équivalent à "SELECT") mais est sensible à la casse pour sa partie JAVA (la classe "Cat" n'est pas la même de "CAT" et la propriété c.weight n'est pas la même que c.WEIGHT).

Pour récupérer toutes les instances persistantes d'un certain type (c'est à dire tous les enregistrements de la table relationnelle associée) , une simple requête du genre "***from Cat***" *(ou bien "**from ppp.Cat**" en précisant optionnellement le package java)* est suffisante .

NB: La requête "***from Cat***" retourne non seulement des instances de la classe "***Cat***" mais également des instances des ***éventuelles sous classe***s (ex: "***DomesticCat***") . Le comportement de JPQL/HQL est bien en accord avec le "***polymorphisme***" du monde objet java.

De façon à pouvoir faire référence à une **instance** (du type précisé par from) dans une autre partie de la requête (ex: where) , on a généralement besoin d'un **alias** que l'on introduit par le mot clef optionnel "**as**" :

"from Cat **as** c" <==> "from Cat c"

"select c from Cat as c where c.weight > 8"

NB: la partie where est exprimée en utilisant la syntaxe ***nom_alias_instance.nom_propriété***
et non pas ~~*nom_alias_table.nom_colonne*~~ .

Autrement dit , la syntaxe de JPQL/HQL est centrée sur le monde objet (java) et pas sur le monde relationnel .

1.3. Quelques exemples de requêtes JPQL/HQL:

Rappel : **syntaxe** générale du **SELECT** du langage **SQL** :

SELECT fieldlist **FROM** tablenames [**WHERE** searchcondition]
[**GROUP BY** fieldlist [**HAVING** searchconditions]] [**ORDER BY** fieldlist]

Exemples JPQL/HQL:

```
SELECT cat.name FROM DomesticCat AS cat WHERE cat.name LIKE 'fri%'

SELECT avg(cat.weight), sum(cat.weight), max(cat.weight),
count(cat)  FROM Cat cat  ==> retourne une liste avec un seul
élément (ligne) de type Object[]


FROM Cat cat WHERE cat.mate.name is not null
```

```
SELECT c FROM Customer c JOIN c.orders o WHERE c.status = 1
(où c.orders est paramétré par @OneToMany ou @ManyToMany ...)
```

... **where** o.country **IN** ('UK', 'US', 'France')

... **where town LIKE** 'P%' où % signifie sous chaine quelconque de 0 à n caractères

... **where** p.age **BETWEEN** 15 and 19

```
SELECT DISTINCT auth FROM Author auth

    WHERE EXISTS

      (SELECT spouseAuth FROM Author spouseAuth WHERE spouseAuth = auth.spouse)


SELECT auth FROM Author auth WHERE auth.salary >= ALL(SELECT a.salary FROM Author a
WHERE a.magazine = auth.magazine)


SELECT mag FROM Magazine mag WHERE (SELECT COUNT(art) FROM mag.articles art) > 10

SELECT pub FROM Publisher pub JOIN pub.magazines mag ORDER BY o.revenue, o.name
```

1.4. Lancement d'une requête JPQL (JPA)

Exemples:

```
import javax.persistence.TypedQuery;
...
public List<Client> getAllClient()
        {
        TypedQuery<Client> query  =
              entityManager.createQuery("Select c from Client as c" , Client.class);
        return query.getResultList();
        }   ...
```

```
public   Categorie  getCategorieprincipale() {
                Categorie cat=null;
                Query q = entityManager.createQuery("select c from Categorie as c  "
                                                + "where  c.parentId  is  null");
                cat = (Categorie) q.getSingleResult();
                return cat;
        }

public List<Element> getElementByDesignation(String designation) {
                Query q = entityManager.createQuery("select e from Element as e "
                        +"where  e.designation=:design");
                q.setParameter("design", designation);          return q.getResultList();
        }
```

Remarque importante:

Les API "Hibernate" et "JPA 1" sont nées à l'époque de la transition entre les jdk <=1.4 et les jdk >= 1.5 (lorsqu'il n'existait pas encore de "generic" : List au lieu de List<T>) .

La méthode createQuery de l'interface EntityManager est surchargée.
Il existe une version qui renvoie un "TypedQuery" et qui accepte un second paramètre de type Class permettant de préciser le type de retour précis ce qui évite un warning .

D'autre part, les méthodes createQuery() et setParameter() ont été prévues pour s'enchaîner et l'on peut donc écrire :

```
return
 entityManager.createQuery("select c from Client as c where c.age >=:ageMini", Client.class)
                .setParameter("ageMini",18)
                .getResultList();
```

1.5. Lancement d'une requête native avec JPA

```
Query query = entityManager.createNativeQuery(
                    "SELECT NAME, SURNAME, AGE FROM PERSON");
List list = query.getResultList();
```

résultat ===>
une liste de tableaux (0bject[]) avec NAME à l'index 0 , SURNAME à l'index 1 et AGE à l'index 2

Via une version surchargée de createNativeQuery ayant un second paramètre on peut quelquefois préciser le type de retour souhaité (si compatible) .

1.6. NamedQuery placée dans une classe d'entité persistante

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

ou bien

```
@Entity
@NamedQueries({
   @NamedQuery(name="Country.findAll",
            query="SELECT c FROM Country c"),
   @NamedQuery(name="Country.findByName",
            query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
  ...
}
```

et appel via createNamedQuery("named_of_query" , ResultType.class) :

```
TypedQuery<Country> query =
    entityManager.createNamedQuery("Country.findAll", Country.class);

 List<Country> results = query.getResultList();
```

Intérêt de "NamedQuery" par rapport aux requêtes ordinaires ?

---> c'est à la fois une **affaire de style** et une **affaire de performance**.

- Dans le cas où un service métier n'utilise aucun DAO en arrière plan et utilise l'entityManager en direct, les "NamedQuery" ont l'avantage de bien séparer détails de persistance avec logique métier.
- Etant placées au dessus des classes "@Entity" chargées en mémoire et analysées dès le démarrage de l'application, les requêtes de type @NameQuery peuvent ainsi être **préparées** par le SGBDR et s'exécutent donc très vite.

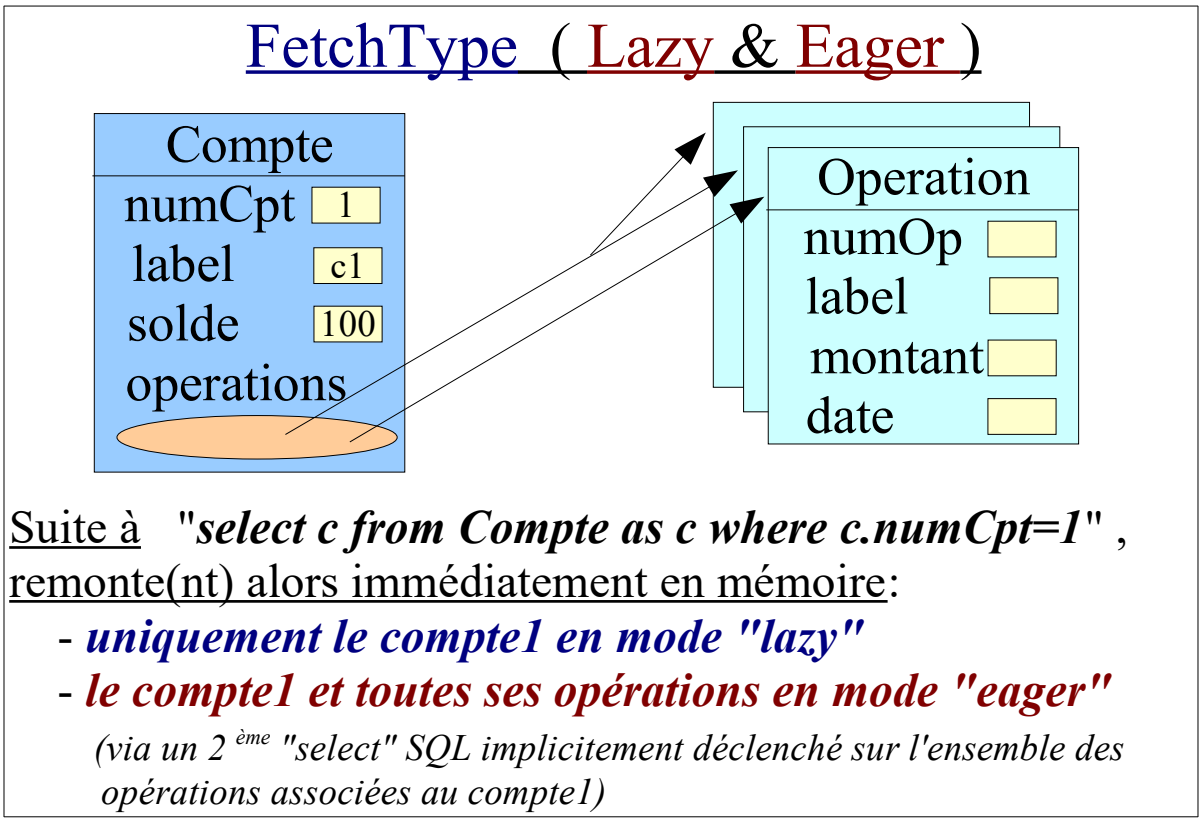
VI - O.R.M. JPA (généralités)

1. Vue d'ensemble sur les entités, valeurs et relations

1.1. Entités et objets valeurs (embedded)

- Une **entité** (@Entity) doit absolument comporter une **clef primaire** (@Id)
- Un sous objet imbriqué (@Embedded) n'a pas de clef primaire et doit absolument être référencé par une entité pour exister.

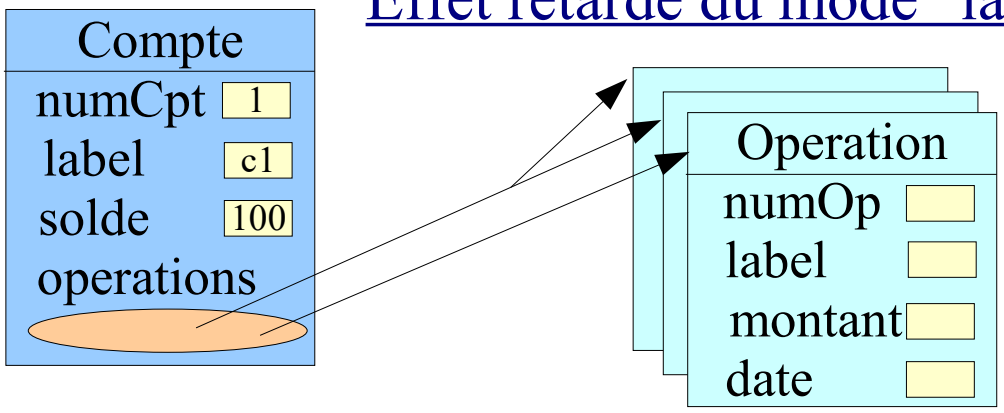
1.2. FecthType (lazy & eager)



L'attribut optionnel **lazy="true"** d'hibernate (ou bien **fetchType=Lazy** de JPA) permet de demander une ***initialisation tardive*** des membres d'une collection (les valeurs des éléments de la collection ne seront recherchées en base que lorsqu'elles seront consultées via par exemple un itérateur java). Ceci permet d'obtenir dans certains cas de meilleurs performances . Il faut cependant veiller à récupérer les valeurs de la collection avant le commit et la fermeture de la session (sinon ==> c'est trop tard : ***LazyInitialisationException***).

Il est fortement conseillé d'affecter explicitement la valeur "lazy" ou "eager" au paramètre fetchType (très important sur le plan des performances).

Effet retardé du mode "lazy"



- Si l'on accède qu'à "c1.getLabel() et c1.getSolde() les opérations ne sont alors jamais remontées en mémoire.
- Par contre, ***dès le parcours de la collection des opérations*** *(via un itérateur ou une boucle for)*, ***Les entités "Opération" sont alors automatiquement remontées en mémoire*** par JPA ***via une série de petits "select"*** SQL (jusqu'à "***1+N***" en tout").

Exemple JPA:

---> **fetch=FetchType.LAZY** or **fetch=FetchType.EAGER;**

```
@OneToMany( fetch=FetchType.LAZY , mappedBy="compte" )
private  List<Operation> dernieres_operations ;
```

NB : Si le mode "LAZY" est configuré dans une annotation JPA (ou dans un fichier hbm.xml), il est tout de même possible d'***expliciter ponctuellement une demande de chargement "tout d'un seul coup" via une requête JPQL*** du type "*select* ***distinct*** *c from Compte c* ***inner join fetch c.operations*** *inner join c.proprietaires cli where cli.numero = :numCli"*

1.3. Operations en cascade

Lorsqu'une opération (delete , updade , save, ...) est effectuée sur une entité persistante, elle est alors automatiquement répercutée sur les sous objets de type "valeurs/embbeded" .

Lorsque par contre on a affaire à une relation (1-1 ou 1-n) de type "entité-entité" , l'opération (save, update , delete) effectuée sur l'entité principale n'est pas automatiquement répercutée sur les entités liées. On peut cependant demander explicitement à JPA (ou hibernate) de répercuter une opération de mise à jour de l'entité principale vers une ou plusieurs entité(s) liée(s) via l'attribut ***cascade*** d'une relation .

Syntaxe JPA (dans les annotations):

```
@OneToMany(cascade=CascadeType.ALL , ....)
```

ou bien avec ensemble de cascades combinées:

```
cascade = {CascadeType.PERSIST, CascadeType.MERGE}, ...
```

NB: depuis JPA2 , la cascade "all" + "***delete-orphan***" permet en plus de supprimer automatiquement en base des entités "enfants" qui seraient détachées de leurs parents (suite un une

suppression de référence(s) dans une collection par exemple) .

NB: ceci fonctionne même si la mention "cascade" n'est pas précisée dans le schéma relationnel de la base de données.

2. Identité d'une entité (clef primaire)

2.1. Générateurs de clefs primaires

Une **clef primaine** (*ex*: numéro de facture) est assez souvent **générée automatiquement** via un **algorithme** basé sur la notion de **compteur**.

Exemple (JPA):

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="numero")
private Long num ;
```

NB :

- **@GeneratedValue(strategy=GenerationType.IDENTITY)** pour dire que la valeur de num (colonne numero) sera générée par **auto-incrémentation** par la base de donnée lors d'un insert into sql (déclenché via entityManager.persist())
- @GeneratedValue permet aussi de bien remonter en mémoire dans l'attribut num la valeur auto-incrémentée par la base
- La stratégie **IDENTITY** est une manière d'auto-incrémenter qui est compatible avec les versions récentes de MySql/MariaDB , Postgres , H2, Oracle , …

D'autres statégies existent (ex : ***SEQUENCE*** pour certaines bases telles que Oracle) .

2.2. Eventuelle clef primaire composée

Avec JPA:

Si une clef primaire est composée, il faut alors utiliser les annotations

@EmbeddedId (pour annoter la propriété lié au sous objet "clef primaire")

et

@Embeddable (pour annoter une c*lasse de clef primaire devant être sérialisable , publique , vu comme un "JavaBean" et devant coder equals() et hashcode()*).

Exemple de classe de clef primaire composée:

```
@Embeddable
public class CustomerPK implements java.io.Serializable
```

```
{
  private long id;
  private String name;

  public CustomerPK()  {  }

  public CustomerPK(long id, String name)  {
    this.id = id;     this.name = name;
  }

  public long getId()  {  return id;  }
  public void setId(long id)  {  this.id = id;  }

  public String getName()  {     return name;  }
  public void setName(String name)  {     this.name = name;  }

  public int hashCode()  {
    return (int) id + name.hashCode();
  }

  public boolean equals(Object obj)
  {
    if (obj == this) return true;
    if (!(obj instanceof CustomerPK)) return false;
    if (obj == null) return false;
    CustomerPK pk = (CustomerPK) obj;
    return pk.id == id && pk.name.equals(name);
  }
}
```

```
@Entity
public class Customer implements java.io.Serializable
{
  CustomerPK pk;
  ...
  public Customer(){ }

  @EmbeddedId
  public CustomerPK getPk() {
    return pk;
  }

  public void setPk(CustomerPK pk){
    this.pk = pk;
  }
  …  }
```

3. Propriétés d'une colonne

```
@Column(updatable = false, name = "flight_name", nullable = false, length = 50)
```

L'annotation @Column se transpose en sous balise <column> des fichiers de mapping hibernate

(.hbm.xml).

4. Relations (1-1, n-1, 1-n et n-n)

Annotations JPA	*Significations*
@OneToOne	1-1 (fk [unique] ---> pk) *[<many-to-one unique='true'> de hibernate]*
@OneToMany	1-n (collection avec clef [pk <----fk=clef])
@ManyToOne	n-1 (fk ----> pk)
@ManyToMany	n-n (fk1 ----> (k1,k2) ---> pk2)

NB: les annotations **@OneToOne , @OneToMany** et **@ManyToMany** peuvent éventuellement comporter un attribut "***mappedBy***" dont la valeur correspond à la ***propriété qui sert à établir une correspondance inverse (et secondaire)*** au sein d'une **relation bidirectionnelle** .

NB: **les mises à jour des relations effectuées uniquement du coté secondaire d'une relation bi-directionnelle ne seront pas automatiquement sauvegardées** (tant que le coté principal de la relation n'aura pas été explicitement réajusté).

L'annotation **@JoinColumn** permet d'indiquer *(via name="...")* le ***nom de la colonne correspondant à la clef étrangère*** dans la base de données.

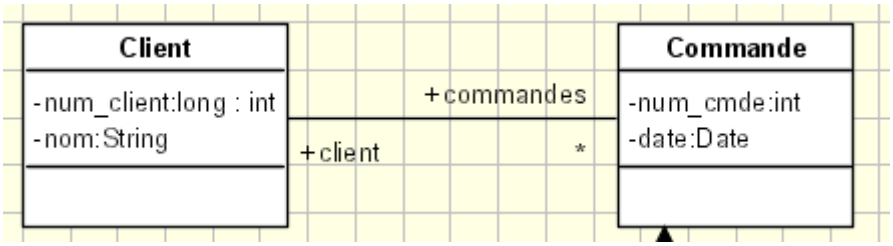
exemple:

```
@OneToOne(cascade = {CascadeType.ALL})
@JoinColumn(name = "ADDRESS_ID")
public Address getAddress()
{
   return address;
}

public void setAddress(Address address)
{
   this.address = address;
}
```

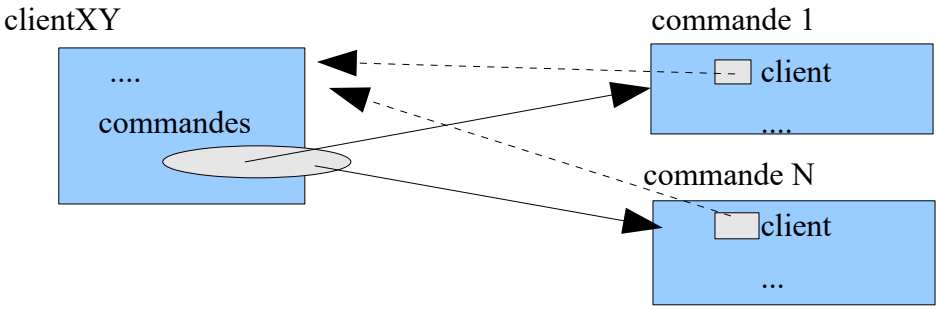
VII - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...)

1.1. Relations 1-n et n-1 (entité-entités)

UML:



Java:



Base de données:

T_Client
client_id (pk) , nom , ...

one-to-many *many-to-one* **T_Commande**
num_cmde (pk) , **id_client (fk)**, date, ...

Annotations JPA:

```
@Entity
@Table(name = "T_Commande")
public class Commande
{
 @ManyToOne
  @JoinColumn(name = "id_client") //fk (nom de colonne dans la table)
private Client client;
...
  public Client getClient()  {     return client;  }
  public void setClient(Client  client)  {     this.client = client;  }
}
```

```
@Entity
@Table(name = "T_Client")
public class Client
{
 //NB : "client" (valeur de mappedBy) = nom (java) de la relation inverse
  @OneToMany(fetch=FetchType.LAZY,mappedBy="client")
 private List<Commande> commandes;
  ...
  public List<Commande> getCommandes()  {  return commandes;  }
  public void setCommandes(List<Commande> cmdes) {  this.commandes = cmdes;  }
```

```
  public void addCommande(Commande c) {
    if (commandes == null) commandes = new ArrayList<Commande>();
    commandes.add(c);   }
}
```

Rappels importants (pour JPA):

- La valeur de l'attribut "***mappedBy***" de l'annotation **@OneToMany** correspond à la ***propriété (dans l'autre classe java) qui sert à établir une correspondance inverse*** au sein d'une **relation bidirectionnelle** .
- L'annotation **@JoinColumn** permet d'indiquer (via name="...") le nom de la colonne correspondant à la clef étrangère dans la base de données.

1.2. Relations 1-n (avec JoinTable) évolutives vers du n-n

Bien qu'une relation 1-n (symétriquement n-1) soit souvent basée sur une jointure simple avec une clef étrangère en base, il est tout de même possible de mettre en place une relation 1-n avec une table de jointure en base . Déjà structurée en base comme une relation n-n , une telle relation 1-n a le mérite d'être facilement évolutive vers du n-n.

Exemple :

```
@Entity
@Table(name="Compte")
public class Compte {
      ...
      @ManyToOne //evolutif vers @ManyToMany
      @JoinTable(name="ClientCompte",
                  joinColumns={@JoinColumn(name="num_compte")},
                  inverseJoinColumns={@JoinColumn(name="num_client")})
      private Client client;
}
```

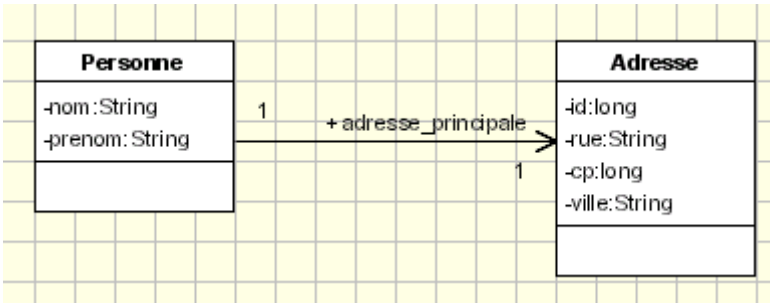
```
@Entity
@Table(name="Client")
public class Client extends Personne {
      ...
      @OneToMany(mappedBy="client")
      private List<Compte> comptes;
      …
}
```

avec

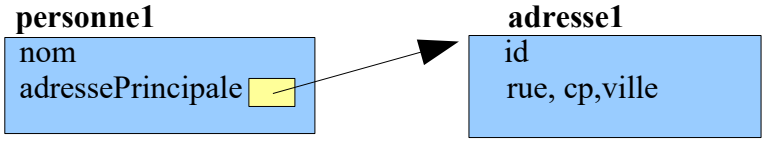
```
create table ClientCompte(   num_client integer,   num_compte integer,
                                primary key(num_client,num_compte));
ALTER TABLE ClientCompte ADD CONSTRAINT client_avec_compte_valide
      FOREIGN KEY  (num_compte) REFERENCES Compte(numero);
ALTER TABLE ClientCompte ADD CONSTRAINT compte_avec_client_valide
      FOREIGN KEY (num_client) REFERENCES Client(num_client);
```

1.3. Relation 1-1 via clef étrangère (évolutif vers n-1)

UML:

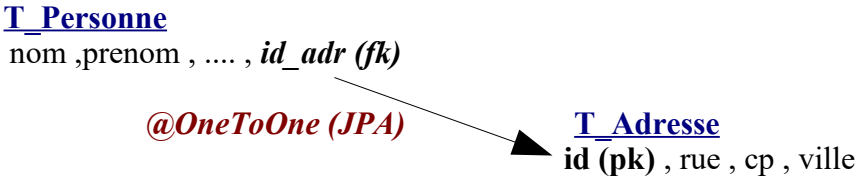


Java:



===> exemple: ***personne1.getAdressePrincipale().getVille()***;

Base de données:



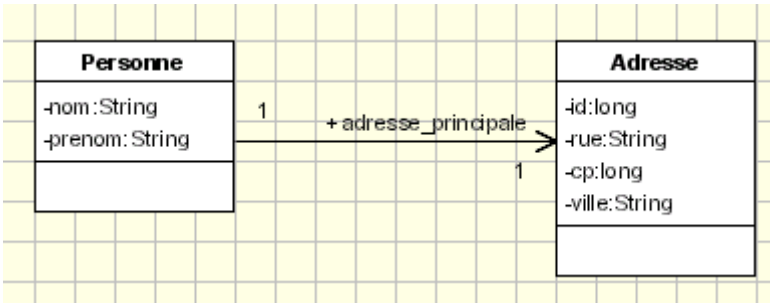
1-1 ici évolutif vers n-1 (exactement structuré et paramétré comme n-1 et @ManyToOne) .
Lorsque c'est possible une contrainte **unique=true** sur la colonne id_adr (fk) peut indiquer une intention de 1-1 plutôt que n-1 .

JPA:

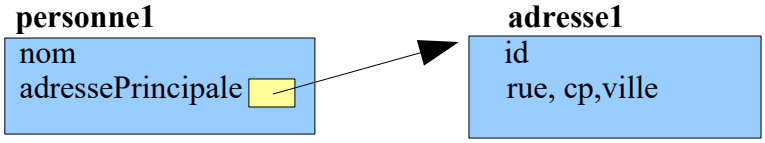
```
@Entity
public class Personne
{
  @OneToOne()
  @JoinColumn(name = "id_adr" , unique=true) //fk

  private  Adresse   adressePrincipale;
  ...
  public  Adresse  getAdressePrincipale()  {
     return adressePrincipale;
  }
  public void setAdressePrincipale(Adresse adressePrincipale) {
     this. adressePrincipale = adressePrincipale;
  }
}
```

1.4. Relation 1-1 via Table secondaire et correspondance pk

UML:



Java:



===> exemple: ***personne1.getAdressePrincipale().getVille()***;

Base de données:

T_Personne

id (pk) , nom ,prenom , ...

@OneToOne (JPA)

T_Adresse_De_Personne

idPers (pk) , rue , cp , ville

Les valeurs de idPers (clef primaire) de la table secondaire "T_Adresse_De_Personne" doivent absolument correspondre à celle de "id" (clef primaire) de la table principale "T_Personne". Cette correspondance rigide entre les clefs primaires des tables principale et secondaire n'est pas évolutif vers du n-1 . c'est du 1-1 pour toujours .

JPA:

```
@Entity
@Table(name="T_Personne")
public class Personne
{
 @Id
 @GeneratedValue(strategy=....IDENTITY)
 private Long id ;

  @OneToOne(optional=true ,
                  mappedBy="personne")
 private   AdresseDePersonne
                  adressePrincipale;
  ...
}
```

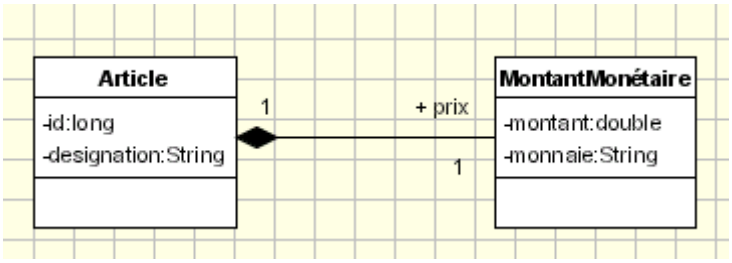
```
@Entity
@Table(name="T_Adresse_De_Personne")
public class AdresseDePersonne
{
 @Id
 private Long idPers;
  private String rue ;
  private String cp;
  private String ville;

  @OneToOne(optional=false)
   @PrimaryKeyJoinColumn
  private   Personne   personne;
}
```

NB : Dans le cas rare où idPers de AdresseDePersonne ne serait pas explicitement définie comme clef primaire / id , l'annotation @PrimaryKeyJoinColumn comporte un attribut facultatif "referencedColumnName" .

1.5. Sous objets incorporable au sein d'une entité

Appelés objets "*Valeur composite*" ou "*Component*" dans la terminologie *Hibernate* , et appelés objets "***Embeddable***" dans la terminologie ***JPA***, d'**éventuels sous objets composés(non partageables et sans clef primaire) peuvent être utilisés pour structurer une entité**.

UML:



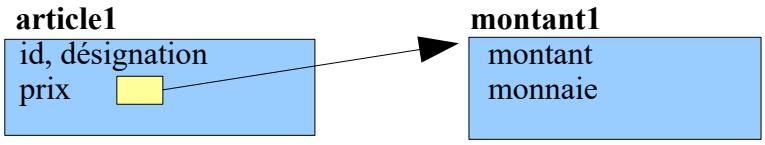
En base de données:

T_Article
id (pk) , designation ,***montant , monnaie*** , ...

==> mapping relationnel associé (par défaut):

- pas de table relationnelle séparée pour les sous objets "valeurs"
- liste des propriétés du composant directement incluse dans la table de l'objet contenant .

En Java:



==> xxx.***getPrix().getMontant()***; et xxx.***getPrix().getMonnaie()***;

NB: La classe java du composant imbriqué n'a pas d' ID (pas de clef primaire) . Elle respecte néanmoins les conventions "**JavaBean**" pour les éléments internes (ici montant et monnaie) .

JPA:

Une classe d'objet incorporable doit normalement être annotée via "**@Embeddable**" .

Une propriété d'une entité qui référence un objet imbriqué doit être annotée via **@Embedded** .

Les spécifications JPA se limitent à un seul niveau d'imbrication.

exemple:

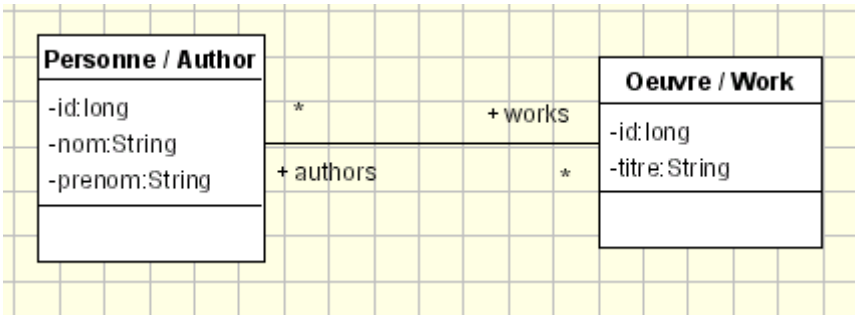
```
@Embeddable
public class MontantMonetaire
{
  private double montant;  // + get/set
  private String monnaie;  // + get/set
  public MontantMonetaire()   { }

  public MontantMonetaire(double montant, String monnaie)   {
    this.montant = montant;      this.monnaie = monnaie;
  }
}
```

```
@Entity
@Table(name = "T_Article")
public class Article
  private long id;  // avec get/set et @Id

  @Embedded
 /*  @AttributeOverrides({
    @AttributeOverride(name = "montant", column = @Column(name = "montant")),
    @AttributeOverride(name = "monnaie", column = @Column(name = "monnaie"))
  }) */
  private MontantMonetaire   prix;
  ...
  public MontantMonetaire getPrix()   {   return prix;   }
  public void setPrix( MontantMonetaire prix)   {    this.prix = prix;   }
}
```

NB : @AttributeOverride **n'est réellement utile que si les noms des colonnes sont différentes des noms des propriétés java de l'objet incorporé .**

1.6. Relation n-n ordinaire

UML:



Mapping relationnel ==> ***Table intermédiaire***

T_Author
auth_id (pk) , nom ,prenom , ...

Author_Work
author_id (fk) , work_id(fk)

many-to-many

T_Work
w_id (pk) , titre , ...

Java:
--> des collections dans les 2 sens
avec un sens principal et un sens secondaire (avec mappedBy).

JPA:

du coté "principal" de la relation (coté où les mises à jour seront "persistées"):

```
@Entity
public class Work
{
@ManyToMany( fetch = FetchType.LAZY)
  @JoinTable(name = "Author_Work",
             joinColumns = {@JoinColumn(name = "work_id")},
             inverseJoinColumns = {@JoinColumn(name = "author_id")})
private List<Author> authors;
...
  public List<Author> getAuthors()  {
    return authors;
  }
  public void setAuthors(List<Author> authors)  {    this. authors = authors;  }
..
}
```

NB: l'attribut ***joinColumns*** (de @JoinTable) correspond à la ***clef étrangère pointant vers l'entité courante*** et l'attribut **inverseJoinColumns** correspond à la **clef étrangère pointant vers les éléments de l'autre coté de la relation (ceux qui seront rangés dans la collection paramétrée)**.

et du coté inverse/secondaire:

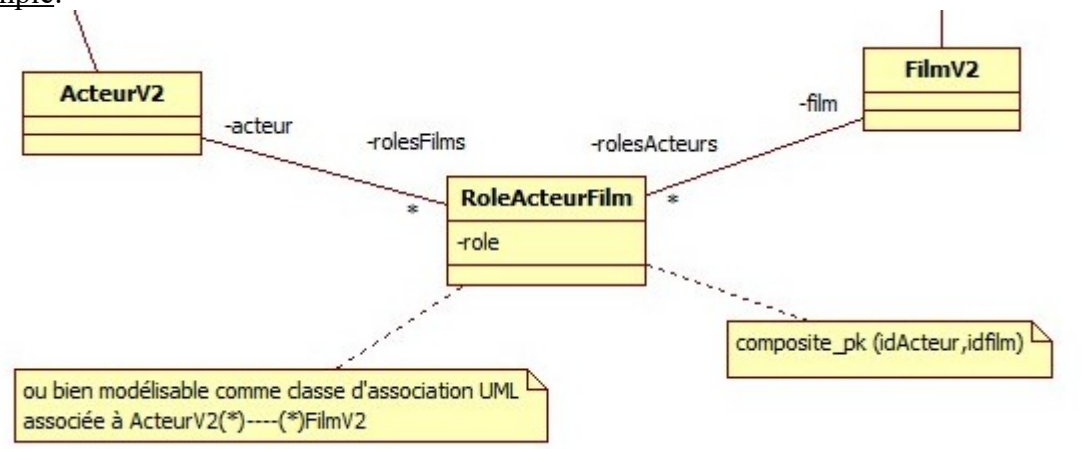
```
@Entity
public class Author
{
 @ManyToMany(mappedBy="authors")
  private List<Work> works;
...
   public List<Work> getWorks()   {
     return works;
   }
   public void setWorks(List<Work> works)   {      this.works = works;   }
}
```

1.7. Relation n-n avec détails d'association

Remarque importante sur les relations (n-n):

Lorsque la table de jointure comporte des informations supplémentaires autres que les clefs étrangères, on est alors obligé de décomposer la relation n-n en deux relations (1-n + n-1) et les éléments de la table de jointure sont alors vus comme des entités ayant une clef primaire (souvent composée par les 2 clefs étrangères).

Exemple:




```
CREATE TABLE ActeurFilmV2
     ( idActeur integer,
       idFilm integer ,
       role VARCHAR(64),
       primary key(idActeur,idFilm));

ALTER TABLE ActeurFilmV2 ADD CONSTRAINT avec_acteur_valideV2
           FOREIGN KEY  (idActeur) REFERENCES Acteur(idActeur);
ALTER TABLE ActeurFilmV2 ADD CONSTRAINT avec_film_valideV2
            FOREIGN KEY  (idFilm) REFERENCES Film(idFilm);
```

```
@Entity
```

```
@Table(name="Acteur")
public class ActeurV2  {

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

	private String nom ;

	@OneToMany(mappedBy="acteur")
	private List<RoleActeurFilm> rolesFilms;

	...
}
```

```
@Entity
@Table(name="ActeurFilmV2")
public class RoleActeurFilm {
	@EmbeddedId
	private RoleActeurFilmCompositePk pk ;

	private String role;

	@ManyToOne
	@JoinColumn(name="idActeur")
	@MapsId("idActeur") //pk.idActeur
	private ActeurV2 acteur;

	@ManyToOne
	@JoinColumn(name="idFilm")
	@MapsId("idFilm") //pk.idFilm
	private FilmV2 film;

	public RoleActeurFilm() {
		super();   pk =  new RoleActeurFilmCompositePk();
	}

	public RoleActeurFilm(String role, ActeurV2 acteur, FilmV2 film) {
		super();
		this.role = role; this.acteur = acteur; this.film = film;
		pk=new RoleActeurFilmCompositePk(acteur.getIdActeur(),
						film.getIdFilm());
	}
	//+get/set , ...
}
```

NB: Avec **JPA 1.0** , @MapsId() n'existait pas et **@JoinColumn**(name="id_...", **insertable=false, updatable=false**) était indispensable pour que JPA puisse gérer un double mapping où les colonnes "idFilm" et "idActeur" sont déjà précisées au sein de la clef primaire (composite pk).

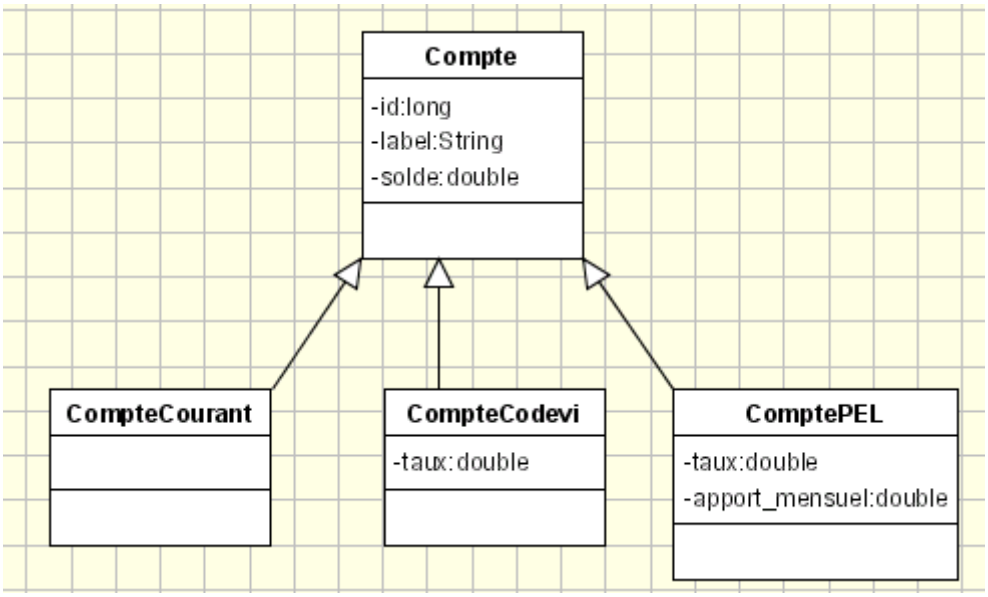
Depuis **JPA 2**, l'annotation **MapsId("id...")** permet de préciser la sous partie de la classe de clef primaire composite java qui sera associée à la clef étrangère (du @ManyToOne / @JoinColumn) .
Un futur appel à roleActeurFilm.***setActeur***(acteur) déclenchera alors automatiquement une mise à jour de ***pk.idActeur*** .

```
@Embeddable
public class RoleActeurFilmCompositePk implements Serializable {
	private static final long serialVersionUID = 1L;

	private long idActeur;
	private long idFilm;

	public RoleActeurFilmCompositePk() {
		super();
	}

	public RoleActeurFilmCompositePk(long idActeur, long idFilm) {
		super();
		this.idActeur = idActeur;
		this.idFilm = idFilm;
	}

	//+get/set , equals et hashcode (generated by eclipse)
}
```

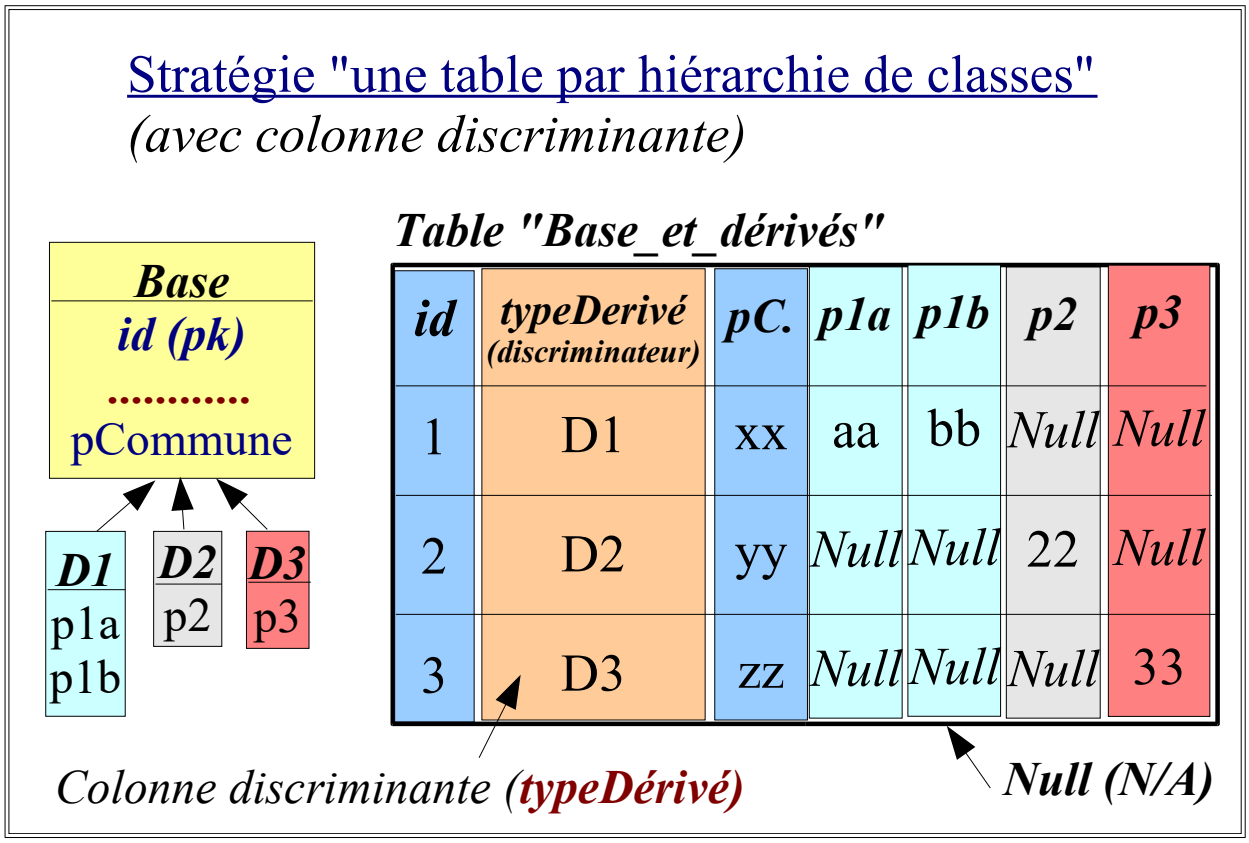
```
@Entity
@Table(name="Film")
public class FilmV2 {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private long idFilm;
	private String producteur;
	private String titre;

	@Column(name="dateSortie")
	@Temporal(DATE) //partie significative = DATE et pas DateTime
	public Date date;


	@OneToMany(mappedBy="film")
	private List<RoleActeurFilm> rolesActeurs ;

	...

}
```

2. Relations d'héritage & polymorphisme

UML:



2.1. stratégie "une seule table par hiérarchie de classes"



Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.

Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

Cette stratégie (relativement simple) est assez pratique et adaptée dans le cas où il y a peu de différences structurelles ente les sous classes .

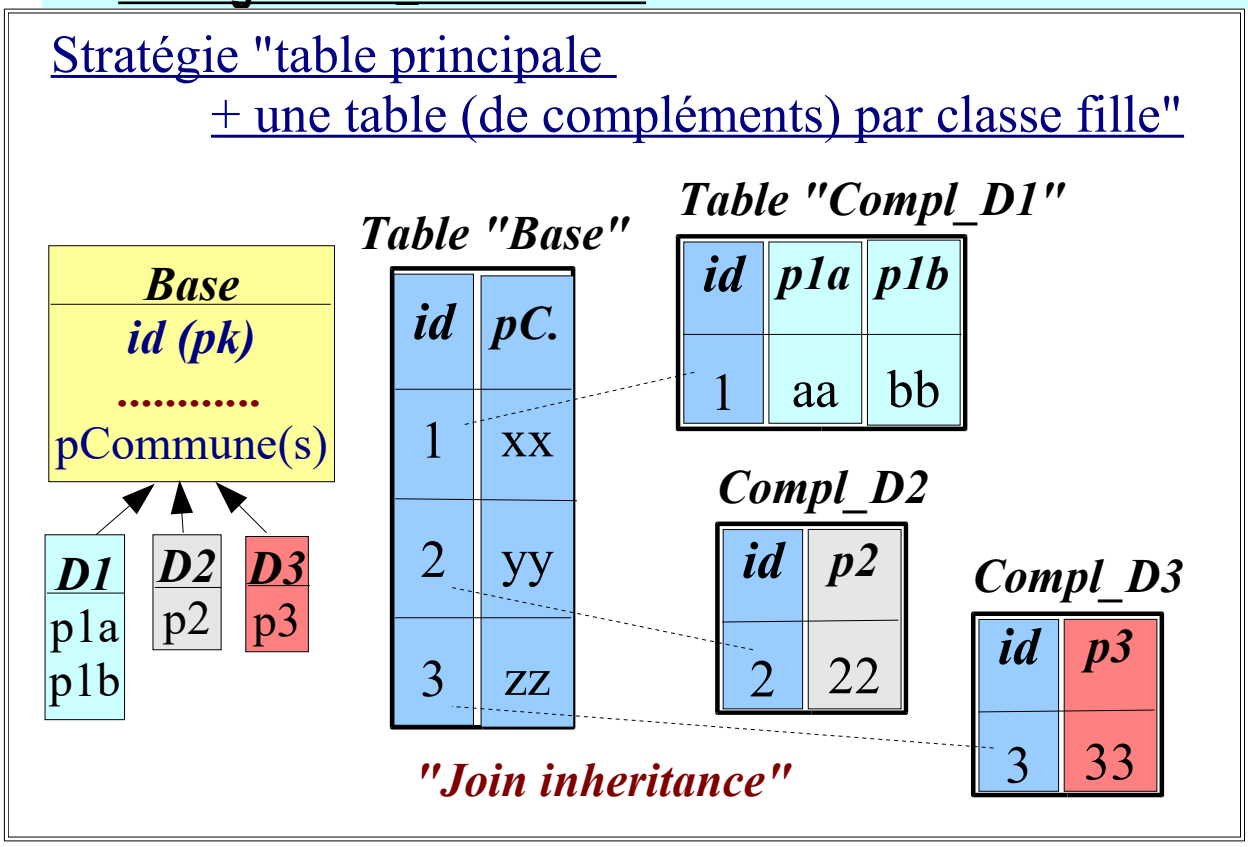
Contrainte : les colonnes liées aux attributs des sous classes ne peuvent pas avoir la contrainte "NOT NULL" . (***Pour une instance de la sous classe D1 , les colonnes inutilisées de la sous classe D2 doivent avoir la valeur NULL)***.

Pour la classe de base (JPA):

```
@Entity
@Table(name="Compte")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "typeCompte",
                     discriminatorType = DiscriminatorType.STRING)
/* @DiscriminatorValue("CompteCourant") //éventuellement en tant que valeur par défaut
   // et si pas de sous classe CompteCourant mais utilisation directe du parent "Compte" */
public class Compte
{
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private long id;
  private String label;
  private double solde;

  public int getId()   { return id;  }
  public void setId(int id)   { this.id = id;   }

  public String getLabel()  {  return label;  }
  public void setLabel(String label)  { this.label = label;  }

  public double getSolde() { return solde;}
  public void setSolde(double solde) {this.solde = solde;  }
}
```

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("PEL")
public class ComptePEL extends Compte
{
  private double tauxInteret;    //+get/set
  private double apportMensuel;  //+get/set
  ...
}
```

2.2. Stratégie "Join_inheritance"




```
@Entity
@Table(name="Compte")
@Inheritance(strategy=InheritanceType.JOINED )
public class Compte
{
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private long id;
  private String label; // +get/set
  private double solde; // +get/set

  public int getId()   { return id;  }
  public void setId(int id)   { this.id = id;   }
...
}
```

NB : Bien que "pas indispensable" , une éventuelle colonne discriminante pourrait éventuellement être ajoutée si l'implémentation de JPA en tient compte du point de vue des performances.

```
@Entity
@Table(name="ComptePEL")
@Inheritance(strategy=InheritanceType.JOINED )
public class ComptePEL extends Compte
{
  private double tauxInteret;    //+get/set
  private double apportMensuel;  //+get/set
  ...
}
```

2.3. Stratégie (assez rare) "une table par classe"

Avec tout un tas de restrictions (voir documentation de référence si besoin)

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Compte
{
   @Id
   @GeneratedValue(strategy=GenerationType.TABLE)
 ...
  public int getId()   {    return id;  }

   public void setId(int id)  {  this.id = id;  }

   ...
}
```

```
@Entity
public class ComptePEL extends Compte
{
   private double tauxInteret; // +get/set
   ...
}
```

2.4. Polymorphisme

Une requête JPQL/HQL exprimée avec un type correspondant à une classe parente retournera (par défaut / sans restriction explicite) **toutes les instances de toute une hiérarchie de classes (classe mère + classe fille)**. Le type exact de chacune des instances retournées sera précis (bien que compatible avec le type de la sur-classe) ==> **polymorphisme complet** (java/mémoire + au niveau O.R.M.).

Autrement dit :

select c from Compte as c" retournera tous les types de Compte (courant , codevi , pel).

"select c from ComptePEL as c" ne retournera que des comptes de types "PEL" .

VIII - JPA : aspects divers et avancés

1. Quelques spécificités de JPA

1.1. Une entité répartie dans 2 tables (principale,secondaire)

NB: avec JPA, il est également possible d'associer quelques propriétés d'une entité à une table secondaire :

```
@Entity
@Table(name = "CUSTOMER")
@SecondaryTable(name = "EMBEDDED_ADDRESS")
public class Customer implements java.io.Serializable
{
  @Column(name = "STREET", table = "EMBEDDED_ADDRESS")
  private String street ;

  @Column(name = "CITY", table = "EMBEDDED_ADDRESS")
  private String city ;
 ...
  public String getStreet()   { return street;  }
  public void setStreet(String street)   {   this.street = street;  }

  public String getCity()   {      return city;   }
  public void setCity(String city)   {      this.city = city;   }

}
```

==> dans ce cas (clef primaire table_secondaire = clef_primaire_table_principale)
[cas du <one-to-one> des .hbm.xml de hibernate]

1.2. Eléments requis sur une classe d'entité (JPA):

- vraie classe (pas interface ni enum) non finale (pas de mot clef "final" , on doit pouvoir en hériter).
- un constructeur par défaut (sans argument) obligatoire et devant être public ou protégé (d'autres constructeurs sont possibles mais non indispensables).
- annotation "@Entity" ou bien <entity ...> dans un descripteur Xml.
- éventuelle implémentation de *java.io.Serializable* si besoin d'un transfert distant à l'état détaché.

NB:

- Des héritages sont possibles (entre "entité" et "non entité" et vice versa) et le polymorphisme peut s'appliquer sans problème.
- L'état d'une entité est liée à l'ensemble des valeurs de ses attributs (devant être privés ou protégés) .

- L'accès aux propriétés d'une entité doit s'effectuer via des "getter/setter" (conventions JavaBean).

La persistance (mapping ORM) s'applique sur toutes les propriétés non "transientes" d'une entité.

Sachant qu'une éventuelle propriété non persistante doit être marquée via "private transient ...." et par l'annotation "@Transient" .

1.3. Verrous (optimistes et pessimistes)

```
@Version
   @Column(name = "OPTLOCK")
   public Integer getVersion()
   {
      return version;
   }

   public void setVersion(Integer i)
   {
      version = i;
   }
```

2. Cycle de persistance (pour @Entity) et annotations/callbacks associées pour "Listener"

Annotations (callback)	*déclenchement (étape du cycle de persistance)*
@PrePersist	juste avant passage à l'état persistant
@PostPersist	juste après passage à l'état persistant (clef primaire souvent affectée)
@PreRemove	juste avant suppression
@PostRemove	juste après suppression et destruction
@PreUpdate	juste avant mise à jour
@PostUpdate	juste après mise à jour
@PostLoad	juste après chargement en mémoire (suite à une recherche)

```
@Entity
@Table(name = "CUSTOMER")
@EntityListeners(CustomerCallbackListener.class)
public class Customer implements java.io.Serializable
{
...
}
```

```
public class CustomerCallbackListener
{
   @PrePersist
   public void doPrePersist(Customer customer) {
      System.out.println("doPrePersist: About to create Customer: " + ...);
   }

   @PostPersist public void doPostPersist(Customer  customer) {
      System.out.println("doPostPersist: Created Customer: " + ...);
   }

   @PreRemove public void doPreRemove(Customer customer) {
      System.out.println("doPreRemove: About to delete Customer: " + ...);
   }

   @PostRemove public void doPostRemove(Customer customer)  {
      System.out.println("doPostRemove: Deleted Customer: " +...);
   }

   @PreUpdate public void doPreUpdate(Customer customer) {
      System.out.println("doPreUpdate: About to update Customer: " +...);
   }

   @PostUpdate public void doPostUpdate(Customer customer) {
      System.out.println("doPostUpdate: Updated Customer: " +...);
   }

   @PostLoad public void doPostLoad(Customer customer) {
      System.out.println("doPostLoad: Loaded Customer: " + ...);
   }
}
```

3. Map et apports de JPA 2.0

3.1. Détails sur persistance d'une énumération

@Enumerated(*EnumType.STRING*)
ou bien
@Enumerated(*EnumType.ORDINAL*) *//by default*

à placer au dessus d'une propriété dont le type est une énumération java.

3.2. Tables d'associations (map)

Des relations 1-n (et n-n) peuvent soit être matérialisées comme des collections , soit être codées comme des tables d'associations ("map") .

Une table d'associations ("map") nécessite absolument une clef (à préciser) .

La propriété d'un objet de la "Map" qui sera associée à cette clef peut être renseignée via l'annotation **@MapKey**(name="...") .

Si l'indication "@MapKey" n'est pas présente , la clef primaire (id) est alors utilisée comme clef par défaut.

JPA 1 ne supportait que des "map" de sous objets "Entity" .

JPA 2 est maintenant capable de gérer en plus des "map" d'éléments simples ("basic").

Etant donné qu'un élément simple n'a pas obligatoirement d'identifiant (pk) , la clef peut être renseignée par l'annotation @MapKeyColumn .

Exemple (JPA 1 ou 2):

```
...
import java.util.Map;
import javax.persistence.*;

@Entity
@Table(name="Qcm")
public class Qcm {

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

	private String titre;

	@OneToMany(mappedBy="qcm")
	@MapKey(name="numQuestion")
	private Map<Long,Question> questions;

	//+get/set
}
```

3.3. Collection d'objets "embedded" (sans id):

Depuis la version 2 de JPA, il est possible de définir (via ***@ElementCollection*** et ***@CollectionTable***) des collections de sous objets imbriqués (sans id) . Cependant ceci ne fonctionne qu'à un seul niveau (pas de sous-sous objets (sans id) dans des sous-objets (sans id)).

En UML, ceci se modélise généralement comme une composition (agrégation forte / losange noir) avec une multiplicité "*" ou "1..*" .

La sémantique associée induit des cascades automatiques !!!

Exemple:

Qcm avec Entités "Question" ayant les caractéristiques suivantes:

- clef primaire composé de (idQcm , numQuestion)
- collection de sous-objets "Réponse" imbriqués (sans id)

Les Questions sont stockés dans une table Question.

Les sous objets réponses seront stockés dans une table "Reponse" ayant une clef primaire composée (idQcm, numQuestion, numReponse) . Cette composition est basée sur

- une sous partie (idQcm , numQuestion) correspondante à la clef primaire le l'entité parente
  Cette sous partie "fk" de liaison avec le parent sera précisée via l'attribut "***joinColumns***"
  de l'annotation ***@CollectionTable***
- une sous partie complémentaire quelconque (numReponse ou numLigne ou index ....)

```
@Embeddable
public class QuestionPK implements Serializable{

        @Column(name="idOfQcm")
        private long idQcm;

        @Column(name="numQuestion")
        private int num;
    ...
}
```

```
@Embeddable
public class Reponse {
    // NB: dans cette classe , on ne fait pas apparaître les propriétés (idQcm,numQuestion) pourtant
    // présentes dans la table "Reponse" , les mécanismes JPA s'en chargeront automatiquement !!!
        private char numReponse;
        private String texte;
        private boolean ok;
        // + get/set
}
```

```
@Entity
public class Question /* de qcm */ {
        @Id
        private QuestionPK pk;

        @Column(insertable=false,updatable=false)  // pour éviter conflit avec partie "num" ok pk
        private int numQuestion;

        @ManyToOne
        @JoinColumn(name="idOfQcm",insertable=false,updatable=false)
        private Qcm qcm;

        private String texte;

        @ElementCollection()
        @CollectionTable(name="Reponse", joinColumns={
                @JoinColumn(name="idOfQcm"),
                @JoinColumn(name="numOfQuestion")})
        private List<Reponse> reponses;
        // + get/set
}
```

Structure de la base de données:

```
CREATE TABLE Qcm(idQcm integer  auto_increment  PRIMARY KEY, titre VARCHAR(128));
INSERT INTO Qcm Values (1,'qcm1');    INSERT INTO Qcm Values (2,'qcm2');

CREATE TABLE Question ( idOfQcm integer,    numQuestion integer,   texte VARCHAR(64),
                                primary key (idOfQcm,numQuestion));
ALTER TABLE Question ADD CONSTRAINT question_avec_qcm_valide
                FOREIGN KEY  (idOfQcm) REFERENCES Qcm(idQcm);
INSERT INTO Question Values (1,2,"question2-qcm1"); INSERT INTO Question Values (1,1,"question1-qcm1");
INSERT INTO Question Values (2,1,"question1-qcm2"); INSERT INTO Question Values (2,2,"question2-qcm2");

CREATE TABLE Reponse ( idOfQcm integer,  numOfQuestion integer,  numReponse CHAR,
      texte VARCHAR(64), ok boolean, primary key (idOfQcm,numOfQuestion,numReponse));
ALTER TABLE Reponse ADD CONSTRAINT reponse_avec_question_valide
  FOREIGN KEY  (idOfQcm,numOfQuestion) REFERENCES Question(idOfQcm,numQuestion);

INSERT INTO Reponse Values (1,1,'A',"repA_question1-qcm1",false);
INSERT INTO Reponse Values (1,1,'B',"repB_question1-qcm1",true);
INSERT INTO Reponse Values (1,2,'A',"repA_question2-qcm1",true);
...
```

3.4. Quelques autres apports de JPA 2

JPA 1 était simplement capable de gérer une relation 1-n (vers des entités) en la définissant (via mappedBy="...") comme la relation inverse d'une relation n-1.
Autrement dit , la spécification de la colonne clef étrangère devait absolument être effectuée du coté "n-1" (via @JoinColumn à coté de @ManyToOne) et le lien devait donc être :

- soit unidirectionnel (que du coté n-1 / @ManyToOne)
- soit bidirectionnel

Depuis JPA 2 , il est maintenant possible d'établir un lien 1-n (unidirectionnel sans lien inverse n-1) en spécifiant la clef étrangère du coté 1-n (@JoinColumn proche de @OneToMany).
Bien que possible, cette nouvelle fonctionnalité ne devrait idéalement être utilisée que pour établir un lien unidirectionnel 1-n entre une entité d'un package et des entités d'un autre package métier/fonctionnel.

Entre deux types d'entités d'un même package les liaisons bidirectionnelles ne posent aucun problème.

4. Api "Criteria" de JPA 2

4.1. Apports et inconvénients de l'API "Criteria"

L'api "Criteria" (qui est apparue avec la version 2 de JPA) correspond à une syntaxe alternative à JQPL pour exprimer et déclencher des requêtes retournant des entités persistantes "JPA" .

Contrairement à JPQL , l'api des criteria ne s'appuie plus sur un dérivé de SQL mais s'appuie sur **une série d'expression de critères de recherche qui sont entièrement construits via des méthodes "java" .**

L'api "Criteria" est plus "orientée objet" et plus "fortement typée" que JPQL.

Le **principal avantage** réside dans **la détection de certaines erreurs dès la phase de compilation**.

A l'inverse, la plupart des erreurs dans l'expression d'une requête JPQL ne sont en général détectées que lors de l'exécution des tests unitaires.

Du coté négatif , **l'API "Criteria" de JPA n'est syntaxiquement pas très simple** (voir franchement complexe) et **il faut un certain temps pour s'y habituer (le code est moins "lisible" que JPQL au premier abord)** .

4.2. Exemple basique (pour la syntaxe et le déclenchement)

```
public List<Devise> getAllDevise() {
	/*return  this.entityManager.createQuery(
		"select d from Devise d",Devise.class).getResultList();*/

	//construction:
	CriteriaBuilder cb = entityManager.getCriteriaBuilder();
	CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

	//préparation (expression des critères de recherche)
	Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);
	criteriaQuery.select(deviseRoot);

	//déclechement (assez semblable) à celui d'une requête JPQL :
	return this.entityManager.createQuery(criteriaQuery).getResultList();
	}
```

4.3. Exemple avec prédicat pour partie ".where"

```
public Devise getDeviseByName(final String deviseName) {
      /* return  this.entityManager.createQuery(
            "select d from Devise d where d.monnaie = :deviseName",
Devise.class)
             .setParameter("deviseName",deviseName)
             .getSingleResult();*/
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Devise> criteriaQuery = cb.createQuery(Devise.class);

Root<Devise> deviseRoot = criteriaQuery.from(Devise.class);

Predicate pEqMonnaie = cb.equal(deviseRoot.get("monnaie") , deviseName);

criteriaQuery.select(deviseRoot);
criteriaQuery.where(pEqMonnaie);

return this.entityManager.createQuery(criteriaQuery).getSingleResult();
}
```

4.4. Exemple relativement simple avec jointure:

```
public List<Compte> findComptesByNumCli(long numCli) {
/*
return this.getEntityManager().createQuery("SELECT cpt FROM
Client cli  JOIN cli.comptes cpt WHERE cli.numero= :numCli",Compte.class)
.setParameter("numCli", numCli)
.getResultList();
*/
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Compte> criteriaQuery = cb.createQuery(Compte.class);

Root<Client> clientRoot = criteriaQuery.from(Client.class);


Predicate pEqNumCli = cb.equal(clientRoot.get("numero") , numCli);
//Predicate pEqNumCli = cb.equal(clientRoot.get(Client_.numero) , numCli);

Join<Client, Compte> joinComptesOfClient =  clientRoot.join("comptes");
//Join<Client, Compte> joinComptesOfClient =  clientRoot.join(Client_.comptes);


criteriaQuery.select(joinComptesOfClient);
criteriaQuery.where(pEqNumCli);

return this.entityManager.createQuery(criteriaQuery).getResultList();
}
```

NB: L'exemple précédent n'est pas "fortement typé" et certaines erreurs ("types incompatibles , "propriété n'existant pas ou mal orthographiée") peuvent passées inaperçues lors de la phase de compilation.

4.5. Classes "modèles" pour des expressions rigoureuses des critères de recherche :

```
...

//Predicate pEqNumCli = cb.equal(clientRoot.get("numero") , numCli);
Predicate pEqNumCli = cb.equal(clientRoot.get(Client_.numero) , numCli);

//Join<Client, Compte> joinComptesOfClient =  clientRoot.join("comptes");
Join<Client, Compte> joinComptesOfClient =  clientRoot.join(Client_.comptes);


....
```

Au sein de cette écriture beaucoup plus rigoureuse ,

Client_.numero sera interprété comme une propriété ".numero" qui existe bien sur une entité de type "Compte" et qui de plus est de type long .

Ces informations sont issue d'une classe java "modèle" dont le nom se termine (par convention) par un caractère "underscore" et dont le début coïncide avec la classe de l'entité persistante décrite.

(La classe modèle "Client_" décrit la structure de la classe "Client" préfixée par @Entity).

Une classe modèle doit absolument être placée dans le même package java que la classe d'entité persistante à décrire. On pourra cependant paramétrer maven et/ou l'IDE eclipse de façon à placer les classes "modèles" dans un autre répertoire car celles-ci sont sont générées automatiquement.

Exemple de classe "modèle"

```
package tp.myapp.minibank.core.entity;

import javax.annotation.Generated;
import javax.persistence.metamodel.ListAttribute;
import javax.persistence.metamodel.SingularAttribute;
import javax.persistence.metamodel.StaticMetamodel;

@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Client.class)
public abstract class Client_ {

	public static volatile SingularAttribute<Client, String> password;
	public static volatile SingularAttribute<Client, Long> numero;
	public static volatile ListAttribute<Client, Compte> comptes;
	public static volatile SingularAttribute<Client, Adresse> adresse;
	public static volatile SingularAttribute<Client, String> telephone;
	public static volatile SingularAttribute<Client, String> nom;
	public static volatile SingularAttribute<Client, String> prenom;
	public static volatile SingularAttribute<Client, String> email;
}
```

NB : La classe modèle _Client a été générée via un plugin maven "hibernate-jpamodelgen" (ou autre) qui peut se déclencher au moment de la compilation des classes ordinaires "Client" , …

…/...

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-jpamodelgen</artifactId>
      <version>4.3.1.Final</version>
      <scope>provided</scope>
</dependency>

...

<plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
            <source>1.7</source>
            <target>1.7</target>
            <compilerArguments>
                  <processor>
                  org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor
                  </processor>
            </compilerArguments>
        </configuration>
</plugin>
```

NB : il existe une autre façon de déclencher **`org.hibernate.jpamodelgen (plus paramétrable) .`**

Les classes modèles "XXX_" et les packages java sont par défaut générés dans le répertoire "**`target/generated-sources/annotations`**"

Il faut ensuite paramétrer le projet eclipse pour qu'il voit les classes générées :

1) activer le menu contextuel "**properties / compiler / annotation-processing**"
depuis le projet eclipse "….-ejb"
2) **enable specific** / "**target/generated-sources/annotations**"
3) ajouter une utilisation de XXX_.yyy dans le code du DAO

IX - Outils pour JPA , ...

1. Approches top-down , down-top , ...

Approches	*Principes*
top-down	On part d'une structure orientée objet (éventuellement modélisée via UML) considérée comme de haut niveau et l'on génère automatiquement des tables avec une structure compatible
down-top	On part d'une structure relationnelle existante (éventuellement modélisée via un MCD Merise) considérée comme de bas niveau et l'on génère automatiquement des classes d'entités persistantes avec une structure compatible
meet-in-the-middle	On se débrouille pour mapper au cas par cas une structure objet et une structure relationnelle (existantes toutes les 2) .

1.1. Rappel: configuration pour approche top-down

```
# si l'option spring.jpa.hibernate.ddl-auto=create est activée, toutes les
# tables nécessaires seront re-crées automatiquement à chaque démarrage
# à vide et en fonction de la structure des classes java (@Entity)
spring.jpa.hibernate.ddl-auto=create
```

```
#tables créés automatiquement au démarrage et fichier sql déclenché automatiquement :
...javax.persistence.schema-generation.database.action=drop-and-create
...javax.persistence.sql-load-script-source=META-INF/data.sql


#fichiers sql générés (pour consultation) mais pas déclenchés:
...javax.persistence.schema-generation.create-source=metadata
...javax.persistence.schema-generation.scripts.action=drop-and-create
...javax.persistence.schema-generation.scripts.create-target=src/main/script/create.sql
...javax.persistence.schema-generation.scripts.drop-target=src/main/script/drop.sql
```

1.2. Outils pour approche down-top

- assistants eclipse "**jpa-tools**"

- ...

2. Projet "JPA" d'eclipse

Mode opératoire à un peu adapter selon la version exacte d'eclipse (3.4 , 3.5 ou 3.6)

1) Paramétrer une connexion vers une base de données existante au niveau d'éclipse

   (menu "Windows / **Show View** / Other ... / connectivity / **DataSourceExplorer** / **New** .).
   Paramétrer à ce niveau l'accès à la base de données (chemin du ".jar" du driver JDBC , url , ...).
   Tester la connexion SQL via l'explorateur de base de données (menu "Windows / Show View / ...../ **DataSourceExplorer** / .... / **connect**).

2) Créer un nouveau ***projet JPA*** (ou bien ajouter la ***facette "JPA"*** à un projet java existant)

3) depuis le projet JPA déclencher le menu contextuel "***JPA Tools ..." / "Generate Entities"*** choisir la connexion SQL , le schéma de la base de données , les tables à prendre en comptes et quelques options (package java , ...)
   ==> **résultats = classes d'entités persistantes avec des annotations JPA** (@Entity , @Table , @Id , @OneToMany , ...).

Remarque:

Ceci permet de générer une version "brute" des classes d'entités persistantes.

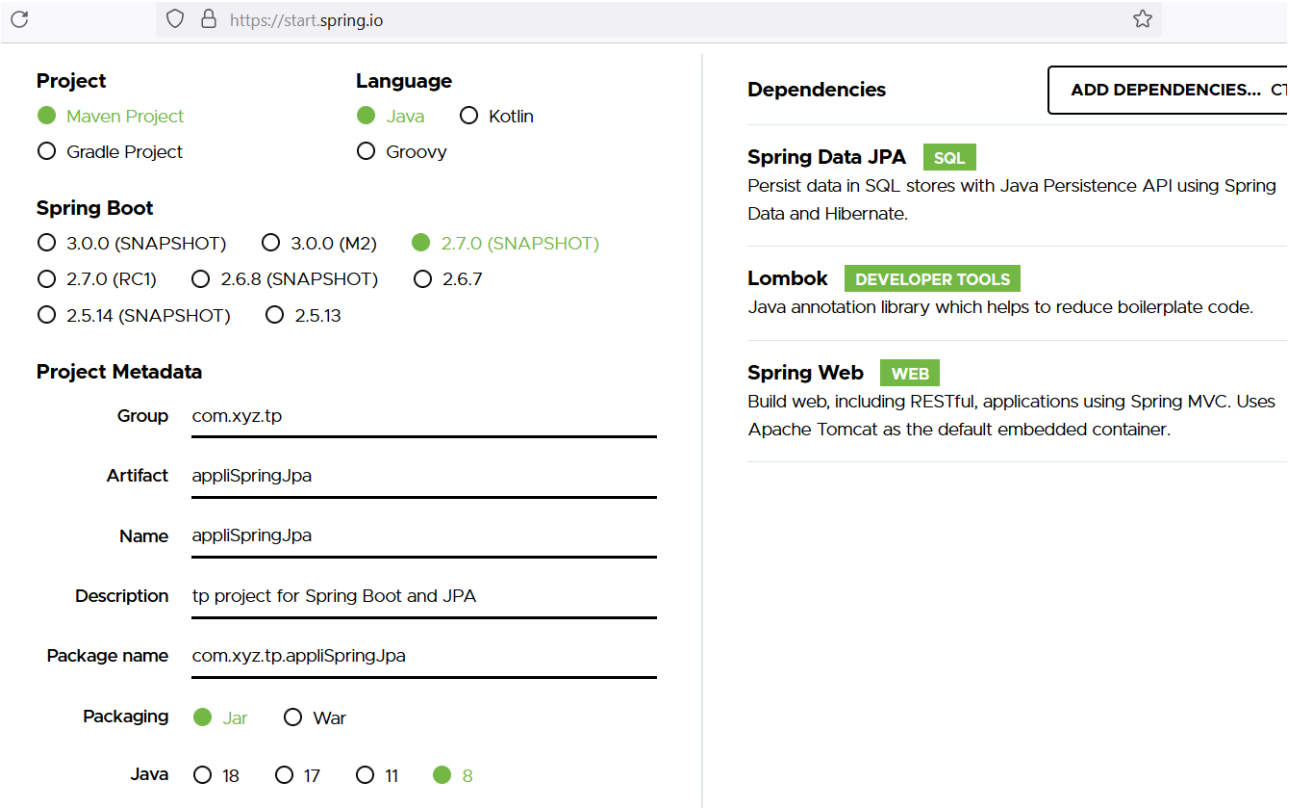
Il est souvent conseillé de recopier ces classes (par exemple dans un autre projet de type "maven") puis les adapter pour les améliorer un peu.

ANNEXES

X - Configuration JPA pour TP

1. Projet appliSpringJPa (pour TP)

Via le site web https://start.spring.io/ (***Spring boot initializer***)
effectuer à peu les paramétrages suivants :



NB: on pourra éventuellement choisir une version plus récente de java (11 ou 17) selon le contexte.

Cliquer ensuite sur "**Generate**" (en bas) pour générer le fichier **appliSpringJpa.zip** à télécharger .

Extraire le contenu du ".zip" téléchargé dans un répertoire local de Tp (ex: c:\tp**appliSpringJpa**)
puis charger le contenu de ce projet dans **eclipse** ou **intelliJ .**

Depuis eclipse (idéalement en version JEE_2021-12) , menu **import … / maven / inport maven project** et sélectionner le répertoire **c:\tp\appliSpringJpa** comportant **pom.xml .**

Ajuster ensuite (si besoin) le fichier pom.xml selon l'exemple suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>2.7.0</version>
```

```
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.xyz.tp</groupId>
	<artifactId>appliSpringJpa</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>appliSpringJpa</name>
	<description>Demo project for Spring Boot</description>
	<properties>
		<java.version>1.8</java.version>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<!-- sous eclipse , windows preferences general workspace text encoding utf8  -->
	</properties>

	<dependencies>

	<!-- lombok sert à générer automatiquement les get/set et toString()
  pas de configuration supplémentaire absolument nécessaire dans intellij
  pour que lombok fonctionne bien dans eclipse:
    1) repérer le chemin menant à lombok.jar
      $HOME\.m2\repository\org\projectlombok\lombok\1.18.22\lombok.jar
    2) effectuer un double click sur lombok.jar
      (ou bien java -jar lombok.jar)
    3) spécifier si besoin le chemin menant à eclipse
    4) cliquer sur "Install" de façon à modifier eclipse.ini
    5) arrêter et redémarrer eclipse
    6) activer le menu Project/clean
-->
	 <dependency>
		<groupId>org.projectlombok</groupId>
		<artifactId>lombok</artifactId>
		<version>1.18.22</version>
		<scope>provided</scope>
	  </dependency>


		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>

		<!-- NB: hibernate-core et jakarta-persistence-api sont des dépendances indirectes
			de  spring-boot-starter-data-jpa -->

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>


		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
```

```
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                    </plugin>
                    <plugin>
                        <artifactId>maven-resources-plugin</artifactId>
                        <version>3.0.2</version>
                        <!-- nb: version 3.2.0 avec bug a priori -->
                    </plugin>
                </plugins>
        </build>

</project>
```

Ajuster (si besoin) le fichier **src/main/resources/application.properties** avec ce genre de configuration :

```
server.servlet.context-path=/appliSpringJpa
server.port=8080
logging.level.root=ERROR
```

Ajuster (si besoin) le fichier **src/main/java/com/xyz/tp/appliSpringJpa/AppliSpringJpaApplication** avec ce genre de configuration :

```
package com.xyz.tp.appliSpringJpa;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AppliSpringJpaApplication {

        public static void main(String[] args) {
                SpringApplication.run(AppliSpringJpaApplication.class, args);
                System.out.println("http://localhost:8080/appliSpringJpa");
        }
}
```

Au sein de **src/main/resources** créer si besoin un sous répertoire **static** (via *new folder* ou autre) et placer y dedans le nouveau fichier html suivant: src\main\resources\static**index.html**

```
<html>
  <body>
    <p>appliSpringJpa</p>
  </body>
</html>
```

Pour effectuer un premier test de démarrage de l'application il faudra peut être mettre temporairement en commentaire la dépendance suivante au sein du fichier ***pom.xml*** :

```
                <!--    <dependency>
                            <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-starter-data-jpa</artifactId>
                        </dependency>       -->
```

2. Premiers TPs sans SpringBoot (JPA seul)

De manière à un peu expérimenter JPA sans spring on pourra mettre **temporairement** en place la configuration suivante dans le projet appliSpringJpa :

src/main/resources/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" 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">
<persistence-unit name="appliSpringJpa"  transaction-type="RESOURCE_LOCAL">
          <description>configuration JPA , base de données , ...</description>

          <!--  on précise ici la technologie choisie pour coder/implementer JPA
                Hibernate (ou OpenJpa ou EclipseLink) -->
           <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

       <!-- indiquer liste des classes avec @Entity (facultatif) -->
      <class>com.xyz.tp.appliSpringJpa.entity.Employe</class>

   <!-- configurer d'une façon ou d'une autre une connexion à la base de données -->
   <!--  aujourd'hui via des propriétés spécifiques Hibernate -->
     <properties>
          <!-- <property name="hibernate.hbm2ddl.auto" value="create-drop"/> -->
          <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect"/>
          <!-- <property name="hibernate.show_sql" value="true"/> -->
          <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
         <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/BaseQuiVaBien" />
         <property name="javax.persistence.jdbc.user" value="root" />
         <property name="javax.persistence.jdbc.password" value="root" />
     </properties>

</persistence-unit>
</persistence>
```

NB: sous eclipse, le répertoire META-INF peut éventuellement être ajouté via "new folder" et le fichier persistence.xml via "new file" .

Selon la version utilisée d'eclipse , il est quequefois possible de générer un début de fichier **META-INF/persistence.xml** via un assistant :
- sélectionner le projet ***appliSpringJpa*** dans projectExplorer .
- click droit / **properties**
- **project Facets**
- selectionner/cocher **JPA** (version 2.2)
- ***futher configuration available***
- choisir "**Disable Library Configuration**" pour Jpa Implementation
- choisir "**Discover annotated classes automatically**"
- "OK" , "Apply & close"

Réactiver si besoin la dépendance suivante dans **pom.xml** :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

src/main/java/com/xyz/tp/appliSpringJpa/**TestSansSpringApp.java**

```
package com.xyz.tp.appliSpringJpa;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.xyz.tp.appliSpringJpa.dao.DaoEmployeJpaSansSpring;
import com.xyz.tp.appliSpringJpa.entity.Employe;

// classe de démarrage de l'application (sans utiliser spring)
public class TestSansSpringApp {

public static void main(String[] args) {
        EntityManagerFactory emf =
                        Persistence.createEntityManagerFactory("appliSpringJpa");
        //NB: appliSpringJpa= name du persistent-unit configuré dans META-INF/persistence.xml

        EntityManager entityManager = emf.createEntityManager();

        DaoEmployeJpaSansSpring daoEmployeJpa = new DaoEmployeJpaSansSpring();
        daoEmployeJpa.setEntityManager(entityManager);

        Employe emp1 = new Employe(null,"prenom1","Nom","0102030405",
                                        "jean.Bon@xyz.com","login","pwd");
        daoEmployeJpa.insertNew(emp1);

        List<Employe> employes = daoEmployeJpa.findAll();
        for(Employe emp : employes) {
                System.out.println(emp);
        }

        entityManager.close();
        emf.close();
 }

}
```

NB: Tous les blocs de code en rouge de l'exemple ci-dessus ne sont nécessaires que dans le cas très rare où la technologie JPA/Hibernate est utilisée seule (sans spring ni EJB).
Ce cas de figure correspond avant tout à un cas d'écode (pédagogie progressive) et est très rare sur un véritable projet d'entreprise .

src/main/java/com/xyz/tp/appliSpringJpa/dao/***DaoEmployeJpaSansSpring.java***

```
package com.xyz.tp.appliSpringJpa.dao;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import com.xyz.tp.appliSpringJpa.entity.Employe;

/*
 *  Version "Sans spring" :
 *   - pas de @PersistenceContext
 *   - setEntityManager
 *   - gestion explicite des transactions entityManager.getTransaction()....
 */
public class DaoEmployeJpaSansSpring implements DaoEmploye{

private EntityManager entityManager;

public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
}

@Override
public Employe findById(Long code) {
        return entityManager.find(Employe.class, code);
                //SELECT .... WHERE empId=...
}

@Override
public List<Employe> findAll() {
        return entityManager.createQuery("SELECT e FROM Employe e",Employe.class)
                                .getResultList();
}

@Override
public Employe insertNew(Employe emp) {
    try {
        entityManager.getTransaction().begin();
        //en entrée , emp est un nouvel objet employé avec .empId à null (encore inconnu)
        //déclenche automatiquement INSERT INTO Employe(firstname, ....) VALUES(emp.getFirstname() , ....)
        entityManager.persist(emp);//.empId n'est normalement plus null si auto-incr
        entityManager.getTransaction().commit();
     } catch (Exception e) {
                entityManager.getTransaction().rollback();
                e.printStackTrace();
                throw new RuntimeException("echec insertNew(employe)");
      }
      return emp; //on retourne l'objet modifié (avec .empId non null)
}

@Override
public Employe update(Employe emp) {
```

```
        entityManager.getTransaction().begin();
        entityManager.merge(emp);
        //déclenche automatiquement UPDATE Employe set .... WHERE idEmp=code
        entityManager.getTransaction().commit();//à peaufiner via try/catch
        return emp;
}

@Override
public void deleteById(long code) {
        entityManager.getTransaction().begin();
        Employe empAsupprimer = entityManager.find(Employe.class, code);
        entityManager.remove(empAsupprimer);
          //déclenche automatiquement DELETE FROM Employe WHERE idEmp=code
        entityManager.getTransaction().commit(); //à peaufiner via try/catch
}

}
```

NB:
- pas d'auto-commit avec JPA/Hibernate .
- toutes les opérations qui modifient des valeurs dans la base de données nécessitent absolument une gestion explicite des transactions.
- seules les opérations en lecture seulement peuvent s'effectuer sans transaction
- le besoin de gérer finement les transactions revient tellement souvent qu'il a été automatisé en utilisant à fond le framework EJB ou le framework Spring .

3. Configuration de base de données par scripts

init-db.sql

```
CREATE DATABASE IF NOT EXISTS BaseQuiVaBien ;
use BaseQuiVaBien;

DROP TABLE IF EXISTS employe;

CREATE TABLE employe(
        EMP_ID INTEGER auto_increment,
        firstname VARCHAR(64),
        lastname VARCHAR(64),
        PHONE_NUMBER VARCHAR(64),
        email VARCHAR(64),
        LOGIN VARCHAR(32),
        password VARCHAR(64),
        PRIMARY KEY(EMP_ID));

INSERT INTO employe (EMP_ID,firstname,lastname,PHONE_NUMBER,email,LOGIN,password)
  VALUES (1,'alain', 'Therieur' , '0102030405' , 'alain.therieur@xyz.com','login1','pwd1');
INSERT INTO employe (EMP_ID,firstname,lastname,PHONE_NUMBER,email,LOGIN,password)
  VALUES (2,'axelle', 'Aire' , '0102030405' , 'axelle.aire@m2i.com','login2','pwd2');

SELECT * FROM employe;
```

et si besoin

```
ALTER TABLE Table2 ADD CONSTRAINT t2_associe_a_t1_valide
FOREIGN KEY (nomClefEtrangere) REFERENCES t1(nomClefPrimaire);
```

init-db.bat

```
REM se placer dans le répertoire courant:
cd /d %~dp0

REM MariaDB est une version complètement open source de MySQL (plus facile à installer)
set MYSQL_HOME=C:\Program Files\MariaDB 10.6


REM option -p pour demander à saisir le mot de passe (ex: root)
"%MYSQL_HOME%/bin/mysql" -u root -p < init-db.sql

pause
```

4. Config JPA avec SpringBoot et transactions

JPA a été à l'origine conçu pour s'intégrer dans un projet à base d'EJB >=3 .
Les EJB 3.x gèrent très bien les transactions et intègre très bien JPA via l'annotation @PersistenceContext() à placer au sein d'un DAO pour initialiser automatiquement l'objet technique entityManager .

Le framework Spring est lui aussi capable de gérer automatiquement/implicitement les transactions.
L'annotation @PersistenceContext() est également utilisable au sein d'un projet Spring avec un comportement identique à la technologie EJB .

4.1. Comportement commun aux EJB et à Spring :

Dès le démarrage d'un projet EJB ou d'un projet Spring , un équivalent de

```
EntityManagerFactory emf =
                        Persistence.createEntityManagerFactory("persistentUnitName");
```

est automatiquement déclenché .

Placée dans un DAO juste au dessus de ***private EntityManager entityManager ;*** l'annotation **@PersistenceContext("*persistentUnitName*")** permet de déclencher automatiquement au moments opportuns:

```
EntityManager entityManager = emf.createEntityManager();
```

et

```
entityManager.close();
```

Finalement certaines annotations (de type **@Transactional**) placées sur la classe d'un DAO permettent de déclencher automatiquement aux moments opportuns des instructions de typeCompte

```
entityManager.getTransaction().begin();
```

et

```
try {
        entityManager.getTransaction().begin();
        // ....
        entityManager.getTransaction().commit();
    } catch (Exception e) {
                entityManager.getTransaction().rollback();
                // ...
     }
```

Dans les grandes lignes et en allant à l'essentiel :
- commit automatique si pas d'exception .
- rollback automatique en cas d'exception héritant de RuntimeException .
- propagation automatique du contexte transactionnel entre service métiers et DAO .
- en fin de transaction réussie : flush() automatique des modifications effectuées sur objets persistants

4.2. Configuration au cas par cas (selon type de projet)

Type de projets	*Configuration JPA et DataSource (connexion DB)*
JPA/Hibernate seul	**META-INF/persistence.xml** + code du main()
JPA dans EJB3 et serveur JEE	**META-INF/persistence.xml** + configuration spécifique au serveur (ex : standalone.xml de Jboss)
JPA dans Spring ancien	**META-INF/persistence.xml** + gros complément dans configuration Spring souvent XML
JPA dans springBoot moderne	Tout dans **application.properties** ou bien **application.yml** *(avec éventuels profils complémentaires/aternatifs possibles)*

4.3. Configuration JPA au sein d'un projet Spring-boot

En démarrage "spring boot" , ~~META-INF/persistence.xml~~ n'est pas analysé (pas pris en compte).

Par contre toutes les lignes préfixées par ***spring.datasource*** et ***spring.jpa*** du fichier ***application.properties*** sont **analysées et prises en compte** .

src/main/resources/**application.properties**

```
server.servlet.context-path=/appliSpringJpa
server.port=8080
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/BaseQuiVaBien?serverTimezone=UTC
#jdbc:mysql://localhost:3306/DbXy?createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

# si l'option spring.jpa.hibernate.ddl-auto=create est activée
# toutes les tables nécessaires seront re-crées automatiquement à chaque démarrage
# à vide et en fonction de la structure des classes java (@Entity)
spring.jpa.hibernate.ddl-auto=create

#enable spring-data (generated dao implementation classes)
#spring.data.jpa.repositories.enabled=true

spring.jpa.properties.javax.persistence.schema-generation.database.action=drop-and-create
#spring.jpa.properties.javax.persistence.sql-load-script-source=META-INF/data.sql
spring.jpa.properties.javax.persistence.schema-generation.create-source=metadata
spring.jpa.properties.javax.persistence.schema-generation.scripts.action=drop-and-create
spring.jpa.properties.javax.persistence.schema-generation.scripts.create-target=src/main/script/create.sql
spring.jpa.properties.javax.persistence.schema-generation.scripts.drop-target=src/main/script/drop.sql
```

Si postgres à la place de mysql :

```
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/xyzDb
spring.datasource.username=postgres
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
```

Structure élémentaire du code pour DAO JPA intégré dans spring (sans spring-data) :

```
package com.xyz.tp.appliSpringJpa.dao;

import java.util.List;

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

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

import com.xyz.tp.appliSpringJpa.entity.Employe;


/*
 * Cette version du DAO sera utilisé par Spring+JPA
 * et spring va initialiser automatiquement le entityManager
 * grace à l'annotation @PersistenceContext.
 *
 * Spring va aussi déclencher automatiquement des commit/rollback
 * si on place @Transactional sur la classe ou une méthode
 */

@Repository  //cas particulier de @Component //pour prise en charge par framework spring
@Transactional //pour commit/rollback automatique
public class DaoEmployeJpa  implements DaoEmploye{

	//NB: @PersistenceContext permet d'initialiser l'objet technique
	//entityManager à partir d'une configuration
	//  src/main/resources/META-INF/persistence.xml
	//  ou bien config spring équivalente dans src/main/resources/application.properties
	@PersistenceContext
	protected EntityManager entityManager;


	@Override
	public List<Employe> findAll() {
		return entityManager.createQuery("SELECT e FROM Employe e",Employe.class)
					.getResultList();
	}
…
}
```

Structure élémentaire du code pour un service métier spring :

```
package com.xyz.tp.appliSpringJpa.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.xyz.tp.appliSpringJpa.dao.DaoEmploye;
import com.xyz.tp.appliSpringJpa.entity.Employe;

@Service //cas particulier de @Component
@Transactional
public class ServiceEmployeImpl implements ServiceEmploye {

	@Autowired  //injection de dépendance
	private DaoEmploye daoEmploye;

	@Override
	public List<Employe> rechercherTousEmploye() {
			return daoEmploye.findAll();  //simple délégation de traitement içi
	}

	//Un service fait souvent en plus des verifs de règles de gestion
}
```

Structure élémentaire du code pour un test de DAO JPA intégré à spring-boot :

```
package com.xyz.tp.appliSpringJpa;

import java.util.List;  import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import com.xyz.tp.appliSpringJpa.dao.DaoEmploye;
import com.xyz.tp.appliSpringJpa.entity.Adresse;   import com.xyz.tp.appliSpringJpa.entity.Employe;

@SpringBootTest
class AppliSpringJpaApplicationTests {

	//initialise daoEmploye pour que ça référence un composant pris en charge
	//par Spring et qui est compatible avec l'interface DaoEmploye
	//dans ce projet , seule la classe DaoEmployeJpa correspond à ce critère
	@Autowired //injection de dépendance via Spring
	private DaoEmploye daoEmploye;

	@Test
	public void testFindEmployeWithNameBeginBy() {
		System.out.println("Liste des employes dont le nom commence par A :");
		List<Employe> employes = daoEmploye.findEmployeWithNameBeginBy("A");
		for(Employe emp : employes) {
			System.out.println("\t" +emp);
		}
	}
```

```
@Test
public void testAvecSpring() {
	//séquence de test idéale:
	//1. créer une nouvelle chose
	Employe emp1 = new Employe(null,"prenom1","Nom","0102030405",
					"jean.Bon@xyz.com","login","pwd");
	Adresse adr1 = new Adresse(null,"12 rue Elle","75001","Par ici");
	emp1.setAdressePrincipale(adr1);

	daoEmploye.insertNew(emp1);
	Long idEmp = emp1.getEmpId(); //clef primaire auto incrémentée
					//du nouvel employé ajouté en base
	//2. afficher tout pour vérifier l'ajout
	List<Employe> employes = daoEmploye.findAll();
	for(Employe emp : employes) {
			System.out.println(emp);
	}

	//3. récupérer une entité précise via sa clef primaire et l'afficher
	Employe emp1Relu = daoEmploye.findById(idEmp);
	System.out.println("emp1Relu=" + emp1Relu);
	System.out.println("avec adresse=" + emp1Relu.getAdressePrincipale());

	//4. modifier les valeurs en mémoire puis en base
	emp1Relu.setPhoneNumber("0504030201");
	daoEmploye.update(emp1Relu);

	//5. re-déclencher étape 3 pour vérifier la mise à jour en base
	Employe emp1Relu2 = daoEmploye.findById(idEmp);
	System.out.println("emp1Relu2 apres update =" + emp1Relu2);

	//6. supprimer la chose ajoutée à l'étape 1
	daoEmploye.deleteById(idEmp);

	//7. on vérifie que ça n'existe plus
	Employe emp1Relu3NormalementNull = daoEmploye.findById(idEmp);
	if(emp1Relu3NormalementNull==null)
		System.out.println("employe bien supprimé dans la base de données");
}
}
```

NB:

Ce code de test élémentaire (volontairement simple) est beaucoup **améliorable** via :

- des assertions de Junit 4 ou 5

- l'utilisation d'une api de log à la place de System.out.println()

- ….