JPA
La persistance

Youssef Saadi

Master Informatique Décisionnelle

Faculté Des Sciences Et Techniques
Université Sultan Moulay Slimane Béni-Mellal

AU: 2019/2020

1ère Partie :
Unité de persistance

Unité de persistance

- L'unité de persistance est la « boîte noire » qui permet de rendre persistants les Entity Beans.
- Une unité de persistance (*Persistence Unit*) est caractérisée par les points suivants :
  - un ensemble **d'Entity Beans**,
  - un fournisseur de persistance (***Provider***),
  - une source de données (***Datasource***).

Unité de persistance

- Une entité persistante étant vouée à être enregistrée dans une source de données, le rôle de l'unité de persistance est :
  - De savoir où et comment stocker les informations.
  - De s'assurer de **l'unicité** des **instances** de chaque identité persistante.
  - De gérer les instances et leur cycle de vie : c'est le gestionnaire d'entité (***Entity Manager***).

Entity Bean : état attaché / détaché

- Quand un Entity Bean est **attaché** à un contexte de persistance, les modifications appliquées à l'objet sont alors automatiquement synchronisées avec la base de données, *via* l'Entity Manager.
- un Entity Bean est dit **détaché** lorsqu'il n'a plus aucun lien avec l'Entity Manager.

Intégration et packaging d'une unité de persistance

- L'utilisation des Entity Beans n'est plus fermée au monde J2EE ou à un conteneur EJB. Vous pouvez désormais « déployer » des Entity Beans dans de nombreuses applications :
  - Application Entreprise (EAR).
  - Module Java Bean Entreprise (EJB-JAR).
  - Application web (WAR).
  - Client d'application entreprise (JAR).
  - Un environnement Java SE compatible.

Paramétrage de l'unité de persistance

- La nouvelle spécification définit un fichier qui regroupe l'ensemble des informations de persistance. Vous devez nommer ce fichier : « **persistence.xml** » et placer ce fichier dans le répertoire « META-INF », à la racine du projet.
- La balise racine **<persistence>** ne contient que des balises **<persistence-unit>.**
- **<persistence-unit>** (0-N) : déclare une unité de persistance.
  - L'attribut **name** affecte un nom unique à cette unité dans votre application.
  - L'attribut **type** définit si l'unité de persistance est gérée et intégrée dans une Java EE (**JTA**) ou si vous souhaitez gérer de façon manuelle les transactions (**RESOURCE_LOCAL**) *via* l'Entity Manager.

Paramétrage de l'unité de persistance

- Les balises qui permettent de paramétrer une unité de persistance:
- **<description>**
- **<provider>** : permet de définir la classe d'implémentation du fournisseur de persistance utilisé dans l'application:
  - JBoss utilise Hibernate : ***org.hibernate.ejb.HibernatePersistence***
  - Oracle Application Server utilise TopLink : ***oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider***
  - GlassFish (Sun Application Server 9) utilise TopLink : ***oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider.***
  - En environnement Java SE, vous devez spécifier le fournisseur de persistance.

Paramétrage de l'unité de persistance

- **<jta-datasource>** : permet de définir le nom JNDI de la source de données transactionnelle à utiliser.
- **<non-jta-datasource>** (0-1) : permet de définir le nom JNDI de la source de données non transactionnelle à utiliser.
- **<mapping-file>** (0-N) : permet de définir le fichier de *mapping* XML pour les Entity Beans (lorsqu'il n'est pas possible d'utiliser les annotations).
- **<properties>** (0-N) : Cette balise vous permet de configurer les attributs de configuration du fournisseur de persistance.

Exemple d'une unité de persistance

```
<?xml version="1.0"?>
<persistence version="1.0">
  <persistence-unit name="transactionUP">
    <jta-data-source>java:StockTransactionDS</jta-data-source>
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>com.labosun.stockmanager.entity.Transaction</class>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLInnoDBDialect" />
    </properties>
    <exclude-unlisted-classes />
  </persistence-unit>
```

Cycle de vie du contexte de persistance

- Le contexte de persistance représenté par l'Entity Manager est lui aussi soumis à une durée de vie.
  - « **transaction-scoped persistence context** » : durée de vie liée à une seule transaction.
  - « **extended persistence contexts** » : durée de vie liée à celle d'un Stateful Session Bean (et donc plusieurs transactions).

Transaction-scoped persistence context

- Le contexte de persistance s'initialise lorsque l'application demande un Entity Manager au sein d'une transaction déjà active.
- Le contexte persistant s'arrête quand la transaction associée est validée ou annulée.

```
@Stateless
public class WebClientServiceBean implements WebClientService {
  @PersistenceContext(unitName="stockManagerUP")
  EntityManager em;

  public User createUser(User user) {
    em.persist(user);
    user.setValid(false);
    return user;
  }
}
```

Extended persistence context

- Chaque instance d'Entity Bean *managée* par ce type de contexte le reste même après la fin d'une transaction. Les instances sont détachées à la fermeture du contexte.
- Dans ce cas, la durée de vie d'un contexte de persistance étendu doit être gérée par l'application (de la création jusqu'à la suppression).
- Toutefois, cette gestion peut être automatiquement gérée avec un Stateful Session Bean.

Extended persistence context

```
@Stateful
public class WebClientServiceBean implements WebClientService {
  @PersistenceContext(unitName="stockManagerUP",
    type=PersistenceContextType.EXTENDED)
  EntityManager em;

  private User currentUser;

  public User loadUser(int idUser) {
    currentUser = em.find(User.class, idUser);
    return currentUser;
  }
  public void updateUser(User updatedUser) {
```

Extended persistence context

```
        currentUser.setAddress(updatedUser.getAddress());
        currentUser.setFirstName(updatedUser.getFirstName());
    }
}
```

Persistance via l'entité manager

- L'Entity Manager étant une interface, la classe d'implémentation est différente suivant le fournisseur utilisé (défini dans le fichier « persistence.xml » avec la balise <provider>).
- Par défaut, c'est le conteneur qui instanciera l'Entity Manager et qui gérera son cycle de vie. On parle alors de gestionnaire d'entités géré par le conteneur (***container-managed entity manager***).
- Il vous est cependant possible de gérer manuellement ce cycle de vie. On parle alors de gestionnaire d'entité géré par l'application (***application-manager entity manager***).

Obtenir un Entity Manager

- Il existe plusieurs manières d'obtenir un objet **Entity Manager** qui ont chacune des spécificités quant à leur utilisation :
  - Injection par **annotation** : utilisée exclusivement en environnement Java EE.
  - Utilisation de la **fabrique EntityManagerFactory** : utilisée pour gérer de façon manuelle la création **d'EntityManager**. Elle est obligatoire pour une application Java SE.
- Dans la plupart des cas, la manipulation des données *via* un conteneur EJB se réalise en définissant un Session Bean en « **façade** ».
- Ce Session Bean accède donc au contexte de persistance et peut ensuite travailler avec les instances des Entity Beans, *via* l'Entity Manager.

L'interface EntityManagerFactory

- Cette interface propose plusieurs méthodes de création :
  - EntityManager **createEntityManager**() : retourne une instance de EntityManager de type « extended ».
  - EntityManager **createEntityManager(Map** properties) : retourne une instance de EntityManager en utilisant les propriétés passées en paramètre.
  - void **close**() : ferme la EntityManagerFactory, permettant de libérer les ressources qu'elle utilise.
  - boolean **isOpen**() : permet de connaître si l'EntityManagerFactory est toujours valide.

@PersistenceContext

- En environnement JEE, afin d'indiquer au conteneur que le Bean est dépendant d'un **EntityManagerFactory** ou d'un **EntityManager**, vous devez annoter la variable d'instance de type EntityManagerFactory ou EntityManager avec **@PersistenceContext**. Cette annotation admet plusieurs attributs :
  - **name** : déclare un nom local référencé sur une unité de persistance déployée
  - **unitName** : définit le nom de l'unité de persistance à utiliser pour injecter l'EntityManager.
  - **type** : indique le type contexte de persistance injecté (**EXTENDED** ou **TRANSACTION** par défaut).

@PersistenceContext

```
@Stateless
@Local({CommonService.class})
public class CommonServiceBean implements CommonService {

  @PersistenceContext(unitName="stockmanagerUP")
  protected EntityManagerFactory emStockManagerFactory;
  //...
}
```

EntityManagerFactory

```
//...
  @PersistenceContext(unitName="stockmanagerUP")
  protected EntityManagerFactory emStockManagerFactory;

  public void addUser(User user) {
    EntityManager em = emStockManagerFactory.createEntityManager();
    em.jointTransaction();
    //...
  }
```

EntityManager

- Comme vous pouvez le constater, l'utilisation de l'EntityManagerFactory est une charge pour les développeurs. Il est, cependant, possible de laisser cette charge au conteneur en lui demandant d'injecter directement une instance **EntityManager**.

```
@Stateless
@Local({CommonService.class})
public class CommonServiceBean implements CommonService {

  @PersistenceContext(unitName="stockmanagerUP")
  protected EntityManager emStockManager;
  //...
}
```

EntityManager & *extended persistence context*

- L'injection d'un *extended persistence context* ne prend de sens que dans un Stateful Session Bean. En effet, la durée de vie de celui-ci a réellement un intérêt dans l'application.

```
@Stateful
@Local({RichClientService.class})
public class RichClientServiceBean implements RichClientService {

  @PersistenceContext(unitName="stockmanagerUP",
    type=PersistenceContextType.EXTENDED)
  protected EntityManager emStockManager;
  //...
}
```

Gestion manuelle : en environnement Java SE

```
public class Launcher {

  public static void main(String[] args) {
    // Utilise la configuration persistence.xml
    EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("contactUP");
    // Retrieve an application managed entity manager
    EntityManager em = emf.createEntityManager();

    // Crée une instance non managée de Contact
    Contact contact = new Contact();
    contact.setFirstName("Cyril");
    contact.setLastName("JOUI");
    // démarre la transaction
    em.getTransaction().begin();
    // opérations dans la transaction
    em.persist(contact);
    //...

    // valide la transaction
    em.getTransaction().commit();

    // ferme le contexte de persistance
    em.close();
    emf.close();
  }
}
```

Les opérations CRUD : Insertion

```
User newUser = new User();
// affecte les relations
newUser.setFirstname("Cyril");
newUser.setEmail("popom@supinfo.com");
//...
entityManager.persist(newUser);
```

Les opérations CRUD : Lecture

- L'Entity Manager a deux méthodes simples pour trouver une entité à partir de sa clé primaire.
  - **<T> T find(Class<T> entity, Object primaryKey)**
  - **<T> T getReference(Class<T> entity, Object primaryKey)**
- La méthode **find**() retourne **null** si aucune entité n'est associée à la clé primaire demandée. Elle initialise également les états de base du ***lazy-loading*** associés aux propriétés.
- La méthode **getReference**() se comporte différemment si l'entité n'est pas trouvée en base de données. Dans ce cas-là, elle lance une **javax.persistence.EntityNotFoundException**. De plus, elle ne garantit pas l'initialisation des états de l'entité.

Lecture

```
User user1 = entityManager.find(User.class, 1);
if(user1 == null) { ... }
```

Les opérations CRUD : Mise à jour

- Lorsque vous récupérez une entité *via* find(), getReference() ou par l'intermédiaire d'une requête, celle-ci se retrouve *managée* jusqu'à la fermeture du contexte de persistance. Vous pouvez alors changer les propriétés de cette instance, les modifications seront synchronisées automatiquement.

```
public void changeUserEmail(int userId, String email) {
  User currentUser = entityManager.find(User.class, userId);
  currentUser.setEmail(email);
}
```

Les opérations CRUD : Mise à jour

- Soit un client qui appelle la méthode findUser.
- Le contexte de persistance se ferme après l'exécution de la méthode findUser() car il est de type « transaction-scoped » (nous supposons que le Session Bean utilise la gestion par défaut des transactions).

```
@PersistenceContext
EntityManager em;

public User findUser(int userId) {
  return em.find(User.class, userId);
}
```

Les opérations CRUD : Mise à jour

- À partir de là, l'instance retournée est détachée et les modifications appliquées ne sont plus synchronisées. Si l'application cliente souhaite enregistrer les modifications faites en local, elle doit renvoyer l'instance au Session Bean.

```
User returnedUser = adminService.findUser(2);
returnedUser.setEmail("nouvelemail@supinfo.com");
adminService.updateUser(returnedUser);
```

Les opérations CRUD : Mise à jour

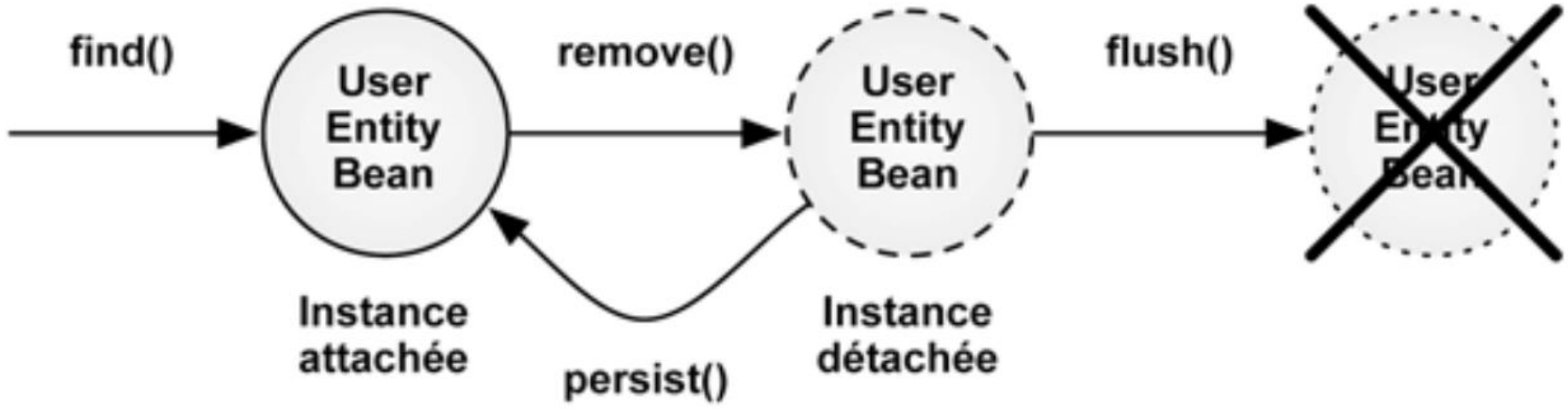
- Dans ce cas-là, la méthode updateUser() doit utiliser EntityManager.merge() afin de rattacher l'instance au contexte.

```
@PersistenceContext
EntityManager em;

public void updateUser(User user) {
  User attachedUser = em.merge(user);
}
```

Les opérations CRUD : Suppression

- La méthode **EntityManager.remove**() sert à demander la suppression d'une entité de la base de données. Nous parlons de « demande » car la suppression n'est pas effective immédiatement mais seulement à l'appel de la méthode **flush**() ou à la fermeture du contexte de persistance.



Les opérations CRUD : Suppression

- L'appel à la méthode **remove**() détache l'entité du contexte de persistance.
- Pour annuler cette suppression (dans le même contexte de persistance) il faut appeler la méthode **persist**() afin de rattacher l'instance au contexte.

```
@PersistenceContext
EntityManager em;



public void removeUser(int userId) {
  User userToRemove = em.find(userId);
  em.remove(userToRemove);
}
```

Recharge d'une instance

- Si vous savez que l'instance d'une entité ne reflète pas les valeurs de la base de données (parce que celle-ci a été modifiée entre-temps…) vous pouvez utiliser la méthode **EntityManager.refresh**() afin de recharger l'entité depuis la base de données.

```
@PersistenceContext
EntityManager em;

public void workOnUser(int userId) {
  User user = em.find(userId);
  user.setEmail("nomail@supinfo.com");
  // recharge l'objet depuis la base de données
  em.refresh(user);
}
```

Flush

- Lorsque vous appelez les méthodes persist(), merge() ou remove(), les changements ne sont pas synchronisés immédiatement.
- Cette synchronisation s'établit automatiquement à la fin d'une transaction ou lorsque l'Entity Manager décide de vider (*to flush*, en anglais) sa file d'attente.
- Toutefois, le développeur peut forcer la synchronisation en appelant explicitement la méthode flush() de l'Entity Manager.

Autres opérations de l'Entity Manager

- boolean **contains(**Object entity) : retourne true si l'entité passée en paramètre est attachée au contexte persistant courant.
- void **clear**() : permet de détacher l'ensemble des entités liées au contexte de persistance courant.

Cycle de vie d'un Entity Bean

- ***new*** (nouveau) : signifie que l'instance n'est associée à aucun contexte persistant. Cet état résulte de l'instanciation de l'Entity Bean *via* new.
- ***managed*** (gérée) : signifie que l'instance possède une identité associée au contexte persistant. Une instance est généralement dans cet état lorsqu'elle vient d'être enregistrée, modifiée ou récupérée.
- ***detached*** (dissociée) : signifie que l'instance n'est plus associée au contexte persistant d'où elle provient. C'est généralement le cas lorsque l'instance est initialisée dans le conteneur puis envoyé à un autre tiers (présentation, web…).
- ***removed*** (supprimée) : signifie que l'instance a une identité associée à un contexte persistant mais qu'elle est destinée à être retirée de la base de données.

Annotations du cycle de vie

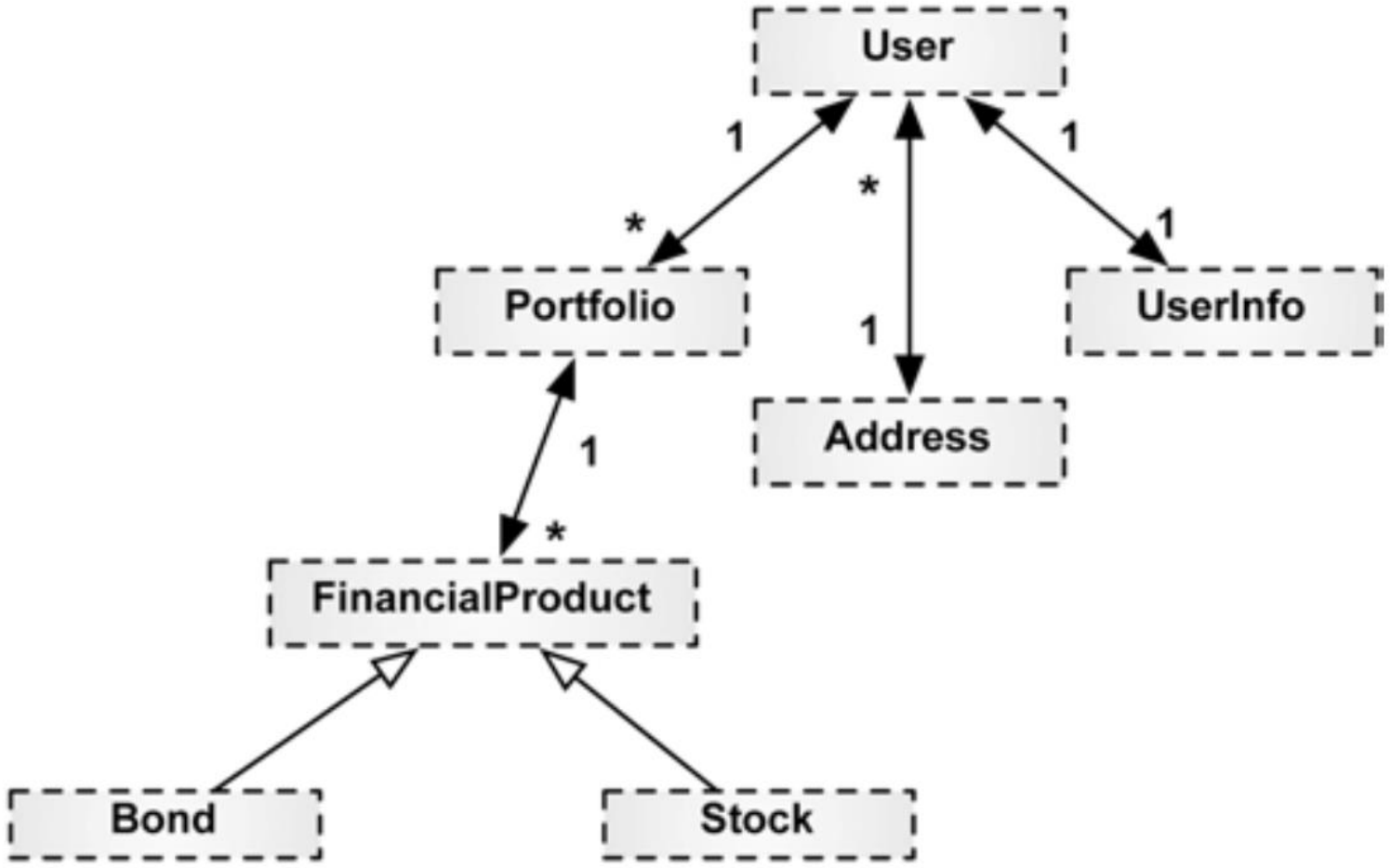
- Les méthodes annotées avec **@PrePersist** ou **@PreRemove** sont invoquées sur un Entity Bean **avant** l'exécution des méthodes persist() et remove() de l'Entity Manager.
- Les méthodes annotées avec **@PostPersist** ou **@PostRemove** sont invoquées sur un Entity Bean **après** l'exécution des méthodes persist() et remove() de l'Entity Manager.
- Les méthodes annotées avec **@PreUpdate** ou **@PostUpdate** sont invoquées sur un Entity Bean respectivement **avant** ou **après** la mise à jour de la base de données.

2ème Partie :
EJB-QL

Introduction

- EJB-QL (Entreprise JavaBean Query Language) est une spécification de langage de requêtes. Intégré au système EJB.
- Ce langage offre une portabilité qui lui permet d'être utilisé à l'identique entre les différentes versions du langage SQL.
- EJB-QL permet d'utiliser les objets des Entity Beans de l'application directement dans les requêtes.

Schéma abstrait des exemples



Sélection

```
String queryString = "SELECT user FROM User AS user";
Query query = entityManager.createQuery(queryString);
```

```
List<User> allUsers = query.resultList();

for (User u : allUsers ) {
  // Traitements sur l'utilisateur …
}
```

Sélection

- On devine assez clairement que l'utilisation du « . » est similaire, dans cet exemple, à appeler la méthode ***Portfolio.getUser()***.

```
SELECT portfolio FROM Portfolio AS portfolio WHERE portfolio.user.id=5
```

Suppression et mise à jour

- Les requêtes DELETE et UPDATE ont la particularité commune est de n'avoir qu'une seule entité dans la clause FROM.

```
DELETE FROM User AS user
WHERE user.id = 5
```

```
UPDATE User AS user
SET user.firstName = "Durand"
WHERE user.id = 1
```

Remarque

- La clause DELETE d'EJB-QL ne supporte pas la suppression en cascade.
- Nous avons alors trois possibilités :
  - Utiliser le gestionnaire de persistance qui applique la suppression en cascade.
  - Écrire explicitement toutes les requêtes EJB-QL de suppressions dans le bon ordre.
  - Utiliser les possibilités de la base de données et gérer la suppression en cascade au niveau de celle-ci.

Clause Where et Between

```
SELECT user FROM User AS user
WHERE user.id BETWEEN 1500 AND 2000
```

```
SELECT user FROM User AS user
WHERE user.firstName LIKE 'jean%'
```

IN ; IS NULL; Order By;

```
SELECT user FROM User AS user
WHERE user.address.country IN  ('France', 'Spain', 'Belgium')
```

```
SELECT user FROM User AS user
WHERE user.login IS NULL
```

```
SELECT user
FROM User AS user
ORDER BY user.id ASC
```

```
SELECT user
FROM User AS user
ORDER BY user.lastName, user.firstName ASC
```

MEMBER OF

- MEMBER OF : teste l'appartenance d'une instance à une collection, tout comme la méthode contains() de l'interface java.util.Collection.

```
SELECT user
FROM User AS user
WHERE ?1 MEMBER OF user.portfolios
```

IS EMPTY

- EMPTY : teste si une collection est vide. Par exemple, nous allons ici retourner la liste des utilisateurs n'ayant aucun portefeuille d'action.

```
SELECT user
FROM User AS user
WHERE user.portfolios IS EMPTY
```

Les fonctions d'agrégation

- Les fonctions d'agrégation servent à effectuer des opérations sur des ensembles. On les retrouve en EJB-QL *via* les instructions suivantes :

- avg() : retourne une moyenne par groupe.
- count() : retourne le nombre d'enregistrements.
- max() : retourne la valeur la plus élevée.
- min() : retourne la valeur la plus basse.
- sum() : retourne la somme des valeurs.

Les fonctions d'agrégation

```
SELECT COUNT(user)
FROM User AS user
```

```
SELECT user.firstName, COUNT(user.portfolios) AS nbrPortfolio
FROM User AS user
GROUP BY user
```

HAVING

```
SELECT user.firstName, COUNT(user.portfolios) AS nbrPortfolio
FROM User AS user
GROUP BY user.firstName
HAVING nbrPortfolio > 2
```

Opérateur IN

- Il doit être placé dans la clause FROM, et permet de déclarer un alias pour les entrées d'une collection. Par exemple, nous pouvons récupérer l'ensemble des portefeuilles créés, comme suit :

```
SELECT portfolio
FROM User AS user, IN (user.portfolios) portfolio
```

Autre exemple : IN

```
SELECT fp
FROM User AS user,
  IN (user.portfolios) portfolio,
  IN (portfolio.financialProducts) fp
WHERE user.id = 42 AND fp.quantity > 10
```

Les jointures

- Thêta jointure

```
SELECT user.firstName, ai.cardNumber
FROM User AS user, AccountInfo AS ai
WHERE u.accountInfo.id = ai.id
```

- En utilisant les propriétés relationnelles :

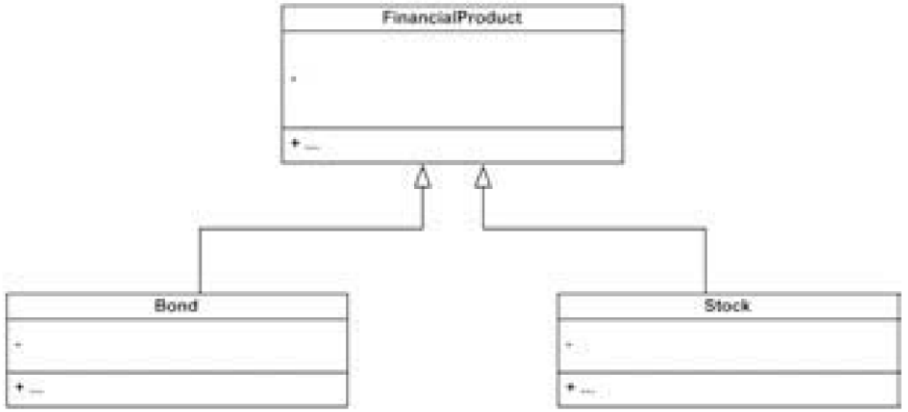
```
SELECT user.firstName, user.accountInfo.cardNumber
FROM User AS user
```

Les jointures

- En utilisant l'opérateur JOIN

```
SELECT user.firstName, ai.cardNumber
FROM User AS user
JOIN user.accountInfo AS ai
```

Le polymorphisme




```
SELECT fi FROM FinancialProduct AS fi
```

Sous requête

```
SELECT user
FROM User user
WHERE (
   SELECT COUNT(portfolio)
   FROM Portfolio AS portfolio
   WHERE portfolio.user=user
   GROUP BY user
) >3
```

L'API Query
L'interface javax.persistence.Query.

- L'API Query est le point essentiel pour la jonction entre l'application et l'exécution de requêtes EJB-QL.

```
Query query = entityManager.createQuery("SELECT user FROM User As user");
List<User> listUsers = query.getResultList();
```

```
Query query =
entityManager.createQuery("SELECT user.id, user.lastName FROM User As user");
List<Object[]> listUsers = query.getResultList();
for(Object[] valueArray : listUsers){
  Integer id = (Integer) valueArray[0];
  String name = (String) valueArray[1];
}
```

java.lang.Object getSingleResult()

- Cette méthode exécute la requête et retourne un unique résultat.

```
Query query = entityManager.createQuery("SELECT user FROM User AS user WHERE
  user.login = 'durand.dupont'");
User currentUser = (User) query.getSingleResult();
```

Query setMaxResults(int max) et Query setFirstResult(int first)

- Ces deux méthodes définissent respectivement le nombre maximal de résultats et l'index du premier élément à retourner.

```
Query query =
entityManager.createQuery("SELECT user FROM User As user");
query.setMaxResults(30);
query.setFirstResults(10);
List<User> listUsers = query.getResultList();
```

Query setParameter(String parameterName, Object value) et Query setParameter(int parameterIndex, Object value)

- Ces deux méthodes assignent la valeur value respectivement au paramètre **parameterName** ou au paramètre placé à l'index **parameterIndex**. Le nom du paramètre est défini dans la requête via « :nomDuParametre », ou via « ?numéro ».

```
SELECT user FROM User AS user WHERE user.id=?0
```

```
SELECT user FROM User AS user WHERE user.id=:userId
```

setParameter

```
Query query = entityManager.createQuery(
"SELECT user FROM User AS user WHERE user.id=?0");
query = query.setParameter(0, new Integer(5));
```

```
Query query = entityManager.createQuery(
"SELECT user FROM User AS user WHERE user.id=:userId");
query = query.setParameter("userId", new Integer(5));
```

int executeUpdate()

- Cette méthode exécute la requête qui doit être de type UPDATE ou DELETE.
- Elle retourne le nombre d'enregistrements supprimés ou modifiés.

```
Query query =
entityManager.createQuery("DELETE FROM User user WHERE user.login = '%s%'");
System.out.println(query.executeUpdate() + " enregistrement(s) supprimé(s)");
```

Les requêtes nommées

- Pour définir une requête nommée, il faut utiliser l'annotation @javax.persistence.NamedQuery.
- Cette annotation doit être placée au niveau de la classe de l'Entity Bean.

```
@NamedQuery (
  name="findAllUserWithName"
  query="SELECT user FROM User user WHERE user.lastName LIKE :lastname" )
@Entity
public class User { ... }
```

Les requêtes nommées

```
//...
@PersistenceContext
public EntityManager entityManager;

  public List<User> findUserByName(String name) {
    List<User> userList =
entityManager.createNamedQuery("findAllUserWithName").setParameter("lastname",na
me).getResultList();
    return userList;
  }
//...
```

Les requêtes natives

- La méthode EntityManager.createNativeQuery() permet de créer une requête native. Cette méthode retourne un objet Query comme les autres types de requêtes.
- Testé sur mysql 5:

```
Query query = em.createNativeQuery (
  "SELECT fp.* " +
  "FROM XUser user, Portfolio p, FinancialProduct fp " +
  "WHERE user.id=? AND user.id=p.userId " +
  "AND p.id=fp.portfolioId;"
);
query.setParameter(1, id);
Collection<FinancialProduct> collection = query.getResultList();
```

Des Questions ??