JPA

Java Persistence API

Laurent Broto

IRIT/ENSEEIHT

du 7 au 17 Février 2012

Le mapping Objet/Relationnel

Contexte

- Le modèle relationnel a fait ses preuves pour stocker des données
- Le modèle objet a fait ses preuves pour programmer

Question

- Quel est le lien entre les deux mondes

Réponses

- Systèmes de base de données orientées objet (O2) : ça n'a pas vraiment percé
- Autre approche : mapping objet/relationnel

Le mapping Objet/Relationnel

- Une classe correspond à une table dans une base de données
- Une instance de classe correspond à un tuple dans une base de données
- Plusieurs problèmes se posent :
  - les transactions
  - les propriétés objets (héritage, association, ...)
  - objets complexes (image, fichiers, ...)
- Un standard : **JPA** (Java Persistence API)
  - utilisation massive des annotations
  - utilisation de POJO/POJI : on garde la puissance objet
    - n'est pas sans poser de problème (héritage, association, ...)
  - anciennement faisant partie des EJB (de type *entity*)
- Occupe la partie **modèle** du schéma MVC

Présentation de JPA

- Une classe = Une table (en première approche)
- Une variable de classe = Une colonne de la table
  - sauf les champs `transient` ou annoté `@Transient`
- Mapping des types de classes avec les types disponibles dans la base de données
  - problème de portabilité : JDBC ne suffit pas
- Syntaxe
  - annotation : `@Entity` et `@Id`
    - la clé primaire
  - un constructeur vide obligatoire
  - des getters et des setters
    - instrumentation du code
- Une gestionnaire de persistence
  - permet de manipuler les Entity
  - utilisé dans un bean Session (pattern Facade) ou dans un client géré par le container

Un entity simple

```
@Entity
public class Etudiant
{
   @Id int numero;
   String nom;
}

@Stateless
public class GestionEtudiant
{
   @PersistenceContext
   private EntityManager em;

   public void addAStudent(int numero, String nom)  {
      Etudiant e=new Etudiant();
      e.numero=numero;
      e.nom=nom;
      em.persist(e);
   }
}
```

Dans la base de données

```
cours=> \d
           Liste des relations
 Schéma |   Nom    | Type  | Propriétaire
--------+----------+-------+--------------
 public | etudiant | table | cours
(1 ligne)

cours=> \d etudiant
            Table « public.etudiant »
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 numero  | integer                | non NULL
 nom     | character varying(255) |
Index :
    "etudiant_pkey" PRIMARY KEY, btree (numero)
```

La gestion des Entity

- Notion d'attachement
  - un bean est attaché quand on vient de le récupérer ou de le merger
  - jusqu'a la fin de la transaction courante (en première approche)
  - tant qu'on reste dans le container d'EJB
- Le contexte de persistance
  - contient l'ensemble des beans attachés (en première approche)
  - se comporte comme un *cache* vis à vis de la BD
- Se manipule grâce à l'**EntityManager** avec ses méthodes

persist	rends un POJO annoté `@Entity` persistant
merge	ré-attache un bean détaché
refresh	met à jour un bean à partir de la base de données
flush	synchronisation forcée du contexte et de la BD
remove	supprime un entity (du contexte et de la BD)

La gestion des Entity

find	trouve un bean dans la BD à partir par sa PK et le met dans le contexte
createQuery	créer une requete EJB-QL
createNativeQuery	créer une requete SQL
createNamedQuery	créer une requete nommée

Méthodes de **Query**

executeUpdate	exécute une requête update ou delete
getResultList	exécute un select et renvoi une List
getSingleResult	exécute un select qui renvoi un seul résultat
setParameter	assigne un paramètre nommé

Recherche d'un étudiant

```
@Stateless
public class GestionEtudiant
{
   @PersistenceContext
   private EntityManager em;

   ...

   public Etudiant findByPK(int numero)  {
      return (Etudiant)em.find(Etudiant.class, numero);
   }

   public Etudiant findByName(String name)  {
      return (Etudiant)em.createQuery("select e from
                  Etudiant where e.nom="+name).getSingleResult();
   }
}
```

Association entre beans

- Comment représenter l'association entre deux classes ?
  - *lier un étudiant à sa cité U*
  - *lier un étudiant à des cours*
- 3 possibilités en fonction des *cardinalités*
  - combien de beans peuvent se trouver de chaque côté de l'association
  - *1 étudiant est relié à 1 cité U*
  - *1 cité U est reliée à plusieurs étudiants*
  - *1 étudiant est relié à plusieurs cours*
  - *1 cours est relié à plusieurs étudiants*
- On retrouve les cardinalités UML
- Annotations dédiées

Association entre beans

Annotation	Cardinalité	Annote un(e)
`@OneToOne`	1-1	Object
`@OneToMany`	1-*	Collection<T>
`@ManyToOne`	*-1	Object
`@ManyToMany`	*-*	Collection<T>

- Stratégie de chargement d'une collection
  - à la demande (fetch=FetchType.LAZY) lors de l'accès à un champs
    - attention lors du détachement d'un bean !
  - à la volée (fetch=FetchType.EAGER) lors du chargement du bean
    - peut être du temps perdu si pas d'accès au champ
- Une Collection peut être
  - un **Set** ⇒ pas de doublons permis
  - une **List** (**ArrayList, . . .**) ⇒ un seul par bean si la stratégie de chargement est *EAGER*

Entity associés simple

```java
@Entity
public class CiteU
{
   @Id
   public String name;
}

@Entity
public class Cours
{
   @Id
   public String name;
}
```

Entity associés simple

```java
@Entity
public class Etudiant
{
   @Id int numero;
   String nom;

   @ManyToOne
   CiteU citeU;

   @ManyToMany
   List<Cours> cours;
}
```

Dans la base de données

```
             Liste des relations
 Schéma |      Nom       | Type  | Propriétaire
--------+----------------+-------+--------------
 public | citeu          | table | cours
 public | cours          | table | cours
 public | etudiant       | table | cours
 public | etudiant_cours | table | cours

             Table ≪ public.citeu ≫
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 name    | character varying(255) | non NULL
Index :
    "citeu_pkey" PRIMARY KEY, btree (name)
Référencé par :
    TABLE "etudiant" CONSTRAINT "fkc55d557c559b2832" FOREIGN KEY (citeu_name) REFERENCES citeu(name)

             Table ≪ public.cours ≫
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 name    | character varying(255) | non NULL
Index :
    "cours_pkey" PRIMARY KEY, btree (name)
Référencé par :
    TABLE "etudiant_cours" CONSTRAINT "fk42cf0ce7e360ffd2" FOREIGN KEY (cours_name) REFERENCES cours(name)
```

Dans la base de données

```
              Table ≪ public.etudiant ≫
  Colonne   |          Type          | Modificateurs
------------+------------------------+---------------
 numero     | integer                | non NULL
 nom        | character varying(255) |
 citeu_name | character varying(255) |
Index :
    "etudiant_pkey" PRIMARY KEY, btree (numero)
Contraintes de clés étrangères :
    "fkc55d557c559b2832" FOREIGN KEY (citeu_name) REFERENCES citeu(name)
Référencé par :
    TABLE "etudiant_cours" CONSTRAINT "fk42cf0ce7d2b8ee93" FOREIGN KEY (etudiant_numero)
          REFERENCES etudiant(numero)

             Table ≪ public.etudiant_cours ≫
     Colonne     |          Type          | Modificateurs
-----------------+------------------------+---------------
 etudiant_numero | integer                | non NULL
 cours_name      | character varying(255) | non NULL
Contraintes de clés étrangères :
    "fk42cf0ce7d2b8ee93" FOREIGN KEY (etudiant_numero) REFERENCES etudiant(numero)
    "fk42cf0ce7e360ffd2" FOREIGN KEY (cours_name) REFERENCES cours(name)
```

Association entre beans

- Navigabilité
  - explicite
  - paramètre `mappedBy`

```
@Entity
public class CiteU
{
   @Id   public String name;
   @OneToMany(mappedBy="citeU")
    public List<Etudiant> etudiants;
}

@Entity
public class Cours
{
   @Id   public String name;
   @ManyToMany(mappedBy="cours")
    public List<Etudiant> etudiants;
}
```

Héritage entre beans

- Comment représenter l'héritage entre deux classes ?
  - le modèle relationnel ne le permet pas de manière directe
- 3 possibilités dans la base de données
  1. Tout stocker dans une seule table en rajoutant des informations
     - les tables peuvent devenir très grosses
  2. Chaque bean a sa table et les tables se référencent par des jointures
     - les requêtes sont plus complexes
  3. Chaque bean a sa table, les tables ne contiennent que les beans associés et des informations sont rajoutées aux tables
     - mix des deux approches précédentes
- 3 paramètres de l'annotation `@Inheritance` dédiées
  1. strategy=InheritanceType.SINGLE_TABLE (stratégie par défaut)
  2. strategy=InheritanceType.JOINED
  3. strategy=InhreitanceType.TABLE_PER_CLASS

Héritage : SINGLE_TABLE

```java
@Entity
public class Personne
{
   @Id
   public String nom;
}

@Entity
public class Etudiant extends Personne
{
   int numero;
}
```

Dans la base de données

```
           Liste des relations
 Schéma |   Nom    | Type  | Propriétaire
--------+----------+-------+--------------
 public | personne | table | cours

            Table « public.personne »
 Colonne |          Type           | Modificateurs
---------+-------------------------+---------------
 dtype   | character varying(31)   | non NULL
 nom     | character varying(255) | non NULL
 numero  | integer                 |
Index :
    "personne_pkey" PRIMARY KEY, btree (nom)
```

Héritage : JOINED

```java
@Inheritance(strategy=InheritanceType.JOINED)
@Entity
public class Personne
{
   @Id
   public String nom;
}

@Entity
public class Etudiant extends Personne
{
   int numero;
}
```

Dans la base de données

```
          Liste des relations
 Schéma |   Nom    | Type  | Propriétaire
--------+----------+-------+--------------
 public | etudiant | table | cours
 public | personne | table | cours

            Table ≪ public.personne ≫
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 nom     | character varying(255) | non NULL
Index :
    "personne_pkey" PRIMARY KEY, btree (nom)
Référencé par :
    TABLE "etudiant" CONSTRAINT "fkc55d557c8f70db70" FOREIGN KEY (nom) REFERENCES personne(nom)

            Table ≪ public.etudiant ≫
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 numero  | integer                | non NULL
 nom     | character varying(255) | non NULL
Index :
    "etudiant_pkey" PRIMARY KEY, btree (nom)
Contraintes de clés étrangères :
    "fkc55d557c8f70db70" FOREIGN KEY (nom) REFERENCES personne(nom)
```

Héritage : TABLE_PER_CLASS

```java
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Entity
public class Personne
{
   @Id
   public String nom;
}

@Entity
public class Etudiant extends Personne
{
   int numero;
}
```

Dans la base de données

```
          Liste des relations
 Schéma |   Nom    | Type  | Propriétaire
--------+----------+-------+--------------
 public | etudiant | table | cours
 public | personne | table | cours

            Table ≪ public.personne ≫
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 nom     | character varying(255) | non NULL
Index :
    "personne_pkey" PRIMARY KEY, btree (nom)

            Table ≪ public.etudiant ≫
 Colonne |          Type          | Modificateurs
---------+------------------------+---------------
 nom     | character varying(255) | non NULL
 numero  | integer                | non NULL
Index :
    "etudiant_pkey" PRIMARY KEY, btree (nom)
```

Les transactions

- Toutes les méthodes des EJB ou des Entity sont *transactionnelles*
  - transaction au sens BD (**begin**,**commit**,**rollback**)
- Commitées par défaut sauf en cas d'Exception
- Les transactions respectent les propriétés ACID
  - **A**tomicité
    - toutes les méthodes se passent bien, ou aucune ne se fait
  - **C**onsistance
    - le contenu de la BD doit être cohérent à l'issue de la transaction
  - **I**solation
    - si deux transactions sont concurrentes, leur modification leur sont invisibles avant le commit
  - **D**urabilité
    - une transaction commitée ne peut être remise en cause
- Problème : gestion des appels de méthode cascadés

Exemple

```
@Stateless
public class ManipuleCompteBancaire
{
   public void virement(Compte src, Compte dst, int montant)
   {
      src.debite(montant);
      dst.credite(montant);
      log("transaction terminee");
   }

   public void log(String str)
   {
      log.addATuple(str);
   }
}
```

Les transactions

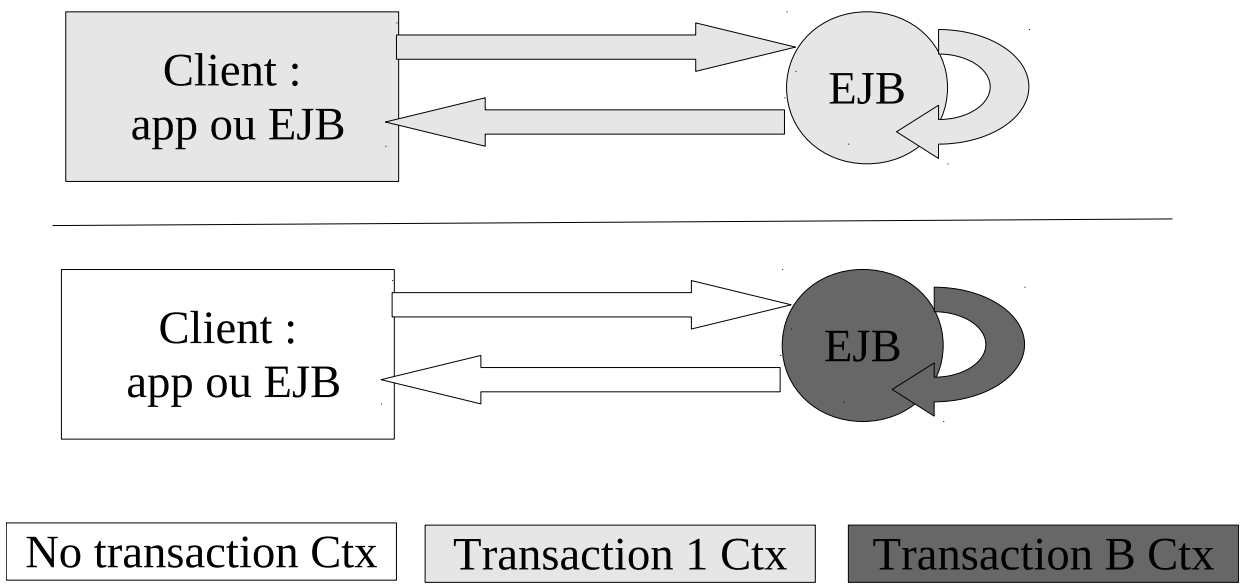
- **A**tomicité
  - le débit et le crédit doivent tous les deux marcher ou aucun
- **C**onsistance
  - si une contrainte dit que les compte doivent avoir un montant > 0, à l'issue de la transaction, le montant doit être > 0
- **I**solation
  - si une transaction consulte le montant du compte débité, elle doit voir soit le montant avant le début du virement soit le montant après
- **D**urabilité
  - les nouveaux montants doivent être enregistrés dans la base de données lors du commit

Les transactions

- Problème : si l'opération de log ne se passe pas bien, veut on réellement *rollbacker* la transaction ?
- On voudrait pouvoir spécifier de manière plus fine quelle méthode prends part ou non à la transaction courante
- On voudrait pour spécifier si le résultat d'une méthode est important ou non pour la transaction courante
- Deux manières de gérer les transactions
  - manuellement : tout est explicite
  - automatiquement : des annotations permettent de spécifier la fonction que prendra une méthode dans une transaction

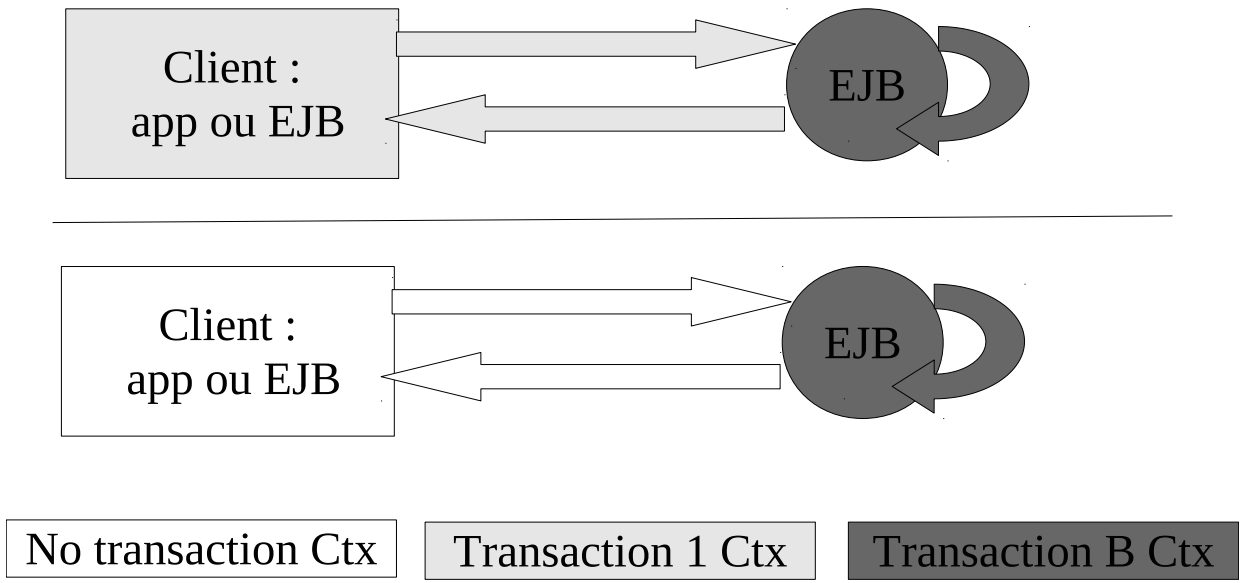
Les transactions : REQUIRED

@TransactionAttribute(TransactionAttributeType.REQUIRED)



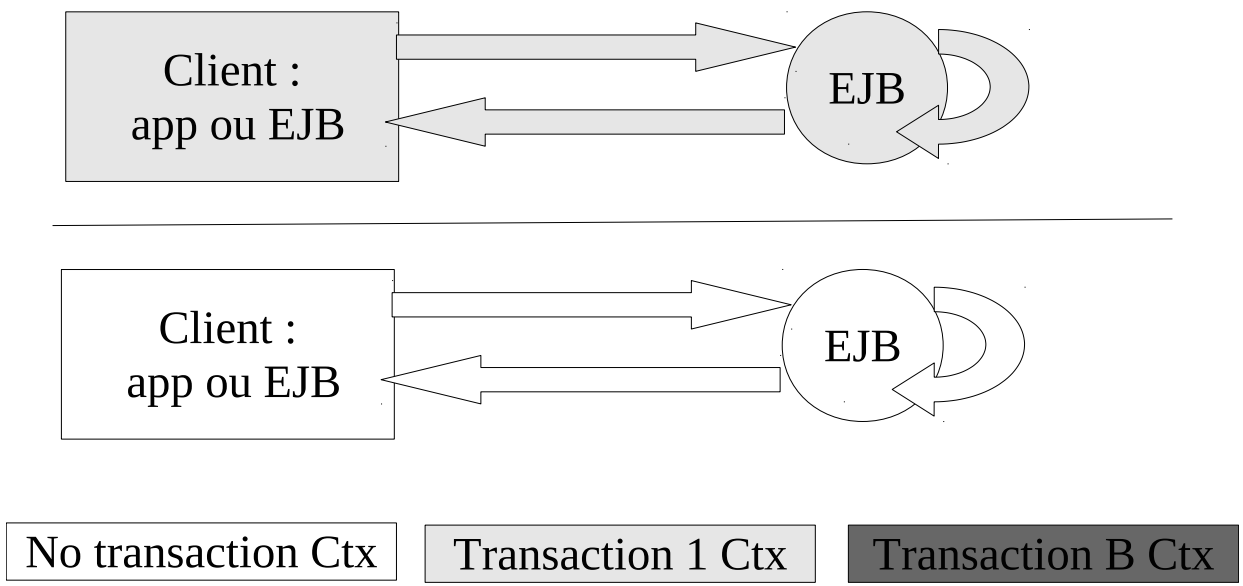
Les transactions : REQUIRES_NEW

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)



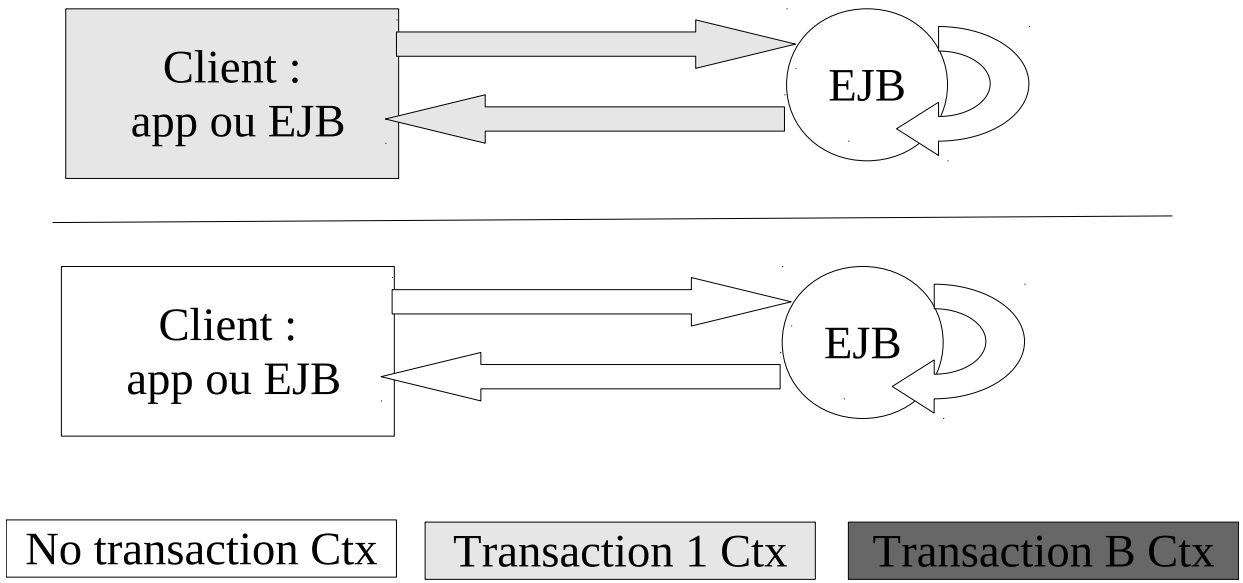
Les transactions : SUPPORTS

@TransactionAttribute(TransactionAttributeType.SUPPORTS)



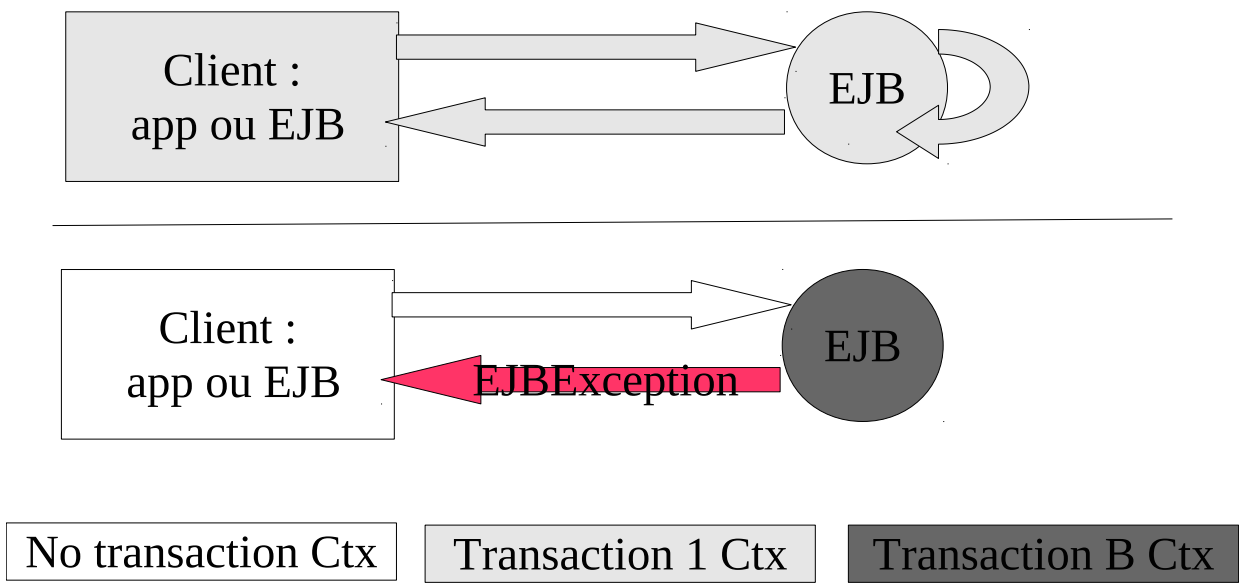
Les transactions : NOT_SUPPORTED

@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)



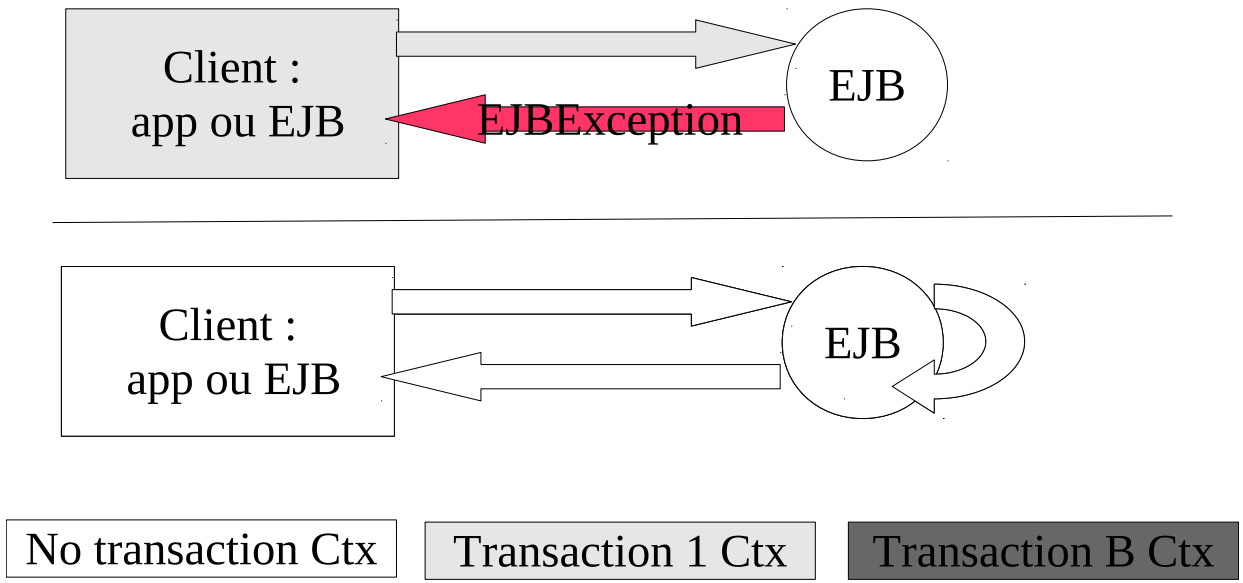
Les transactions : MANDATORY

@TransactionAttribute(TransactionAttributeType.MANDATORY)



Les transactions : NEVER

@TransactionAttribute(TransactionAttributeType.NEVER)



Exemple

```
@Stateless
public class ManipuleCompteBancaire
{
   public void virement(Compte src, Compte dst, int montant)
   {
      src.debite(montant);
      dst.credite(montant);
      log("transaction terminee");
   }

   @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
   public void log(String str)
   {
      log.addATuple(str);
   }
}
```

Les transactions

- À grain encore plus fin : gestion des transactions manuelles
  - BMT (Bean Managed Transaction) : API JTA (Java Transaction API)
- À l'aide du gestionnaire de transaction
  - une instance **UserTransaction**
  - obtenu par injection de dépendance (`@Ressource`)

begin()	début de transaction
commit()	validation de la transaction
rollback()	annulation de la transaction

Exemple

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class ManipuleCompteBancaire
{
  @Resource private UserTransaction ut;
  public void virement(Compte src, Compte dst, int montant)
  {
    ut.begin();
    try {
      src.debite(montant);
      dst.credite(montant);
      ut.commit();
    }
    catch(Exception e) {
      ut.rollback();
    }
    finally  {
      log("transaction terminee");
    }
  }

  public void log(String str)
  {
    log.addATuple(str);
  }
}
```

Les transactions

- Niveaux d'isolation couramment définis
  - ReadUncommitted
    - peut lire des modifications non validées (dirty read)
  - ReadCommitted
    - ne peut lire que des modifications validées
  - RepeatableRead
    - Une transaction qui lit plusieurs fois lit la même valeur
  - Serializable
    - Lectures et écritures exclusives, le plus restrictif
- TRANSACTION_READ_COMMITTED est adéquate pour la plupart des applications
- TRANSACTION_SERIALIZABLE est trop pénalisant (performances)

Le pattern Facade

- Objectif : simplification de l'interface des entity pour fournir une interface simple au client
- Association d'un EJB Stateless, la **Facade**, à un ou plusieurs Entity
- Les méthodes métiers applicables à l'entity sont encapsulées dans le bean Facade
- Limite la bande utilisée $\Rightarrow$ traitements plus rapides
- Par exemple :
  - sélection des Entity dans un ResultSet : gain de performance
  - ajout d'un étudiant en passant en paramètre sa cité U et un tableau de cours : simplification de l'interface

JBOSS

Mise en œuvre avec JBoss

- Définir un fichier data-source. Il doit se terminer par `-ds.xml`
  - ce fichier définit une base de données, user, mot de passe et driver JDBC
  - il doit se trouver dans le répertoire $JBOSS_HOME/server/<server>/deploy
- La base de données doit être préalablement créée
- La data source utilisée pour les EJB est définie dans le fichier **persistence.xml**
  - Plusieurs data-source peuvent être définie dans le fichier persistence.xml
- Mettre le driver JDBC dans $JBOSS_HOME/server/<server>/deploy

Exemple de fichier data-source

```
<datasources>
  <local-tx-datasource>
    <jndi-name>cours</jndi-name>

    <connection-url>
      jdbc:postgresql://localhost:5432/cours
    </connection-url>

    <driver-class>
      org.postgresql.Driver
    </driver-class>

    <user-name>cours</user-name>
    <password>cours</password>

    <connection-property name="autoReconnect">
      true
    </connection-property>

  </local-tx-datasource>
</datasources>
```

Exemple de fichier persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
          http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
        version="1.0">
        <persistence-unit name="cours" transaction-type="JTA">
                <provider>org.hibernate.ejb.HibernatePersistence</provider>
                <jta-data-source>java:/cours</jta-data-source>
                <properties>
                        <property name="hibernate.hbm2ddl.auto" value="update"/>
                </properties>
        </persistence-unit>
</persistence>
```

Packaging

2 méthodes : avant EJB 3.1 et après

- Ancienne méthode ($<$ Java EE 6)
  - les Servlets, pages HTML et JSP dans un fichier WAR
  - les EJB et les entities JPA dans un fichier JAR
    - le fichier *persistence.xml* sera dans le répertoire META-INF de ce JAR
  - le tout dans un fichier EAR
- Méthode EJB3.1
  - tout est dans le fichier WAR
    - le fichier *persistence.xml* sera dans le répertoire WEB-INF de ce WAR
    - le fichier *persistence.xml* devra contenir la liste des entities JPA à déployer $\Rightarrow$ `<class>entity.MyEntity</class>` dans la balise `persistence-unit`

Conclusion

- JPA permet d'abstraire la base de données
- Des annotations sont introduites pour décrire les liens entre les beans au sein des POJO
  - liens horizontaux (relations n-m)
  - liens verticaux (héritages)
- Les relations étaient naturellement présentes dans le SGBD relationnel, l'héritage a été rajouté
  - aucun n'est complètement satisfaisant
- Le requêtage peut se faire maintenant en SQL natif
  - cela introduit des problèmes de mapping entre les tables et les beans
- Le packaging a été simplifié grâce à l'utilisation unique du fichier WAR
  - le programmeur doit spécifier manuellement les entities présents dans ce fichier