

Les applications n-tiers

La couche Persistance

Introduction : Java Persistence API

Les entités JPA

Les relations entre entités

Introduction

Les objets métier sont souvent enregistrés :

- Dans une base de données relationnelle
 - Au travers de JDBC par exemple (module 4)
- Parfois dans de simples fichiers
 - A travers de JAXB par exemple pour les fichiers XML

On parle de “persistance” de ces objets.

Introduction

Les différents SGBDR sont incompatibles entre eux en termes d'organisation des données (représentation des données dans des fichiers de format spécifique), mais ils parlent tous SQL – ou plus précisément un **dialecte SQL** (syntaxe commune SQL plus des extensions spécifiques).

Pour rendre l'accès à une base de données relationnelle indépendant du SGBDR qui la représente, il a été défini dans le monde Java l'API **JDBC**.

Introduction

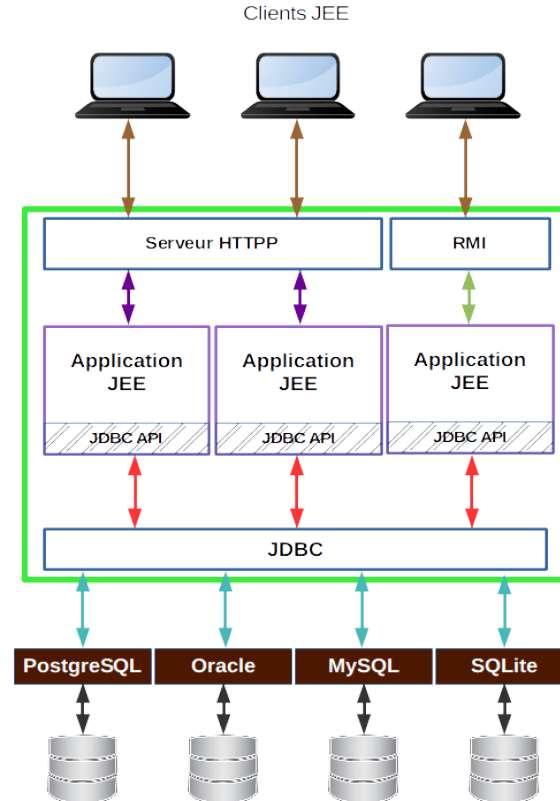
JDBC est un moyen de réaliser de manière **à peu près** indépendante d'un SGBDR l'accès aux données stockées.

Néanmoins, si l'utilisation de JDBC est faite dans toutes les classes utilisant des données persistées, il y a création d'un couplage fort entre toutes ces classes et des tables relationnelles – voire un SGBDR précis.

Le pattern DAO (**Data Access Object**) vise à limiter ce couplage en définissant des objets par lesquels tous les accès aux données se feront.

Ce pattern permet d'ailleurs d'encapsuler les accès à **d'autres datasources** (fichiers plain, fichiers XML, bases de données orientées objet, etc.).

L'accès aux données façon JDBC



ORM et entités

Il existe une autre manière de réaliser les accès aux données stockées : en établissant une correspondance directe entre le monde relationnel et le monde Java.

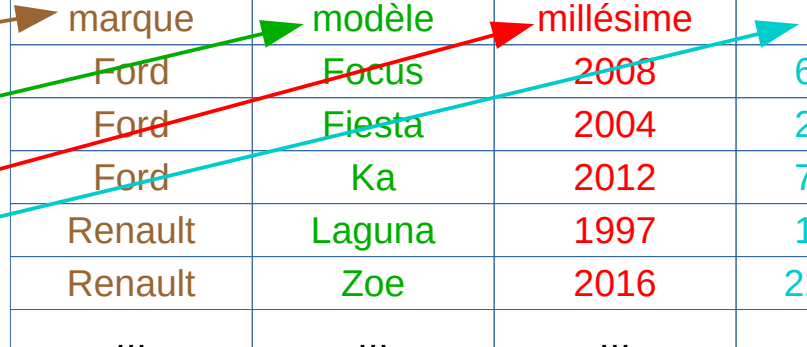
Dans cette manière de faire, à une **classe** est associée une **table** dans la base de données, et à chaque **attribut** de la classe est associé un **champ** dans cette table.

Cette correspondance est appelée ORM (Object-Relational Mapping).

En Java, une classe persistante est appelée une **entité**.

Exemple de mapping objet/relationnel

```
class Voiture {  
  private String marque;  
  private String modele;  
  private Date millésime;  
  float prix;  
}
```



marque	modèle	millésime	prix
Ford	Focus	2008	6 000 €
Ford	Fiesta	2004	2 500 €
Ford	Ka	2012	7 500 €
Renault	Laguna	1997	1 000 €
Renault	Zoe	2016	22 000 €
...

Table Voiture

Les EJB Entités

Avant la version 5 de JEE (version 3.0 des EJB), la gestion de la persistance se faisait sans mapping objet/relationnel, au travers d'EJB appelés Entités (**Entity Beans**).

Ces EJB utilisaient généralement JDBC, soit via du code généré par le conteneur d'EJB en se basant sur un fichier XML (*deployment descriptor*), soit directement via le code du composant :

- Lourd
- Complexe

Java Persistence API

Avec JEE 5 (vers 2006) est apparue une API standard pour réaliser ce mapping : **Java Persistence API** (JPA).

Cette API décrit :

- Des **annotations** pour faire le lien entre les objets Java (classe, attributs de classe) et les objets relationnels (tables et champs de table)
- Un **langage de haut niveau** (pseudo SQL) permettant de réaliser des requêtes vers la base de données : Java Persistence Query Language (JPQL)
- Une **API de requêtage** similaire à JPQL (Criteria API), mais basée sur des API Java.

Java Persistence API

La persistance JPA se base sur JDBC, mais ceci devient invisible du développeur :

- On n'utilise pas l'API JDBC en direct, on se contente de définir une datasource JDBC

NB. Une classe persistante peut être définie en dehors de JEE, comme un simple POJO (Plain Old Java Object).

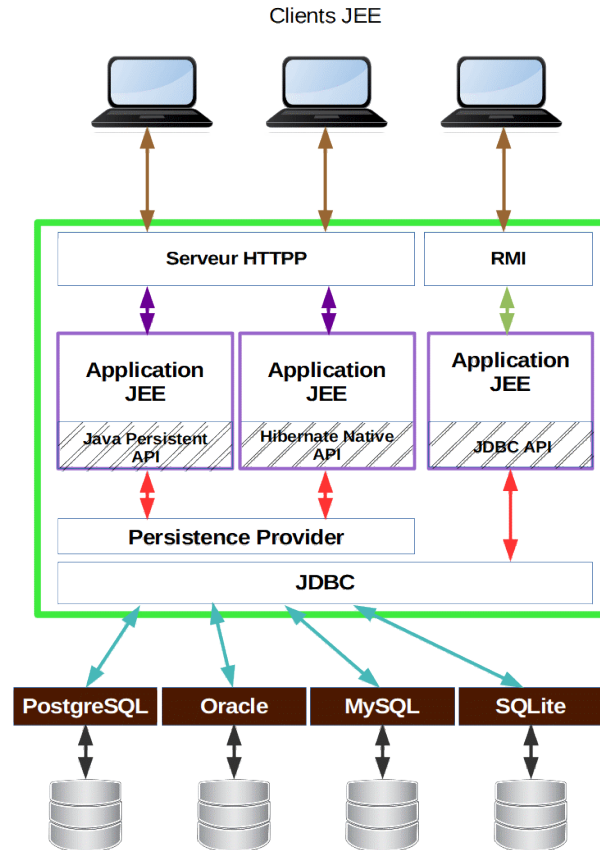
Les Persistence Providers

L'outil implémentant l'API JPA (et peut-être plus) est appelé un **Persistence Provider**.

Il existe différentes implémentations :

- EclipseLink, l'implémentation dite “de référence” car fournie par Oracle
- Hibernate, très utilisée et fournissant des services additionnels à JPA
- Apache Open JPA
- ...

L'accès aux données vu par JPA



Principales annotations JPA

La version 2.1 de JPA définit 89 annotations différentes.



Nous allons utiliser les principales :

`@Entity` ==> obligatoire, définit une classe Java comme persistante

`@Id` ==> obligatoire, définit un attribut de classe comme la clé primaire de la table

`@Column` ==> utilisé pour préciser les caractéristiques d'un champ

`@Temporal` ==> utilisé pour définir un champ Date

`@Enumerated` ==> indique comment un attribut de type énuméré doit être mappé sur un champ de la table (par chaîne de caractères ou par valeur numérique)

Principales annotations JPA

`@OneToOne` ==> utilisé pour indiquer une jointure 1-1 entre tables

`@OneToMany` ==> utilisé pour indiquer une jointure 1-n entre tables

`@ManyToOne` ==> jointure n-1

`@ManyToMany` ==> jointure n-p

`@Transient` ==> indique un attribut non persisté

Les entités JPA

Pour être acceptée comme Entité, une classe Java doit respecter les contraintes suivantes :

- Porter l'annotation `@Entity`
- Fournir un constructeur par défaut (i.e. sans argument)
- Implémenter l'interface `Serializable` (si l'entité doit être sérialisée, par exemple pour être envoyée sur une autre machine)
- La classe, ses méthodes et ses attributs persistants ne doivent pas être marqués `final`

Les entités JPA

Un attribut persisté peut avoir l'un des types Java suivants :

- Type primitif
- `String`
- Wrapper de type primitif (`Integer`, `Long`, `Double`, `Float`, ...)
- `BigInteger`, `BigDecimal`, `java.util.Date`, `java.sql.Date`, `java.sql.Time`, ...
- Tableau d'octets (`byte`) ou de caractères
- Type défini par l'utilisateur (si sérialisable)
- Autre type d'entité
- Type `Embeddable`
- Collection Java.

Les entités JPA

Une instance de type `Embeddable` n'est pas une entité mais peut faire partie d'une entité.

Exemple : une adresse postale.

```
@Embeddable  
  
public class AdressePostale {  
    String rueEtNumero;  
  
    String ville;  
  
    String codePostal;  
  
    String pays;  
}
```

Les entités JPA

Une entité peut contenir plusieurs attributs de type `Embeddable` ou plusieurs collections de tels attributs.

```
@Entity
public class Client {
    @Id
    protected long id;
    String nom;
    int numero;
    @Embedded
    AdressePostale adresse;
}
```

Clé primaire d'entité JPA

Toute entité doit avoir une clé primaire unique, mais cette clé peut être simple ou composite (formée de plusieurs attributs).

Une clé primaire simple doit appartenir à l'un des types suivants :

- Types primitifs et `String`
- Wrappers de type primitif (`Integer`, `Long`, `Double`, `Float`, ...)
- `BigInteger` **ou** `BigDecimal`
- `java.util.Date` **ou** `java.sql.Date`

Clé primaire d'entité JPA

Une clé primaire composite doit être composée d'attributs de la classe ou d'un attribut d'un type marqué comme `Embeddable`.

La Persistence Unit

Dans quelle base de données seront stockées les informations ? Autrement dit, comment est fait le lien entre les entités Java et la base de données ?

Réponse : au travers d'une **Persistence Unit** (PU).

La PU associée à un ensemble d'entités est définie dans un fichier `persistence.xml`.

La Persistence Unit

Le fichier descriptif d'une **Persistence Unit** (PU) permet de spécifier (entre autres) :

- Le contexte transactionnel de l'application
- Le Persistence Provider utilisé
- Les entités faisant partie de la PU
- La datasource utilisée
- Les opérations à réaliser dans la base de données au moment du démarrage/déploiement de l'application et au moment de sa fermeture

La Persistence Unit

Il existe plusieurs formes descriptives d'une Persistence Unit, selon que l'application est une application JEE (donc gérée par un container d'EJB ou un container web) ou une simple application Java.

`persistence.xml` non JEE

Cas d'un fichier `persistence.xml` pour une **application non-JEE**.

persistence.xml non JEE

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.1">
```

```
  <persistence-unit name="ProjectPU" transaction-  
type="RESOURCE_LOCAL">
```

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</  
provider>
```

```
  <class>project.Entite1</class>
```

```
  <class>project.Entite2</class>
```



Nom de la PU

persistence.xml non JEE

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.1">
```

```
  <persistence-unit name="ProjectPU" transaction-  
type="RESOURCE_LOCAL">
```

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</  
provider>
```

```
  <class>project.Entite1</class>
```

```
  <class>project.Entite2</class>
```



Contexte
transactionnel

persistence.xml non JEE

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.1">
```

```
  <persistence-unit name="ProjectPU" transaction-  
type="RESOURCE_LOCAL">
```

```
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</  
provider>
```

```
      <class>project.Entite1</class>
```

```
      <class>project.Entite2</class>
```



Persistence
provider

persistence.xml non JEE

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.1">
```

```
  <persistence-unit name="ProjectPU" transaction-  
type="RESOURCE_LOCAL">
```

```
<provider>org.eclipse.persistence.jpa.PersistenceProvider</  
provider>
```

```
  <class>project.Entite1</class>
```

```
  <class>project.Entite2</class>
```



Entités gérées

persistence.xml non JEE

```
<properties>

  <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/la_base?zeroDateTimeBehavior=convertToNull"/>

  <property name="javax.persistence.jdbc.user" value="toto"/>

  <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>

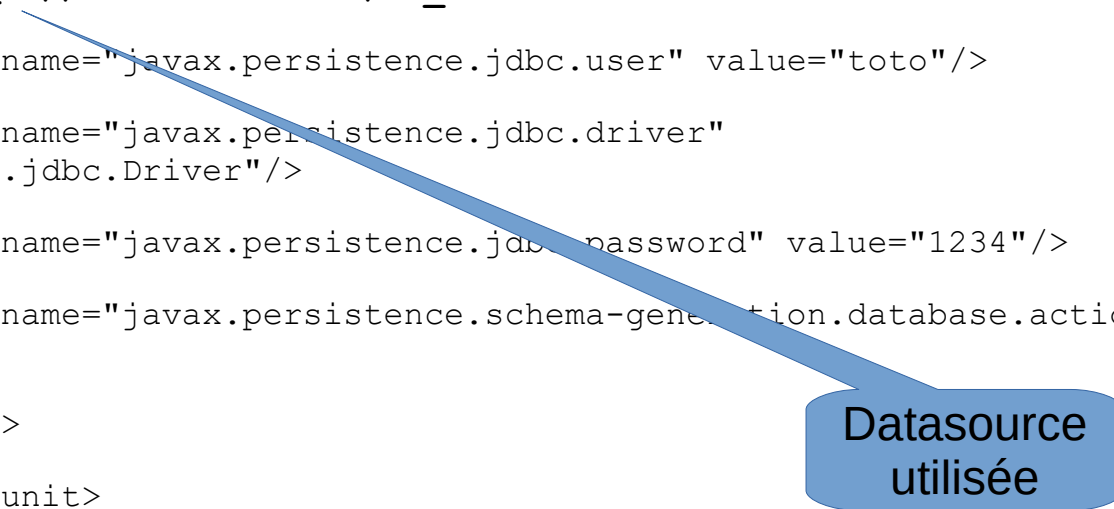
  <property name="javax.persistence.jdbc.password" value="1234"/>

  <property name="javax.persistence.schema-generation.database.action"
value="create"/>

</properties>

</persistence-unit>

</persistence>
```



Datasource
utilisée

persistence.xml non JEE

```
<properties>

  <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/la_base?zeroDateTimeBehavior=convertToNull"/>

  <property name="javax.persistence.jdbc.user" value="toto"/>

  <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>

  <property name="javax.persistence.jdbc.password" value="1234"/>

  <property name="javax.persistence.schema-generation.database.action"
value="create"/>

</properties>

</persistence-unit>

</persistence>
```



Compte utilisé

persistence.xml non JEE

```
<properties>

  <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/la_base?zeroDateTimeBehavior=convertToNull"/>

  <property name="javax.persistence.jdbc.user" value="toto"/>

  <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>

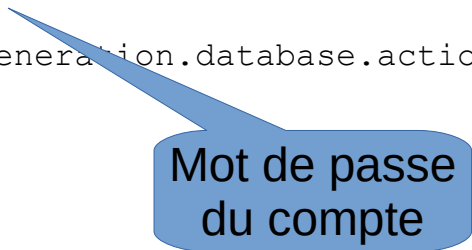
  <property name="javax.persistence.jdbc.password" value="1234"/>

  <property name="javax.persistence.schema-generation.database.action"
value="create"/>

</properties>

</persistence-unit>

</persistence>
```



Mot de passe
du compte

persistence.xml non JEE

```
<properties>

  <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/la_base?zeroDateTimeBehavior=convertToNull"/>

  <property name="javax.persistence.jdbc.user" value="toto"/>

  <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>

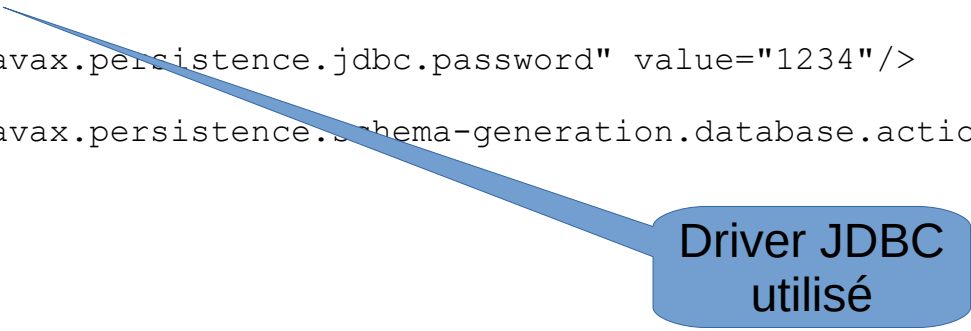
  <property name="javax.persistence.jdbc.password" value="1234"/>

  <property name="javax.persistence.schema-generation.database.action"
value="create"/>

</properties>

</persistence-unit>

</persistence>
```



Driver JDBC
utilisé

persistence.xml non JEE

```
<properties>

  <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/la_base?zeroDateTimeBehavior=convertToNull"/>

  <property name="javax.persistence.jdbc.user" value="toto"/>

  <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>

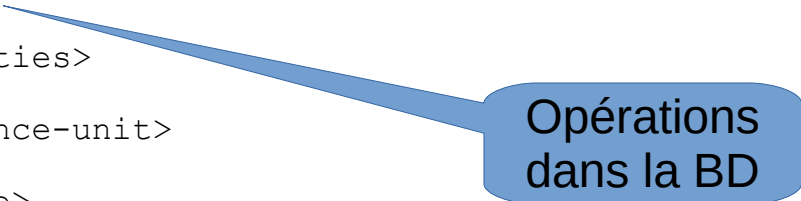
  <property name="javax.persistence.jdbc.password" value="1234"/>

  <property name="javax.persistence.schema-generation.database.action"
value="create"/>

</properties>

</persistence-unit>

</persistence>
```



Opérations
dans la BD

`persistence.xml` JEE

Cas d'un fichier `persistence.xml` pour une **application JEE**.

persistence.xml JEE

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.1">
```

```
  <persistence-unit name="AnotherProjectPU" transaction-type="JTA">
```

```
    <jta-data-source>jdbc/MySqlDS</jta-data-source>
```

```
    <exclude-unlisted-classes>>false</exclude-unlisted-classes>
```

```
    <properties>
```

```
      <property name="javax.persistence.schema-  
generation.database.action" value="drop-and-create"/>
```

```
    </properties>
```

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

```
</persistence>
```

persistence.xml JEE

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.1">
```

```
  <persistence-unit name="AnotherProjectPU" transaction-type="JTA">
```

```
    <jta-data-source>jdbc/MySqlDS</jta-data-source>
```

```
    <exclude-unlisted-classes>>false</exclude-unlisted-classes>
```

```
    <properties>
```

```
      <property name="javax.persistence.schema-  
generation.database.action" value="drop-and-create"/>
```

```
    </properties>
```

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

```
</persistence>
```



PU name


persistence.xml JEE

Contexte
transactionnel

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1">
  <persistence-unit name="AnotherProjectPU" transaction-type="JTA">
    <jta-data-source>jdbc/MySqlDS</jta-data-source>
    <exclude-unlisted-classes>>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```

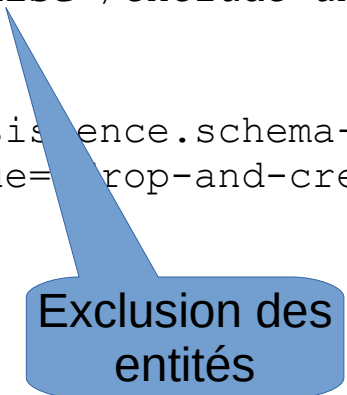
persistence.xml JEE

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1">
  <persistence-unit name="AnotherProjectPU" transaction-type="JTA">
    <jta-data-source>jdbc/MySQLDS</jta-data-source>
    <exclude-unlisted-classes>>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```



persistence.xml JEE

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1">
  <persistence-unit name="AnotherProjectPU" transaction-type="JTA">
    <jta-data-source>jdbc/MySQLDS</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```



Exclusion des entités

persistence.xml JEE

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1">
  <persistence-unit name="AnotherProjectPU" transaction-type="JTA">
    <jta-data-source>jdbc/MySQLDS</jta-data-source>
    <exclude-unlisted-classes>>false</exclude-unlisted-classes>
    <properties>
      <property name="javax.persistence.schema-
generation.database.action" value="drop-and-create"/>
    </properties>
  </persistence-unit>
</persistence>
```



Exclusion des
entités

L'Entity Manager

Comment est réalisé le mapping entre les entités Java et les objets dans la base de données (tables et champs) ?

Réponse : au moyen d'un `EntityManager`.

L'Entity Manager

Un `EntityManager` est un objet à qui est associé un “cache” de la base de données : c’est le point de passage entre une application JPA et une base de données.

Le cache ou **Persistence Context** (PC) est rempli d’objets Java :

- soit par lecture de données depuis la base de données
- soit par écriture d’entités depuis l’application.

L'Entity Manager

Principales méthodes de l'EntityManager	Résultat
<code>persist()</code>	Insère une entité dans le PC
<code>remove()</code>	Efface une entité du PC
<code>merge()</code>	Copie une entité dans le PC depuis l'application Java
<code>refresh()</code>	Rafraîchit une entité dans le PC depuis la BD
<code>getTransaction()</code>	Retourne l'EntityTransaction associée au PC
<code>find()</code>	Cherche une entité dans la BD et la place dans le PC
<code>flush()</code>	Ecrit les entités du PC dans la BD

L'Entity Manager

Principales méthodes de l'EntityManager	Résultat
<code>clear()</code>	Vide le PC (toutes les entités deviennent détachées)
<code>close()</code>	Ferme un PC (seulement si géré par l'application)
<code>createNamedQuery()</code>	Crée une Query pour exécuter du JPQL ou du SQL
<code>createNativeQuery()</code>	Crée une Query pour exécuter du SQL
<code>createQuery()</code>	Crée une TypedQuery pour exécuter une requête Criteria
<code>detach()</code>	Détache une entité dans le PC

Les états d'une entité JPA

Une entité JPA peut être dans différents états selon qu'elle est placée dans le Persistence Context ou pas :

- **new** : l'entité vient d'être créée (au sens Java), elle n'est pas connue du PC
- **managed** : l'entité est connue du PC, soit parce qu'elle a été persistée via l'`EntityManager`, soit parce qu'elle a été récupérée de la base de données par une requête :

```
EntityManager em;  
em.persist(entity);
```

Les états d'une entité JPA

- **removed** : l'entité précédemment "managed" a été effacée via son `EntityManager`, elle sera supprimée de la base au plus tard lorsque la transaction en cours sera committée, ou lorsque l'opération `flush()` sera invoquée sur l'`EntityManager` :

```
em.remove(entity);
```

- **detached** : l'entité a été retirée du PC via l'opération `clear()` sur l'`EntityManager`, ou via la fermeture de cet `EntityManager` :

```
em.clear();
```

Modification d'une entité JPA

Toute modification d'une entité JPA “managed” est prise en compte par l'`EntityManager`, qui enregistrera l'état final de l'entité dans la base de données au plus tard :

- lorsque la transaction en cours se terminera,
- ou lorsqu'une opération `flush()` sera effectuée sur cet `EntityManager`.



NB. Cet enregistrement n'est **pas** déclenché par une opération `em.persist()`.

Modification d'une entité JPA

Toute modification d'une entité JPA “detached” n'est **pas** prise en compte par l'`EntityManager`.

Synchronisation entités / BD

La base de données est mise en synchronisation avec les entités JPA “managed” d’un PC lors :

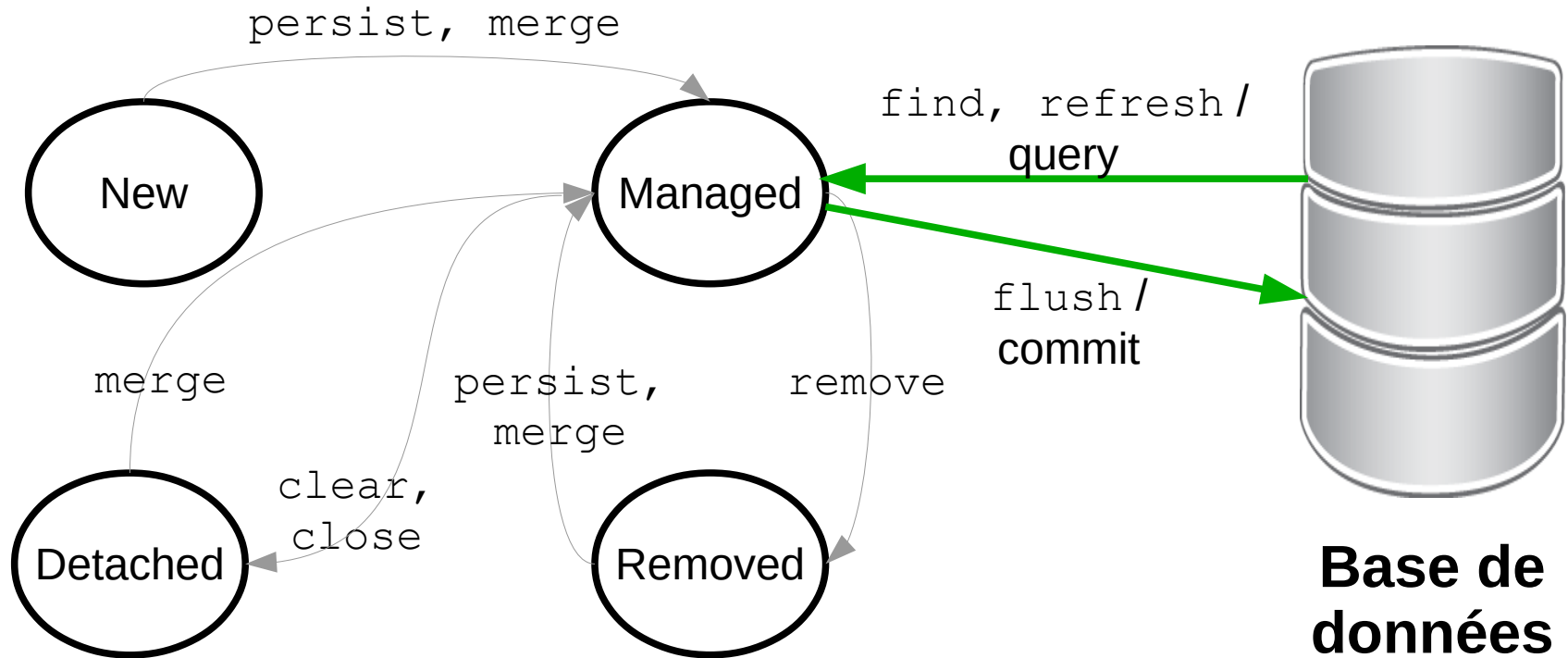
- du commit de la transaction en cours
- ou d’une opération `flush()` effectuée sur l’`EntityManager` gérant le PC.

Synchronisation entités / BD

Les entités “managed” d’un PC sont rafraichies par rapport à la BD lorsque l’opération `refresh()` est effectuée sur l’`EntityManager` gérant le PC.

Les entités associées à une entité rafraichie sont également rafraichie si la relation est marquée comme **cascadante** pour le rafraichissement.

Les états d'une entité JPA : synthèse



L'Entity Manager

Un `EntityManager` est géré selon deux modalités différentes :

- soit il est géré automatiquement par un container JEE (container d'EJB ou container web) ==> on parle de **container-managed entity manager**
- soit il est géré par l'application ==> on parle d'**application-managed entity manager**.

Evidemment, une application standalone (non JEE) doit gérer elle-même ses `EntityManager` puisque par définition elle n'est elle-même pas gérée par un container...

Types d'Entity Manager

La distinction entre container-managed EntityManager et application-managed EntityManager.

Type d'application / type d'entity manager	Application JEE	Application standalone
Container-managed EM	Standard	
Application-managed EM	Possible	Obligatoire

Entity Manager Factory

Un `EntityManager` est créé au moyen d'une **fabrique** d'EM :
une `EntityManagerFactory`.

Dans le cas d'un application-managed EM, la création de
l'`EntityManager` est faite par l'application.

Dans le cas d'un container-managed EM, la création de
l'`EntityManager` est faite par le container.

Une `EntityManagerFactory` est associée à une `Persistence Unit`, alors qu'un `EntityManager` est associé à un `Persistence Context`.

Entity Manager / Entity Manager Factory

Distinction entre `EntityManager` et `EntityManagerFactory`.

<i>Concept / Propriétés</i>	EntityManager	EntityManagerFactory
Concept de persistance	Persistence Context	Persistence Unit
Lien avec base de données	Une connexion	Fabrique de connexions

Entity Manager / Entity Manager Factory

Distinction entre `EntityManager` et `EntityManagerFactory`.

<i>Concept / Propriétés</i>	<code>EntityManager</code>	<code>EntityManagerFactory</code>
Container-managed EM dans application JEE	Injecté ou obtenu par JNDI	
Application-managed EM dans application JEE	Créé via l'EMF	Injectée ou obtenue par JNDI
Application-managed EM dans application standalone	Créé via l'EMF	Créée par utilisation de la classe <code>Persistence</code>

Accès aux EM et EMF

L'accès à un `EntityManager` ou à une `EntityManagerFactory` peut se faire de deux manières différentes :

- par **injection** → CDI (Context and Dependency Injection)
- par **consultation d'annuaire** → JNDI (Java Naming and Directory Interface)

L'injection est réalisée par un conteneur, donc dans le contexte JEE.

La consultation d'annuaire peut être faite dans le contexte JEE ou pas ==> mécanisme obligatoire pour une application SE.

Accès à un Entity Manager

L'injection d'un container-managed `EntityManager` se fait ainsi.

```
@PersistenceContext(unitName="ProjectPU")  
EntityManager em;
```

La chaîne spécifiée par `unitName` est le nom de la Persistence Unit indiquée dans le fichier `persistence.xml`.

Accès à un Entity Manager

L'obtention d'un container-managed `EntityManager` par JNDI se fait ainsi.

```
@PersistenceContext(name="MyEM")

public class MySessionBean implements MyInterface {

    @Resource SessionContext ctx;

    public void doSomething() {

        EntityManager em =

            (EntityManager) ctx.lookup("MyEM");

        ...
    }
}
```

Entity Manager

Toutes les instances d'`EntityManager` créées à partir d'une `EntityManagerFactory` sont configurées de la même manière.

En particulier, elles se connectent toutes à la même base de données.

Injection d'une Entity Manager Factory

L'injection d'une `EntityManagerFactory` dans un container JEE se fait ainsi.

```
@PersistenceUnit(unitName="ProjectPU")  
EntityManagerFactory emf;
```

La chaine spécifiée par `unitName` est le nom de la Persistence Unit indiquée dans le fichier `persistence.xml`.

Création d'une Entity Manager Factory

La création d'une `EntityManagerFactory` et subséquemment d'un `EntityManager` dans une application standalone se fait ainsi.

```
EntityManagerFactory emf =  
    javax.persistence.Persistence.createEntityManagerFa  
        ctory("ProjectPU");  
  
EntityManager em = emf.createEntityManager();
```

Entity Manager et transactions

Les opérations dans le Persistence Context doivent se faire au travers d'une **transaction** pour que l'état des entités puisse être sauvegardé dans la base de données.

Cette sauvegarde est faite lorsque la transaction est committée, ou lorsque on demande à l'EntityManager de flusher le PC.

Les transactions associées à un `EntityManager` peuvent être gérées de deux manières différentes : soit par **JTA**, soit par **l'application elle-même**.

Entity Manager et transactions

“Depending on the transactional type of the entity manager, transactions involving EntityManager operations may be controlled either through JTA or through use of the resource-local `EntityManagerTransaction` API, which is mapped to a resource transaction over the resource that underlies the entities managed by the entity manager.”

Spécification de JPA version 2.1.

Entity Manager et transactions

Un `EntityManager` dont les transactions sous-jacentes sont contrôlées par JTA est appelé un “**JTA entity manager**”.

Un `EntityManager` dont les transactions sous-jacentes sont contrôlées directement par l'application (au travers d'une interface `EntityTransaction`) est appelé un “**resource-local entity manager**”.

Entity Manager et transactions

La distinction entre container-managed EntityManager et application-managed EntityManager.

<i>Contrôle de transaction selon le type d'entity manager</i>	JTA Entity Manager	RL Entity Manager
Container-managed EM	Obligatoire	
Application-managed EM	Possible	Possible

Entity Manager et transactions

Un exemple simple : un programme Java standalone permettant de modifier le mot de passe d'un compte dont l'identifiant et le mot de passe sont passés en arguments sur la ligne de commande.

C'est une application JPA, un `EntityManager` est créé à partir d'une `EntityManagerFactory`.

La transaction définie dans le Persistence Context associé à l'`EntityManager` est récupérée à partir de l'`EntityManager`.

Elle sert à réaliser la modification dans la base de données.

Entity Manager et transactions

```
import javax.persistence.*;

public class PasswordChanger {

    public static void main (String[] args) {

        EntityManagerFactory emf =

            Persistence.createEntityManagerFactory("ExemplePU");

        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin() ;
    }
}
```

Entity Manager et transactions

```
User user = (User)em.createQuery("SELECT u FROM User  
u WHERE u.name=:name AND u.pass=:pass")  
  
.setParameter("name", args[0])  
  
.setParameter("pass", args[1])  
  
.getSingleResult();
```

Entity Manager et transactions

```
if (user != null) {  
    user.setPassword(args[2]);  
}  
  
em.getTransaction().commit();  
  
em.close();  
  
emf.close();  
}  
}
```

La couche Persistance

Réaliser l'exercice 1 (entité simple).

La couche Persistance

Lancer l'exécution du test (`Run file` sur la classe `CategorieEntityTest`) plusieurs fois, puis regarder le contenu de la table MySQL `categorieentity`, par exemple à partir de MySQL Workbench.

Il n'y a pas un problème ?

Comment pourrait-on le régler ?

QUIZZ

Unicité de valeur d'attribut

JPA propose le paramètre booléen `unique` sur l'annotation `@Column` :

```
@Column (unique = true)
```

Il se place bien sûr sur l'attribut dont la valeur doit être unique dans la table.

La couche Persistance

Réaliser l'exercice 2 (entité simple avec attribut unique).

JPA et l'héritage

La notion d'héritage de type est supportée par JPA.

Une entité peut étendre une classe qui est une entité ou pas, et une entité peut être la classe-mère d'une classe qui est une entité ou pas.

JPA et l'héritage - exemple

Classe de base :

```
@Entity  
public abstract class Employee {  
    @Id  
    protected Integer employeeId;  
    ...  
}
```

JPA et l'héritage - exemple

Classe dérivée :

```
@Entity  
public class FullTimeEmployee extends Employee {  
    protected Integer salary;  
  
    ...  
}
```

JPA et l'héritage - exemple

Autre classe dérivée :

```
@Entity
```

```
public class PartTimeEmployee extends Employee {  
    protected Float hourlyWage;  
}
```

JPA et les types énumérés

JPA supporte également les types énumérés.

```
public enum EmployeeStatus { FULL_TIME, PART_TIME, CONTRACT }

public enum SalaryRate { JUNIOR, SENIOR, MANAGER, EXECUTIVE }

@Entity public class Employee {

    ...

    public EmployeeStatus getStatus() {...}

    @Enumerated(String)    // stocke "JUNIOR" par exemple

    public SalaryRate getPayScale() {...}

    ...

}
```

JPA et les types énumérés

Autre possibilité.

```
public enum EmployeeStatus { FULL_TIME, PART_TIME, CONTRACT }

public enum SalaryRate { JUNIOR, SENIOR, MANAGER, EXECUTIVE }

@Entity public class Employee {

    ...

    public EmployeeStatus getStatus() {...}

    @Enumerated(ORDINAL)    // stocke 0, 1, etc.

    public SalaryRate getPayScale() {...}

    ...

}
```


Les relations dans JPA

Les relations entre entités peuvent être déclarées au moyen des annotations suivantes :

`@OneToOne` ==> utilisée pour indiquer une relation 1-1 entre entités (c'est-à-dire une jointure 1-1 entre les tables représentant les instances)

`@OneToMany` ==> utilisée pour indiquer une relation 1-n entre entités

`@ManyToOne` ==> relation n-1 entre entités

`@ManyToMany` ==> relation n-p entre entités

Sens d'une relation

Une relation peut être **unidirectionnelle** ou **bidirectionnelle** :

- relation unidirectionnelle : seule une entité “**porte la relation**”, au moyen d'un attribut qui est une référence (éventuellement multiple) vers une ou plusieurs instances de l'autre entité
- relation bidirectionnelle : chaque entité porte la relation au moyen d'un attribut qui est une référence (éventuellement multiple) vers un ou plusieurs instances de l'autre entité

Sens d'une relation

Dans le cas d'une relation bidirectionnelle, un côté doit être déclaré **maître** et l'autre **esclave**.

Le côté **maître** est celui qui sera responsable des mises à jour dans la base de données : le service provider détectera les changements d'état relatifs à la relation et les reflètera dans la base de données.

Les modifications de cette relation effectuées sur l'entité côté **esclave** ne seront pas prises en compte par le service provider.

Sens d'une relation

Règles concernant le choix des côtés maître et esclave.

1) Les relations **symétriques** (1-1 et n-p) autorisent le **libre choix** du côté esclave et du côté maître.

2) Si la relation est asymétrique, le **côté 1** doit être le côté **esclave**.

La raison derrière ces règles sera vue plus tard...

Les relations 1-1

Exemple de relation 1-1 : une personne et son numéro de sécurité sociale.

```
@Entity
public class NumeroSecu implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private Long id;
    // Pas d'attribut référençant une personne
    // ==> il s'agit d'une relation unidirectionnelle
}
```

Les relations 1-1

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

    @Id
    private Long id;
    private String name;

    @OneToOne
    NumeroSecu numeroSecu; // cet attribut porte la relation
}
```

Les relations 1-1

Pour représenter la relation, le persistence provider créera une **table de jointure** ou une **colonne de jointure** dans l'une des tables, en fonction du contexte.

Cas d'une relation 1-1 unidirectionnelle :

- La table correspondant à l'entité portant la relation (source de la relation) se verra adjoindre une clé étrangère qui sera la clé primaire de l'entité destination de la relation

Les relations 1-1

Cas d'une relation 1-1 bidirectionnelle :

- La table correspondant à l'entité maître se verra adjoindre une clé étrangère qui sera la clé primaire de l'autre entité

Les relations 1-1

The screenshot displays a database management interface with the following components:

- SCHEMAS Panel:** A tree view on the left showing the database structure. The 'personne' table is highlighted in green. Under its 'Foreign Keys' section, 'FK_PERSONNE_NUMEROSECU_ID' is highlighted in yellow. A red oval highlights the 'NUMEROSECU_ID' column in the 'personne' table's column list.
- Table: personne:** A detailed view at the bottom showing the table's columns: ID (bigint(20) PK), NOM (varchar(255)), and NUMEROSECU_ID (bigint(20)).
- Related Tables:** A section below the columns showing the relationship: 'numerossecu (NUMEROSECU_ID → ID)'. A red oval highlights this relationship.
- Result Grid:** A table on the right showing data for the 'personne' table. A red oval highlights the 'NUM' column, which contains values 602, 604, 606, 608, and NULL.
- Action Output:** A table at the bottom right showing the results of a DROP operation on the 'personne' table.

ID	NOM	NUM
601	Patrick	602
603	Vincent	604
605	Emma	606
607	Claire	608
NULL	NULL	NULL

#	Time	Action
8	20:41:09	DROP
9	20:41:18	DROP

Les relations 1-n

Cas d'une relation unidirectionnelle 1-n (l'entité du côté 1 porte la relation) : une table de jointure est créée pour éviter la duplication d'informations

- Sinon, il faudrait qu'il y ait autant de lignes dans la table représentant l'entité côté 1 que d'entités côté n en relation avec une instance de cette entité
==> pas franchement optimal...

Les relations 1-n

Exemple dans le cadre du projet Fil Rouge : l'association entre une idée et une catégorie.

Plusieurs idées peuvent appartenir à la même catégorie.

Si on établit une relation unidirectionnelle du côté 1 (catégorie)...

Les relations 1-n

```
@Entity
public class CategorieEntity implements Serializable {
    @Id
    private Long id;
    private String name;
    @OneToMany // cet attribut porte la relation
    Set<IdeeEntity> ideesAssociees;
}
```

Les relations 1-n

@Entity

```
public class IdeeEntity implements Serializable {  
    @Id  
    private Long id;  
    private String titre;  
    private String description;  
    // Pas d'attribut référençant une catégorie  
    // ==> il s'agit d'une relation unidirectionnelle  
}
```

Les relations 1-n

Table de jointure

Table: **categorieentity_ideeentity**

Columns:

- CategorieEntity_ID** bigint(20) PK
- ideesAssociees_ID** bigint(20) PK

Related Tables:

- Target: **categorieentity** (CategorieEntity_ID → ID)
- On Update: RESTRICT
- On Delete: RESTRICT
- Target: **ideeentity** (ideesAssociees_ID → ID)
- On Update: RESTRICT
- On Delete: RESTRICT

Column	Type
CategorieEntity_ID	bigint(20)
ideesAssociees_ID	bigint(20)

Count: 2

Output:

#	Time	Action
62	17:23:13	SELECT * FROM formation
63	17:23:31	DROP TABLE 'formation';
64	17:23:34	DROP TABLE 'formation';
65	17:23:37	DROP TABLE 'formation';

Les relations 1-n

Cette table de jointure porte :

- une clé étrangère vers la clé primaire correspondant à la première entité
- et une clé étrangère vers la clé primaire correspondant à la seconde entité.

Les relations 1-n

Cas d'une relation unidirectionnelle n-1 (l'entité du côté n porte la relation) :

- la table correspondant à l'entité portant la relation (source de la relation) se verra adjoindre une clé étrangère qui sera la clé primaire de l'entité destination de la relation

Les relations 1-n

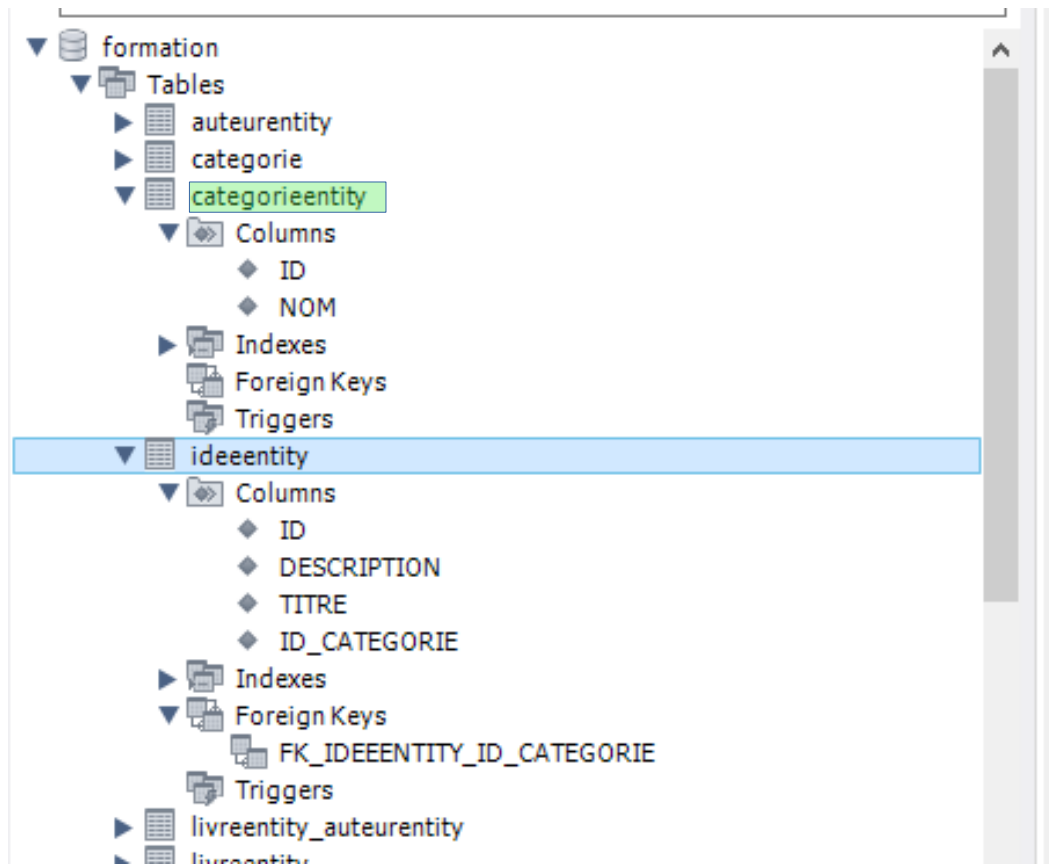
Cas d'une relation bidirectionnelle 1-n :

- même solution : une colonne de jointure est créée dans l'entité maître (côté n), elle porte une clé étrangère qui est la clé primaire de l'entité esclave (côté 1).

Exemple dans le cadre du projet Fil Rouge : plusieurs idées peuvent appartenir à la même catégorie.

Si on établit une relation bidirectionnelle entre idées et catégories...

Les relations 1-n



Les relations n-p

Cas d'une relation n-p (unidirectionnelle ou bidirectionnelle) :

- une table de jointure est créée, elle comporte une association de deux clés étrangères, l'une vers la table de la première entité, l'autre vers la table de la seconde entité.

Les relations dans JPA : synthèse

Relation	Unidirectionnelle	Bidirectionnelle
1-1	Colonne de jointure dans l'entité déclarant la relation	Colonne de jointure dans l'entité maître
1-n	Table de jointure avec clés étrangères vers chacune des entités	Colonne de jointure dans l'entité maître
n-1	Colonne de jointure dans l'entité déclarant la relation	
n-p	Table de jointure avec clés étrangères vers chacune des entités	

Les relations dans JPA

Retour vers l'exemple de relation unidirectionnelle 1-1

Le persistence provider est au courant de la relation existant entre une personne et un numéro de sécurité sociale

→ une personne peut être créée et associée à un numéro de sécurité sociale non encore persisté dans la base : lorsque la personne sera persistée, alors le numéro de sécurité sociale sera aussi persisté.

Les relations dans JPA

```
EntityManager em = EntityManagerFactory.create();

em.getTransaction().begin();

for (int i = 0; i < titresEtDescs.length; ++i) {

    CategorieEntity categorie = new CategorieEntity(noms[i]);

    IdeeEntity idee = new IdeeEntity(titresEtDescs[i][0],
titresEtDescs[i][1], categorie);

    em.persist(idee);

}

em.getTransaction().commit();
```

Les relations dans JPA

Dans l'exemple précédent, la catégorie `categorie` est enregistrée dans la base bien que l'instruction `em.persist()` soit appliquée sur la variable `idee`, car une relation a été établie de l'idée (variable `idee`) vers la catégorie (`categorie`) dans le constructeur de `IdeeEntity`.

Ou du moins...

Les relations dans JPA

```
36 @ManyToOne
37 CategorieEntity categorie;
38
39 public IdeeEntity() {
40     this.titre = "";
41     this.description = "";
42 }
43
44 public IdeeEntity(String titre, String description, CategorieEntity categorie) {
45     this.titre = titre;
46     this.description = description;
47 }
```

exercice3.IdeeEntity

Output - Exercise3 (test) HTTP Server Monitor Search Results Test Results X

exercice3.CategorieEntityTest X

Tests passed: 50,00 %

1 test passed, 1 test caused an error. (3,521 s)

- exercice3.CategorieEntityTest Failed
 - testInsert passed (0,09 s)
 - testOtherInsert caused an ERROR: java.lang.IllegalStateException: During synchronization a new object was found through a relationship that was not marked as managed. (56 chars)
 - java.lang.IllegalStateException: During synchronization a new object was found through a relationship that was not marked as managed. (56 chars)
 - javax.persistence.RollbackException
 - at org.edipse.persistence.internal.jpa.transaction.EntityTransactionImpl.commit(EntityTransactionImpl.java:157)
 - at org.edipse.persistence.internal.sessions.RepeatableWriteUnitOfWork.discoverUnregisteredNewObjects(RepeatableWriteUnitOfWork.java:273)
 - at org.edipse.persistence.internal.sessions.UnitOfWorkImpl.calculateChanges(UnitOfWorkImpl.java:723)
 - at org.edipse.persistence.internal.sessions.UnitOfWorkImpl.commitToDatabaseWithChangeSet(UnitOfWorkImpl.java:1516)
 - at org.edipse.persistence.internal.sessions.RepeatableWriteUnitOfWork.commitRootUnitOfWork(RepeatableWriteUnitOfWork.java:273)
 - at org.edipse.persistence.internal.sessions.UnitOfWorkImpl.commitAndResume(UnitOfWorkImpl.java:1169)

Le cascading

Pourquoi cette exception est-elle générée ?

Parce que la relation `IdeeEntity` → `CategorieEntity` n'a pas été indiquée comme **cascadante pour l'enregistrement (*persist*)**.

Voir le message d'erreur :

```
java.lang.IllegalStateException: During synchronization  
a new object was found through a relationship that was  
not marked cascade PERSIST:  
exercice1.CategorieEntity[ id=null ].
```

Le cascading

Il faut préciser au persistence provider que les opérations sur une entité `IdeeEntity` doivent être “cascadées” vers la ou les entités en relation avec elle.

Il est possible de préciser au persistence provider quels traitements doivent être cascades, sachant qu'un `EntityManager` permet d'effectuer 5 types d'opérations sur une entité :

- enregistrement initial dans la base de données (`persist`)
- effacement dans la base de données (`remove`)
- rafraichissement depuis la base de données (`refresh`)
- mise à jour de l'entité dans la base de données (`merge`)
- détachement du contexte de persistence (`detach`).

Le cascading

Exemple, si l'on veut que seuls les enregistrements initiaux d'entité et les effacements soient cascades.

```
@OneToOne (cascade={ CascadeType.PERSIST,  
                    CascadeType.REMOVE}, orphanRemoval=true)  
CategorieEntity categorie;
```

Le cascading

Autre exemple, si l'on veut que toutes les opérations soient cascadées.

```
@OneToOne(cascade={ CascadeType.ALL }, orphanRemoval=true)  
CategorieEntity categorie;
```

Le cascading

L'information `orphanRemoval=true` permet d'indiquer au persistence provider que si un effacement d'entité à la source de la relation (ici `IdeeEntity`) fait qu'une entité destination (ici une `CategorieEntity`) n'est plus attachée à une entité d'origine (on parle d'entité "orpheline"), alors cette entité doit être effacée.

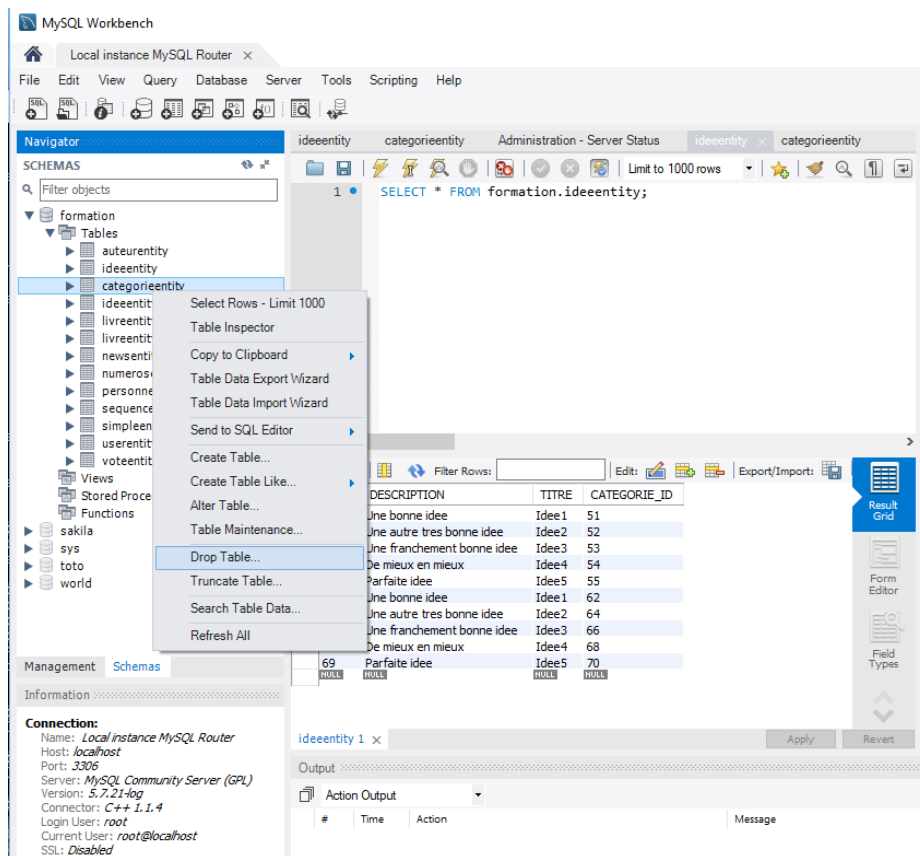
```
@OneToOne (cascade={CascadeType.PERSIST,  
CascadeType.REMOVE}, orphanRemoval=true)  
CategorieEntity categorie;
```

Les relations dans JPA

Réaliser l'exercice 3 (relation 1-1 unidirectionnelle).

Les relations dans JPA

Tentative
d'effacement de la
table des
catégories.



Les relations dans JPA

Résultat de la tentative d'effacement de la table des catégories.

The screenshot shows the MySQL Workbench interface. On the left, the 'SCHEMAS' pane shows a tree view of databases, with 'formation' expanded and its tables listed. The 'Tables' list includes 'ideentity', 'categorieentity', and others. The 'Management' tab is active, showing the 'Schemas' section. The 'Connection' information for 'Local instance MySQL Router' is displayed at the bottom left.

The central SQL editor shows the following query:

```
SELECT * FROM formation.ideentity;
```

The 'Result Grid' pane displays the results of the query, showing a table with columns 'ID', 'DESCRIPTION', 'TITRE', and 'CATEGORIE_ID'. The data includes 13 rows of information about ideas and their categories.

The 'Output' pane at the bottom shows an error message:

#	Time	Action	Message	Duration / Fetch
1	22:40:10	DROP TABLE formation.`categorieentity`	Error Code: 1217. Cannot delete or update a parent row: a...	0.015 sec

The error message indicates that the table 'categorieentity' cannot be dropped because it is a parent row in a foreign key relationship.

Les relations dans JPA

Résultat de la tentative d'effacement de la table des catégories.

```
22:21:05 DROP TABLE `formation`.`categorieentity` Error  
Code: 1217. Cannot delete or update a parent row: a  
foreign key constraint fails 0.016 sec
```

Le choix d'un type de relation

Bien sûr, une catégorie peut être associée à plus d'une idée.

Par contre, une idée n'appartient qu'à une seule catégorie – dans le cadre du projet FilRouge en tout cas.

Il faut donc établir une relation 1-n entre une `CategorieEntity` et une `IdeeEntity`.

Inversement, on pourrait définir une relation n-1 entre une `IdeeEntity` et une `CategorieEntity`.

Le choix d'un type de relation

Est-ce que cela présente un intérêt que de définir ces deux relations, ou bien une seule d'entre elles peut-elle suffire ? Et dans ce dernier cas, laquelle choisir ?

La question est de savoir si on veut retrouver **les idées associées à une catégorie**, ou bien si on veut rechercher **la catégorie associée à une idée**, ou bien encore si l'on veut **faire les deux types de recherche**.

Le choix d'un type de relation

Si l'on veut seulement retrouver les idées associées à une catégorie, on établit une relation 1-n unidirectionnelle de `CategorieEntity` **vers** `IdeeEntity`.

`@OneToMany`

```
private Set<IdeeEntity> ideesAssociees;
```

Le choix d'un type de relation

Si l'on veut seulement retrouver la catégorie à laquelle appartient une idée, on établit une relation n-1 unidirectionnelle de `IdeeEntity` vers `CategorieEntity`.

```
@ManyToOne
```

```
private CategorieEntity categorie;
```

Le choix d'un type de relation

Et si l'on veut faire les deux types de recherche ?

Alors on définit une relation 1-n **bidirectionnelle** entre `CategorieEntity` et `IdeeEntity`.

Quel que soit le choix, le service provider va toujours organiser les tables de la même manière.

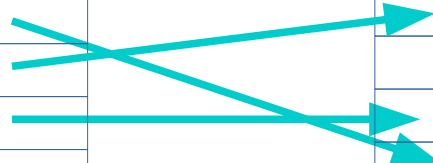
Spécifier un type de relation

La table du côté n va porter la relation vers la table du côté 1, sous forme d'une **clé étrangère**.

Dans notre exemple : la table `IdeeEntity` va inclure une clé étrangère vers la table `CategorieEntity`.

Id	Titre	Descr.	Id_categ
1	Idée 1	Desc 1	4
2	Idée 2	Desc 2	1
3	Idée 3	Desc 3	3
...

Id	Nom
1	Science
2	Musique
3	Médias
4	Divers



Spécifier un type de relation

Comment indique t'on au service provider qu'une relation doit être établie, dans ces trois cas ?

Premier cas : relation 1-n unidirectionnelle.

On décrit la relation dans l'entité `CategorieEntity` :

@OneToMany

`@JoinColumn(name="ID_CATEGORIE")`

`private Set<IdeeEntity> ideesAssociees;`



*La **multiplicité** est **aussi** indiquée par le fait que l'on utilise une Collection Java. Seule une Collection Java est autorisée.*

Spécifier un type de relation

Deuxième cas : relation n-1 unidirectionnelle.

On décrit la relation dans l'entité `IdeeEntity` :

@ManyToOne

`@JoinColumn(name="ID_CATEGORIE")`

`private CategorieEntity categorie;`

Spécifier un type de relation

Troisième cas : relation 1-n bidirectionnelle.

Première solution : on décrit la relation dans les deux entités de manière symétrique :

@ManyToOne

`@JoinColumn(name="ID_CATEGORIE")`

`private CategorieEntity categorie;`

@OneToMany

`@JoinColumn(name="ID_CATEGORIE")`

`private Set<IdeeEntity> ideesAssociees;`

Spécifier un type de relation

Première remarque : il faut mettre à jour les deux côtés de la relation.

1) Indiquer la catégorie associée à une idée :

```
uneIdee.categorie = uneCategorie;
```

2) Ajouter l'idée dans l'ensemble des idées associées à la catégorie :

```
uneCategorie.ideesAssociees.add(uneIdee);
```

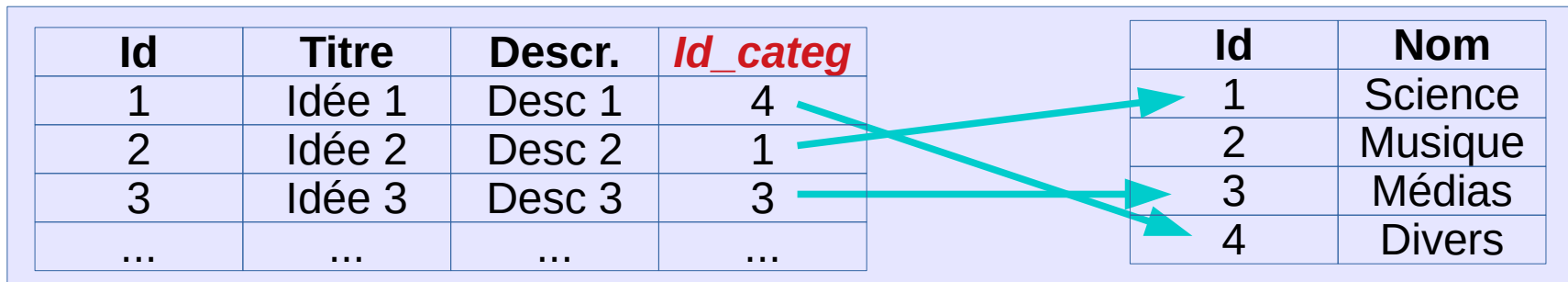
En effet, quand on met à jour un objet Java, la JVM ne “sait pas” déduire l'autre côté de la relation. Il faut donc le lui dire.

Spécifier un type de relation

Deuxième remarque : du coup, le service provider va effectuer deux mises à jour de la base de données. Or, une seule suffirait, puisque la base de données, à la différence de la JVM, stocke la relation en un seul endroit (la clé étrangère dans la colonne de jointure).

Spécifier un type de relation

Dans le monde relationnel : un seul *lien*.



```
Long id;  
String titre;  
String description;  
CategorieEntity categorie;
```

```
Long id;  
String nom;  
Set<IdeeEntity> idees;
```

Dans le monde Java : deux *liens*.

Spécifier un type de relation

La solution consiste à déclarer un côté responsable des mises à jours dans la base de données.

Le côté **maître** est responsable des mises à jour.

Rappel : *si la relation est asymétrique, le **côté 1** doit être le côté **esclave**.*

Appliqué au projet Fil Rouge, cela indique que le côté esclave est le côté `CategorieEntity`.

Spécifier un type de relation

Nouvelle déclaration de la relation.

```
@ManyToOne  
  
@JoinColumn(name="ID_CATEGORIE")  
private CategorieEntity categorie;
```

Fichier IdeeEntity.java

```
@OneToMany(mappedBy="categorie")  
private Set<IdeeEntity> ideesAssociees;
```

Fichier CategorieEntity.java

Spécifier un type de relation

Le fait d'utiliser le paramètre `mappedBy` dans une annotation indique que l'attribut sur lequel porte ce paramètre appartient à l'entité **esclave**, celle qui n'est pas responsable de la mise à jour de la base de données.

Ce paramètre signifie qu'une instance de la classe portant la relation (`IdeeEntity` ici) établit un lien avec une instance de la classe portant l'attribut marqué (`CategorieEntity`), et que ce lien est établi par l'attribut dont le nom est indiqué par la valeur du paramètre `mappedBy`.



Spécifier un type de relation

Explication



```
@ManyToOne
@JoinColumn(name="ID_CATEGORIE")
private CategorieEntity categorie;
```

IdeeEntity

Côté **maître**

```
@OneToMany(mappedBy="categorie")
private Set<IdeeEntity> ideesAssociees;
```

CategorieEntity

Côté **esclave**

Spécifier un type de relation

The screenshot displays a database management interface with two main panes. The left pane shows the 'formation' database structure, including tables like 'categorie', 'categorieentity', and 'ideentity'. The 'ideentity' table structure is expanded, showing columns: ID, DESCRIPTION, TITRE, and ID_CATEGORIE. The 'ID_CATEGORIE' column is highlighted in pink. The right pane shows the 'Result Grid' for the 'ideentity' table, displaying 11 rows of data. The 'ID_CATEGORIE' column in the data grid is also highlighted in pink.

Table Structure: ideentity

- Columns: ID, DESCRIPTION, TITRE, ID_CATEGORIE
- Indexes: (none listed)
- Foreign Keys: FK_IDEEENTITY (ID_CATEGORIE)
- Triggers: (none listed)

Result Grid Data:

ID	DESCRIPTION	TITRE	ID_CATEGORIE
2106	Une bonne idee	Idee1	2101
2107	Une autre tres bonne idee	Idee2	2102
2108	Une franchement bonne idee	Idee3	2103
2109	De mieux en mieux	Idee4	2104
2110	Parfaite idee	Idee5	2105
2111	Une bonne autre idee	AutreIdee1	2101
2112	Une autre tres bonne autre idee	AutreIdee2	2102
2113	Une franchement bonne autre idee	AutreIdee3	2103
2114	Autre de mieux en mieux	AutreIdee4	2104
2115	Parfaite autre idee	AutreIdee5	2105
NULL	NULL	NULL	NULL

Spécifier un type de relation

Avec cette nouvelle manière de déclarer la relation, une seule mise à jour de la base de données sera réalisée quand les deux entités (idée et catégorie correspondante) seront mises à jour, celle qui résultera de l'instruction :

```
uneIdee.categorie = uneCategorie;
```

L'autre instruction ne donnera pas lieu à mise à jour :

```
uneCategorie.ideesAssociees.add(uneIdee) ;
```

Cohérence objets / tables relationnelles

On a donc amélioré l'efficacité de notre application.

Par contre, on introduit ainsi un **risque d'incohérence** entre les objets Java et la base de données.

Dans quel cas de figure pourrait-il y avoir incohérence ?

Et que pourrait-on faire pour éviter cet écueil ?

QUIZZ

Cohérence objets / tables relationnelles



Solution : on réalise l'ajout d'une idée dans la liste des idées associées à une catégorie au moyen d'une méthode de `CategorieEntity`, et dans cette méthode on réalise **aussi** la relation **inverse**, celle qui déclenchera la mise à jour de la base de données.

Cohérence objets / tables relationnelles

```
public class CategorieEntity {  
    ...  
    @OneToMany(mappedBy="categorie")  
    @JoinColumn(name="ID_CATEGORIE")  
    private Set<IdeeEntity> ideesAssociees;  
    ...  
    public void ajouteIdee(IdeeEntity idee) {  
        this.ideesAssociees.add(idee);  
        idee.setCategorie(this);  
    }  
}
```

Les relations dans JPA

La règle à retenir est que **la création d'une relation 1-n bidirectionnelle entre deux entités** doit se faire :

- par une méthode **portée par le côté 1** (côté esclave),
- et que cette méthode doit **établir la relation dans les deux sens**.

Les relations dans JPA

Et si ça n'est pas parfaitement clair...



ça s'éclaircira avec la pratique.

En tout cas, appliquer les règles indiquées :

- choix du type de relation en fonction du besoin
- placement du côté maître et du côté esclave si nécessaire et en fonction du type de relation
- faire établir une relation par une méthode du côté 1

fonctionnera.

Les relations dans JPA

Réaliser l'exercice 4 (relation 1-n bidirectionnelle).

Spécifier un type de relation

Cas des relations n-p.

Exemple : une bibliothèque.

La bibliothèque recense des livres.

Un livre est rédigé par un ou plusieurs auteurs.

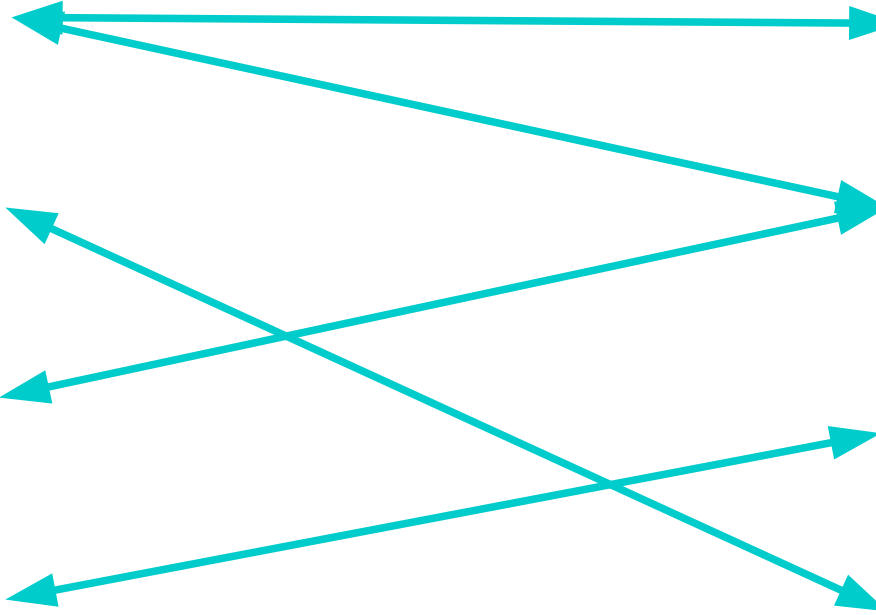
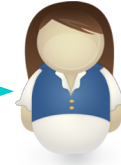
Un auteur peut avoir rédigé plusieurs livres.

Spécifier un type de relation

*Livres
(si si !)*



*Auteurs
(je vous assure)*



Spécifier un type de relation

L'entité LivreEntity et ses attributs

```
@Entity
public class LivreEntity implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private Long id;
    private String titre;
}
```

Spécifier un type de relation

L'entité AuteurEntity et ses attributs

```
@Entity
public class AuteurEntity implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    private Long id;
    private String nom;
    private String prenom;
}
```

Spécifier un type de relation

Si l'on souhaite pouvoir rechercher les livres qu'un auteur a rédigés, mais que la recherche inverse (quels auteurs ont écrit un livre donné ?) ne nous intéresse pas, on peut se contenter d'une relation unidirectionnelle de `AuteurEntity` vers `LivreEntity`:

```
@ManyToMany
```

```
Set<LivresEntity> livresRediges;
```

```
Classe AuteurEntity
```

Spécifier un type de relation

```
@Entity
```

```
public class AuteurEntity implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    private Long id;  
    private String nom;  
    private String prenom;  
    @ManyToMany  
    Set<LivreEntity> livresRediges;  
}
```

Spécifier un type de relation

Si l'on souhaite pouvoir rechercher les auteurs d'un livre donné, mais que la recherche inverse ne nous intéresse pas, on peut se contenter d'une relation unidirectionnelle de `LivreEntity` vers `AuteurEntity` :

```
@ManyToMany
```

```
Set<AuteurEntity> auteurs;
```

```
Classe LivreEntity
```


Spécifier un type de relation

```
@Entity
```

```
public class LivreEntity implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    private Long id;  
    private String titre;  
    @ManyToMany  
    Set<AuteurEntity> auteurs;  
}
```

Spécifier un type de relation

Très généralement, on veut effectuer des recherches dans les deux sens; on a donc besoin d'une relation **bidirectionnelle**.

Pour optimiser les accès à la base de données, il est nécessaire de définir un côté **maître** et un côté **esclave**.

Il s'agit d'une relation **symétrique**.

Rappel : *Les relations symétriques (1-1 et n-p) autorisent le libre choix du côté esclave et du côté maître.*

Spécifier un type de relation

Sachant que :

- les deux représentations de la relation dans le monde Java doivent être établies / modifiées en cas de création / modification d'une relation,
- et que pour éviter les incohérences entre le monde Java et le monde relationnel il est préférable que le côté esclave fournisse une méthode réalisant l'établissement / la modification des deux représentations de la relation

...

Spécifier un type de relation

... il faut se poser la question de savoir quelle est la manière la plus naturelle de créer cette relation : à quelle entité veut-on ajouter des éléments du type l'autre entité, de manière à créer la relation ?

Appliqué au cas de la bibliothèque, cela consiste à se demander si l'on veut **ajouter les auteurs à un livre**, ou plutôt à **ajouter les livres qu'il a rédigés à un auteur**.

Spécifier un type de relation

Si on veut ajouter des auteurs à un livre, par exemple au moyen d'une méthode de l'entité `LivreEntity` :

```
public void ajouteAuteur(AuteurEntity auteur) {  
    this.auteurs.add(auteur);  
}
```

alors il vaut mieux que cette entité soit l'entité **esclave**, et que cette méthode **établis**e aussi la relation inverse (l'ajout d'un livre à l'ensemble des livres rédigés par cet auteur).

Spécifier un type de relation

```
@Entity
```

```
public class LivreEntity implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    @Id  
    private Long id;  
    private String titre;  
  
    @ManyToMany (mappedBy="livresRediges") // côté esclave  
    Set<AuteurEntity> auteurs;  
  
    ...  
}
```

Spécifier un type de relation

...

```
public void ajouteAuteur(AuteurEntity auteur) {  
    this.auteurs.add(auteur);  
    auteur.ajouteLivre(this); // relation inverse  
}
```

La couche Persistance

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

    @Id
    private Long id;
    private String nom;
    private String prenom;

    @ManyToMany // côté maître
    Set<LivreEntity> livresRediges;

    ...
}
```


Spécifier un type de relation

...

```
public void ajouteLivre(LivreEntity livre) {  
    this.livresRediges.add(livre);  
}
```

Spécifier un type de relation

Si on veut ajouter des livres à un auteur, par exemple au moyen d'une méthode de l'entité `AuteurEntity` :

```
public void ajouteLivre(LivreEntity livre) {  
    this.livresRediges.add(livre);  
}
```

alors il vaut mieux que cette entité soit l'entité **esclave**, et que cette méthode **établis**se aussi la relation inverse (l'ajout d'un auteur à l'ensemble des auteurs d'un livre).

Spécifier un type de relation

```
@Entity
```

```
public class LivreEntity implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    private Long id;  
    private String titre;  
    @ManyToMany // côté maître  
    Set<AuteurEntity> auteurs;  
    ...  
}
```

Spécifier un type de relation

...

```
public void ajouteAuteur(AuteurEntity auteur) {  
    this.auteurs.add(auteur);  
}
```

Spécifier un type de relation

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

    @Id
    private Long id;
    private String nom;
    private String prenom;

    @ManyToMany (mappedBy="auteurs")           // côté esclave
    Set<LivreEntity> livresRediges;

    ...
}
```

Spécifier un type de relation

...

```
public void ajouteLivre(LivreEntity livre) {  
    this.livresRediges.add(livre);  
    livre.ajouteAuteur(this); // relation inverse  
}
```

La couche Persistance

Réaliser l'exercice 5 (relation n-p bidirectionnelle).

Autres informations

Surcharger les annotations

Les annotations peuvent être surchargées en plaçant des valeurs dans un fichier XML appelé le *deployment descriptor*.

Cela permet de modifier le comportement des entités sans modifier le code source ni même recompiler.

Le langage de requêtes JPA

JPA définit un langage pseudo-SQL pour définir des requêtes : JPQL (Java Persistence Query Language).

Ce langage est indépendant d'un moteur de bases de données relationnelles et donc portable.

Plus d'infos sur le site d'Oracle :

`https://docs.oracle.com/javaee/7/tutorial/persistence-querylanguage.htm#BNBTG`

La Java Persistence Criteria API

JPA définit également une API permettant de définir des requêtes : Java Persistence Criteria API.

Ce langage est indépendant d'un moteur de bases de données relationnelles et donc portable.

Plus d'infos sur le site d'Oracle :

`https://docs.oracle.com/javaee/7/tutorial/persistence-criteria.htm#GJITV`

JPQL vs. Java Persistence Criteria API

Différences entre JPQL et Criteria API :

- les requêtes JPQL sont plus concises et claires
- elles ressemblent beaucoup au SQL
- elles nécessitent d'utiliser le bon cast lors de la récupération des résultats
- cet inconvénient n'existe pas avec la Criteria API
- ces dernières requêtes sont également plus performantes.

La couche Persistance

Pour conclure :

- JPA est un modèle assez complexe... car la réalité est assez complexe
 - la spécification de JPA :
 - ❗ *JSR 338: Java™ Persistence API, Version 2.1*
fait quand même **570 pages**
- il y a des règles à retenir (voir planches précédentes)
- il faut pratiquer (implémenter la persistance de données différentes)

Quiz (niveau avancé)

L'annotation `@ManyToOne` ne propose pas de paramètre `mappedBy`.

Pourquoi ?

La couche Business

Rappel

La couche Business (couche Métier) inclut tous les composants implémentant la **logique applicative** et **l'accès aux données**.

En gros :

- Logique applicative ⇒ EJB Session et “Message-driven”
- Objets métier ⇒ entités persistentes