



HIBERNATE





Cours Sur Java Hibernate Framework



Concept De Persistance



Qu'est-ce que la persistance ?



La persistance :




- ☕ **Capacité d'un objet à survivre à la fin de son programme.**
- ☕ **Utilisée pour stocker les données à long terme.**
- ☕ **Utilisation de la base de données**



Quels sont les avantages de la persistance ?



La persistance permet de :

-  **Conserver les données d'une exécution à l'autre de l'application.**
-  **Conserver les données même si l'application ou l'ordinateur s'arrête.**
-  **Partager les données entre plusieurs applications.**



Quels sont les inconvénients de la persistance ?



La persistance a aussi des inconvénients :






- ☕ **Complicée à mettre en œuvre.**
- ☕ **Plus lente que les données en mémoire.**
- ☕ **Plus coûteuse en termes de ressources.**



Quel gestionnaire de base de donnée utiliser ?



Les gestionnaires de base de données les plus populaires sont :

-  **MySQL**
-  **PostgreSQL**
-  **Oracle**
-  **SQL Server**
-  **SQLite**



Mysql



MySQL est un système de gestion de base de données relationnelle (SGBDR) libre et open source. Il est développé, distribué et soutenu par Oracle Corporation.

C'est un système de gestion de base de données relationnelle (SGBDR) qui utilise le langage SQL pour gérer les données. Il stocke les données dans des tables, qui sont des ensembles de données organisées en lignes et en colonnes.

Il est utilisé par de nombreuses applications Web, telles que WordPress, Drupal, Joomla, Magento, phpBB, et d'autres.



Installation de mysql



Pour windows :

- ☕ **Télécharger le fichier d'installation de mysql**
- ☕ **Lancer l'installation**
- ☕ **Suivre les étapes d'installation**

Pour linux :

- ☕ **Sudo apt-get install mysql-server**
- ☕ **Sudo mysql_secure_installation**



Les Entité Persistante



Qu'est-ce qu'une entité persistante ?



Une entité persistante est une classe dont les instances sont stockées dans une base de données relationnelle.

Les entités persistantes sont les objets d'une application qui sont stockés dans une base de données. Les entités sont les objets que l'application manipule.



Quels sont les avantages des entités persistantes ?



- ☕ Les entités persistantes sont plus faciles à gérer que les données en mémoire.
- ☕ Les entités persistantes sont plus durables que les données en mémoire.
- ☕ Les entités persistantes sont plus faciles à partager que les données en mémoire.



Quels sont les inconvénients des entités persistantes ?



- ☕ Les entités persistantes sont plus difficiles à mettre en œuvre que les données en mémoire.
- ☕ Les entités persistantes sont plus lentes que les données en mémoire.
- ☕ Les entités persistantes sont plus coûteuses en termes de ressources que les données en mémoire.



Les POJO







Qu'est-ce qu'un POJO ?



POJO signifie Plain Old Java Object. Un POJO est un objet Java simple qui n'a pas de dépendances à des bibliothèques tierces.

Il contient :

-  **Des attributs privés**
-  **Constructeur par défaut**
-  **Des getters et des setters**
-  **Une méthode toString()**



A quoi sert un POJO ?



Un POJO sert à représenter les données dans une application java.

Nous allons utiliser des POJO pour représenter les données de notre base de données.

Il est important de créer des POJO pour chaque table de notre base de données.



exemple de POJO :

Rappel



```
public class Personne {
    private int id;
    private String nom;
    private String prenom;
    private String email;
    private String telephone;

    public Personne() {
    }

    public Personne(int id, String nom, String prenom, String email, String telephone) {
        this.id = id;
        this.nom = nom;
        this.prenom = prenom;
        this.email = email;
        this.telephone = telephone;
    }

    //Setters and Getters

    @Override
    public String toString() {
        return "Personne{" +
            "id=" + id +
            ", nom='" + nom + '\'' +
            ", prenom='" + prenom + '\'' +
            ", email='" + email + '\'' +
            ", telephone='" + telephone + '\'' +
            '}';
    }
}
```



Relation Database - POJO



Schema de base de données



shema de base de données :

id	nom	prenom	email	telephone
---	----	-----	-----	-----
1	john	Doe	Jon.Doe@gmail.com	0450957518
2	jane	Doe	Jane.Doe@gmail.com	0450957519



Relation entre la base de données et le POJO



```
| id | attribut | attribut | attribut | attribut |  
| 1 | nom | prenom | email | telephone |
```



Table



La table est l'élément de base d'une base de données relationnelle.

Elle est composée de lignes et de colonnes, et chaque ligne représente une entité persistante.



id	titre
---	-----
1	Harry Potter
2	Le seigneur des anneaux
3	Le hobbit



POJO :



```
public class Livre {  
    private int id;  
    private String titre;  
  
    public Livre() {  
    }  
  
    public Livre(int id, String titre) {  
        this.id = id;  
        this.titre = titre;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getTitre() {  
        return titre;  
    }  
  
    public void setTitre(String titre) {  
        this.titre = titre;  
    }  
  
    @Override  
    public String toString() {  
        return "Livre{" +  
            "id=" + id +  
            ", titre='" + titre + '\'' +  
            '}';  
    }  
}
```



Colonne



**Chaque colonne de la table est un attribut de l'entité persistante.
En reprenant l'exemple de la table "livre", chaque colonne est un
attribut de l'entité persistante "livre".**

**Si nous ajoutons un attribut "auteur" à l'entité persistante "livre",
nous devons ajouter une colonne "auteur" à la table "livre".**



POJO :



```
public class Livre {  
    private int id;  
    private String titre;  
    private String auteur;  
  
    public Livre() {  
    }  
  
    public Livre(int id, String titre, String auteur) {  
        this.id = id;  
        this.titre = titre;  
        this.auteur = auteur;  
    }  
  
    //setteur et getteur.  
  
    @Override  
    public String toString() {  
        return "Livre{" +  
            "id=" + id +  
            ", titre='" + titre + '\'' +  
            ", auteur='" + auteur + '\'' +  
            '}';  
    }  
}
```



id	titre	auteur
1	Harry Potter	J.K Rowling
2	Le seigneur des anneaux	J.R.R Tolkien
3	Le hobbit	J.R.R Tolkien



Ligne



Chaque ligne est une instance de l'entité persistante.

Elle doit avoir une valeur pour l'attribut "id" car c'est la clé primaire de la table.

Chaque ligne doit avoir une valeur pour chaque colonne, sinon la ligne ne sera pas considérée comme une entité persistante.

Elle permet de distinguer les entités persistantes.



Cellule



**Les cellules sont les valeurs des attributs de l'entité persistante.
En reprenant l'exemple de la table "livre", chaque cellule est une
valeur de l'entité persistante "livre".**



De JDBC A JPA








Qu'est-ce que JDBC ?



JDBC : Java Database Connectivity.

Avec JDBC, nous pouvons :




-  **Se connecter à une base de données relationnelle**
-  **Executer des requêtes SQL**
-  **Récupérer les résultats de la requête SQL**
-  **Récupérer les informations de la base de données**
-  **Fermer la connexion**



Avantages de JDBC



Les avantages de JDBC sont :





-  **JDBC est une API Java, donc elle est multiplateforme.**
-  **JDBC est une API standard, donc elle est utilisée par tous les fournisseurs de base de données.**
-  **JDBC est une API simple, donc elle est facile à utiliser.**



Inconvénients de JDBC



Les inconvénients de JDBC sont :

-  **JDBC est une API bas niveau.**
-  **JDBC est une API verbeuse.**
-  **JDBC est une API difficile à utiliser.**
-  **JDBC est une API difficile à maintenir.**






Qu'est-ce que JPA ?



JPA est l'acronyme de Java Persistence API.

C'est une API Java qui permet de communiquer avec une base de données relationnelle.

Avec JPA, nous pouvons faire la même chose que JDBC, mais en plus :

-  **Nous n'avons pas besoin de connaître la syntaxe SQL.**
-  **Nous n'avons pas besoin de fermer la connexion.**
-  **Nous n'avons pas besoin de gérer les connexions.**





JPA est donc plus simple et plus facile à utiliser et à maintenir que JDBC.



Avantages de JPA



Les avantages de JPA sont :

-  **JPA est une API Java, donc elle est multiplateforme.**
-  **JPA est une API standard, donc elle est utilisée par tous les fournisseurs de base de données.**
-  **JPA est une API simple, donc elle est facile à utiliser.**
-  **JPA est une API haut niveau, donc elle est facile à maintenir.**



Comment utiliser JPA ?

Pour utiliser JPA, nous devons :

- ☕ Ajouter les dépendances JPA, nous récupérerons les dépendances depuis glassfish.
- ☕ Créer une classe qui représente l'entité persistante.
exemple : la classe "Livre" qui représente l'entité persistante "livre".
- ☕ Créer une classe qui représente l'entité manager.
exemple : la classe "LivreManager" qui représente l'entité manager "livre".
- ☕ Créer une classe qui représente l'entité manager factory.
exemple : la classe "LivreManagerFactory" qui représente l'entité manager factory "livre".





l'entité persistante avec annotation :



```
@Entity
@Table(name = "livre")
public class Livre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String titre;
    private String auteur;

    public Livre() {
    }

    public Livre(int id, String titre, String auteur) {
        this.id = id;
        this.titre = titre;
        this.auteur = auteur;
    }

    //setteur et getteur.

    @Override
    public String toString() {
        return "Livre{" +
            "id=" + id +
            ", titre='" + titre + '\'' +
            ", auteur='" + auteur + '\'' +
            '}';
    }
}
```



l'entité manager : Comment utiliser JPA ?



```
public class LivreManager {
    private EntityManagerFactory emf;
    private EntityManager em;

    public LivreManager() {
        emf = Persistence.createEntityManagerFactory("livre");
        em = emf.createEntityManager();
    }

    public void addLivre(Livre livre) {
        em.getTransaction().begin();
        em.persist(livre);
        em.getTransaction().commit();
    }

    public void deleteLivre(Livre livre) {
        em.getTransaction().begin();
        em.remove(livre);
        em.getTransaction().commit();
    }

    public Livre getLivre(int id) {
        return em.find(Livre.class, id);
    }

    public List<Livre> getAllLivres() {
        return em.createQuery("select l from Livre l", Livre.class).getResultList();
    }

    public void close() {
        em.close();
        emf.close();
    }
}
```



Comment utiliser JPA ?



l'entité manager factory :

```
public class LivreManagerFactory {  
    public static LivreManager getLivreManager() {  
        return new LivreManager();  
    }  
}
```



Comment utiliser JPA ?



le code principal :

```
public class Main {  
    public static void main(String[] args) {  
        LivreManager lm = LivreManagerFactory.getLivreManager();  
  
        Livre l1 = new Livre(1, "titre1", "auteur1");  
        Livre l2 = new Livre(2, "titre2", "auteur2");  
  
        lm.addLivre(l1);  
        lm.addLivre(l2);  
    }  
}
```



Comment utiliser JPA ?



Après avoir créer les classes, nous devons créer un fichier persistence.xml dans le dossier META-INF.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">

    <persistence-unit name="livre">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <class>com.formation.jee.domain.Livre</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/livre"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password" value="root"/>
            <property name="eclipselink.ddl-generation" value="create-tables"/>
            <property name="eclipselink.ddl-generation.output-mode" value="database"/>
        </properties>
    </persistence-unit>
</persistence>
```




Hibernate



Qu'est-ce qu'hibernate ?



Hibernate est un framework Java qui permet de communiquer avec une base de données relationnelle.

Il implémente JPA, donc il permet de faire la même chose que JPA, mais avec plus de facilité, et possède des fonctionnalités supplémentaires.

Il est donc plus simple et plus facile à utiliser et à maintenir que JPA.

hibernate appartient à la famille des ORM (Object Relational Mapping).



ORM



ORM est l'acronyme de Object Relational Mapping.

C'est un mécanisme qui permet de faire le lien entre les objets et les tables de la base de données.

Nous avons vu que les objets sont des instances de classes et que les tables sont des instances de relations.




Les frameworks ORM permettent de faire le lien entre les objets et les tables de la base de données.



Avantages d'hibernate



Le framework hibernate propose des fonctionnalités supplémentaires par rapport à JPA :

-  **Il permet de générer automatiquement les tables de la base de données à partir des classes Java.**
-  **Il permet de faire des requêtes SQL plus complexes que JPA.**
-  **Hibernate permet de gérer les transactions, les connexions, les sessions, les requêtes, les résultats, automatiquement.**



Installation De Hibernate



Maven

Maven est un outil de gestion de dépendances.

Il permet de gérer les dépendances de notre projet.

Il permet de télécharger automatiquement les dépendances de notre projet.

De cette façon, nous n'avons pas besoin de télécharger manuellement les dépendances de notre projet.

Nous pouvons juste ajouter les dépendances dans le fichier pom.xml de notre projet.



```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.4.10.Final</version>
  </dependency>
</dependencies>
```



Jar



Si nous n'utilisons pas Maven, nous devons télécharger les dépendances manuellement.

Nous pouvons télécharger les dépendances sur le site hibernate.org.

Nous pouvons télécharger les drivers sur le site du fournisseur de base de données.(MySQL dans notre cas)

**Nous allons ajouter les jars dans le dossier lib de notre projet.
Il suffira alors de les ajouter au classpath de notre projet, pour pouvoir les utiliser.**







Projet Hibernate



La Strcture d'un projet hibernate



Un projet hibernate est composé de 3 packages :

-  **Entity qui contient les classes Java, et le dao**
-  **Util qui contient la classe HibernateUtil**
-  **Client qui contient le client**
-  **Le fichier hibernate.cfg.xml**



Database



Creation de la database



Pour utiliser hibernate, nous devons créer une base de données. C'est dans cette base de données que nous allons créer nos tables.

Pour créer une base de données, nous allons utiliser un terminal sql.

```
create database project1; -- creation de la base de données
use project1; -- selection de la base de données
create table Personne (id int primary key, nom varchar(50), prenom varchar(50)); -- creation de la table Personne
show tables; -- affichage des tables de la base de données
```



Le Fichier De Configuration XML



Hibernate.cfg.xml



☕ Le fichier de configuration XML est le fichier de configuration de hibernate.

☕ Il contient les informations de connexion à la base de données.

☕ Il se décompose en plusieurs parties :

les propriétés de connexion à la base de données

☕ Driver, url, username, password

les propriétés de configuration de hibernate

☕ Dialect, show_sql, format_sql, hbm2ddl.auto

les propriétés de configuration de la base de données

☕ Les propriétés : connection.pool_size, mapping, ...





Session-factory



```
<session-factory>  
<!-- toutes les propriety de session-factory -->  
</session-factory>
```

C'est dans cette session-factory que nous allons déclarer les propriétés de configuration de hibernate :

-  Elle sert à créer une session hibernate.
-  La session hibernate est l'interface entre notre application et la base de données.



Driver



```
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
```

- ☕ **Le driver est le pilote de la base de données.**
- ☕ **Il permet de se connecter à la base de données.**
Pour le driver de mysql, nous utilisons le driver
"com.mysql.jdbc.Driver".








Url



```
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/db_hibernate</property>
```

L'URL de la base de données.

Elle est composée de :

-  **Le protocole de connexion : jdbc**
-  **Le type de base de données : mysql**
-  **L'adresse de la base de données : localhost**
-  **Le port de la base de données : 3306**
-  **Le nom de la base de données : db_hibernate**



Hibernate.dialect



```
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
```

La dialecte est une classe qui permet de définir la syntaxe SQL de la base de données.

Elle est utilisée par hibernate pour générer les requêtes SQL.

Il existe une dialecte pour chaque base de données.



Hibernate.show_sql



```
<property name="hibernate.show_sql">true</property>
```

La propriété show_sql permet d'afficher les requêtes SQL générées par hibernate.

La valeur de la propriété est true ou false.

La valeur de la propriété est false par défaut.

Si la valeur de la propriété est true, hibernate affiche les requêtes SQL.



Hibernate.format_sql



```
<property name="hibernate.format_sql">true</property>
```

La propriété format_sql permet d'afficher les requêtes SQL générées par hibernate avec un formatage.

La valeur de la propriété est true ou false.

La valeur de la propriété est false par défaut.

Si la valeur de la propriété est true, hibernate affiche les requêtes SQL avec un formatage.



Hibernate.hbm2ddl.auto



```
<property name="hibernate.hbm2ddl.auto">update</property>
```

La propriété hbm2ddl.auto permet de définir le mode de création des tables de la base de données.

La valeur de la propriété est : create, create-drop, update, validate, none.

La valeur de la propriété est update par défaut



User



```
<property name="hibernate.connection.username">root</property>
```

Le nom d'utilisateur de la base de données.

si la base de données n'a pas d'utilisateur, nous utilisons root.

C'est le nom d'utilisateur par défaut de mysql, nous n'avons pas besoin de le [spécifier. Il](#) a été défini dans le fichier de configuration de mysql.



Password



```
<property name="hibernate.connection.password">root</property>
```

Le mot de passe de la base de données.

Vous pouvez utiliser le mot de passe par défaut de mysql, qui est vide.



Exemple de fichier de configuration XML



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/db_hibernate</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">none</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">root</property>
  </session-factory>
</hibernate-configuration>
```



Mapping



Qu'est ce que le mapping ?



Le mapping est le processus de création des tables de la base de données à partir des classes Java.

C'est la partie qui permet de faire le lien entre les classes Java et les tables de la base de données.

Il contient donc les informations sur les classes Java, elles seront utilisées pour initialiser les tables de la base de données.



Mapping xml



```
<mapping resource="domain/entities"/>
```

Le mapping xml est le mapping qui utilise un fichier xml pour définir les classes Java.

Il permet de récupérer les informations sur les classes Java dans le fichier xml, elles seront utilisées pour initialiser les tables de la base de données.



Hibernate-mapping



```
<hibernate-mapping>  
</hibernate-mapping>
```

Notre fichier de mapping xml contient une balise hibernate-mapping.

Toutes les informations sur les classes Java sont définies dans entre ces deux balises.

Elle contient les balises suivantes :

 **Class**

 **Id**

 **Property**



Class



```
<hibernate-mapping>
  <class name="domain.entities.User" table="user">
    <!-- contenu de la classe -->
  </class>
</hibernate-mapping>
```

La balise class permet de définir une classe Java.

Elle contient les attributs suivants :

☕ **Name** : le nom de la classe Java

☕ **Table** : le nom de la table de la base de données



Id



```
<hibernate-mapping>
  <class name="domain.entities.User" table="user">
    <id name="id" column="id" type="int">
      <generator class="native"/>
    </id>
  </class>
</hibernate-mapping>
```

La balise id indique que l'attribut est la clé primaire de la table de la base de données.

Elle contient les attributs suivants :

☕ **Name** : le nom de l'attribut

☕ **Column** : le nom de la colonne dans la base de données

☕ **Type** : le type de l'attribut



Property



```
<hibernate-mapping>
  <class name="domain.entities.User" table="user">
    <id name="id" column="id" type="int">
      <generator class="native"/>
    </id>
    <property name="name" column="name" type="string"/>
    <property name="email" column="email" type="string"/>
  </class>
</hibernate-mapping>
```




La balise property indique que l'attribut est une colonne de la table de la base de données.



Property



Elle contient les attributs suivants :

-  **Name** : le nom de l'attribut
-  **Column** : le nom de la colonne dans la base de données
-  **Type** : le type de l'attribut

Nous définissons les attributs de la classe User.



Exemple de mapping



```
package domain.entities;

class User {
    private int id;
    private String name;
    private String email;

    User() {
    }

    User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    // getters and setters

    toString() {
        return "User [id=" + id + ", name=" + name + ", email=" + email + "];"
    }
}
```




```
<hibernate-mapping>
  <class name="domain.entities.User" table="user">
    <id name="id" column="id" type="int">
      <generator class="native"/>
    </id>
    <property name="name" column="name" type="string"/>
    <property name="email" column="email" type="string"/>
  </class>
</hibernate-mapping>
```



TP : faire son premier mapping xml



En tenant compte du cours et de l'exemple precedent, faire le mapping xml de la classe Personne.



Util



Qu'est-ce que l'objet util d'hibernate ?



**Le but de cet utilitaire est de faciliter l'utilisation d'hibernate.
Nous allons créer une classe utilitaire qui aura pour but de créer
une session hibernate.**



Qu'est-ce qu'une session hibernate ?



Une session hibernate est un objet qui permet de faire des opérations sur la base de données.

Elle permet de faire des opérations CRUD (Create, Read, Update, Delete).

C'est l'objet qui permet de faire le lien entre les classes Java et les tables de la base de données.

L'utilitaire hibernate a pour seul but de créer une session hibernate.



Structure de HibernateUtil





SessionFactory





Exemple de classe HibernateUtil



```
class HibernateUtil {  
    private static final SessionFactory sessionFactory = buildSessionFactory();  
  
    private static SessionFactory buildSessionFactory() {  
        try  
        {  
            return new Configuration().configure().buildSessionFactory();  
        }  
        catch (Throwable ex)  
        {  
            System.err.println("Initial SessionFactory creation failed." + ex);  
            throw new ExceptionInInitializerError(ex);  
        }  
    }  
  
    public static SessionFactory getSessionFactory() {  
        return sessionFactory;  
    }  
  
    public static void shutdown() {  
        getSessionFactory().close();  
    }  
}
```




Client



Le client est la classe qui va utiliser l'utilitaire hibernate pour créer une session hibernate.

Il va utiliser la méthode `getSessionFactory()` de l'utilitaire pour créer une session hibernate.

Nous écrirons le main dans cette classe.



Transaction



Qu'est-ce qu'une transaction ?



Une transaction est une opération qui va être effectuée sur la base de données.

Elle peut être une insertion, une modification ou une suppression.

Une transaction est une opération atomique, c'est à dire que si une erreur survient pendant l'exécution de la transaction, toutes les modifications effectuées sur la base de données seront annulées.



Methode Save



Qu'est-ce que la methode save ?



La méthode save permet d'insérer un objet dans la base de données.

Elle prend en paramètre l'objet à insérer.

Elle s'utilise de la manière suivante :

```
session.save(objet);
```



Exemple de save



A l'interieur du client, nous allons créer un objet User et l'insérer dans la base de données.

```
session.beginTransaction();  
User user = new User("chris", "loisel");  
session.save(user);  
transaction.commit();
```



TP : Save Avec Le Fichier De Configuration Xml



Struture Transactionelle



Commit



**La méthode commit permet de valider la transaction.
Si la transaction est validée, toutes les modifications effectuées
sur la base de données seront effectuées.**

exemple :

```
Transaction transaction = session.beginTransaction();
```



Rollback



**La méthode rollback permet d'annuler la transaction.
Si la transaction est annulée, toutes les modifications effectuées sur la base de données seront annulées.
Nous utilisons la méthode rollback dans un bloc catch, c'est à dire que si une erreur survient pendant l'exécution de la transaction, la méthode rollback sera appelée.**



Exemple de transaction



```
Transaction transaction = session.beginTransaction();
try
{
    // code de la transaction
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



Annotation mapping



Nous avons vu comment faire un mapping xml.

Il existe une autre manière de faire un mapping, c'est avec les annotations.

Nous allons voir comment faire un mapping avec les annotations.



Mapping Par Annotation



Qu'est-ce que le mapping par annotation ?



Le mapping par annotation est une autre manière de faire un mapping.

Il permet de faire un mapping sans avoir à créer un fichier xml.

Il faut ajouter une dépendance dans le pom.xml pour pouvoir utiliser les annotations.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.2.10.Final</version>
</dependency>
```



@Entity



**Elle permet de dire que la classe est une entité.
Elle prend en paramètre le nom de la table.**

```
@Entity(name="personne")  
public class Personne {  
    ...  
}
```



@Table



Elle permet de dire que la classe est une table.
Elle prend en paramètre le nom de la table.

```
@Table(name="personne")  
public class Personne {  
    ...  
}
```




@Column



Elle permet de dire que l'attribut est une colonne.
Elle prend en paramètre le nom de la colonne.

```
@Column(name="nom")  
private String nom;
```



@Id



Elle permet de dire que l'attribut est une clé primaire.

Elle prend en paramètre le nom de la colonne.

Elle s'utilise avec @GeneratedValue qui prends ces paramètres :

☕ **Strategy** : permet de choisir le type de génération de la clé primaire

☕ **Generator** : permet de choisir le générateur de clé primaire

```
@Id  
@Column(name="id")  
private int id;
```



@GeneratedValue



Dans le cas d'une clé primaire auto-incrémentée, il faut ajouter l'annotation @GeneratedValue.

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



Strategy

Strategy permet de choisir le type de génération de la clé primaire.

Il existe 3 types de génération de clé primaire :

- 🔥 AUTO : permet de choisir le type de génération de clé primaire en fonction de la base de données
- 🔥 IDENTITY : permet de générer une clé primaire avec un auto-increment
- 🔥 SEQUENCE : permet de générer une clé primaire avec une séquence
- 🔥 TABLE : permet de générer une clé primaire avec une table

UUID : permet de générer une clé primaire avec un UUID





Exemple d'annotation mapping



```
@Entity(name="personne")
@Table(name="personne")
public class Personne {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;
    @Column(name="nom")
    private String nom;
    @Column(name="prenom")
    private String prenom;
    @Column(name="age")
    private int age;

    public Personne() {
    }

    public Personne(String nom, String prenom, int age) {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }

    // getters et setters

    // toString
}
```



TP : save avec le mapping par annotation



Faire un programme qui permet d'insérer un objet User dans la base de données.

L'objet User doit être créé avec le mapping par annotation.

Il suffit de faire un save de l'objet User.

L'objet User doit avoir les attributs suivants :

 Id

 Nom

 Prenom

 Age



Revenir Sur Le Fichier Xml



Hbm2ddl



**Dans le fichier xml, nous avons vu l'attribut hbm2ddl.
Il permet de créer la base de données à partir du mapping.
Il facilitera votre travail car vous n'aurez pas à créer la base de données à la main.**



Auto



L'attribut auto permet de créer la base de données à partir du mapping.

L'attribut auto est utilisé par défaut.



Create drop



L'attribut create drop permet de créer la base de données à partir du mapping et de la supprimer à la fin de l'exécution du programme.



Update



L'attribut update permet de mettre à jour la base de données à partir du mapping.

Il ne supprime pas les données déjà présentes dans la base de données.

Mais il peut supprimer des colonnes si elles ne sont plus présentes dans le mapping.



None



L'attribut none permet de ne pas créer la base de données à partir du mapping.

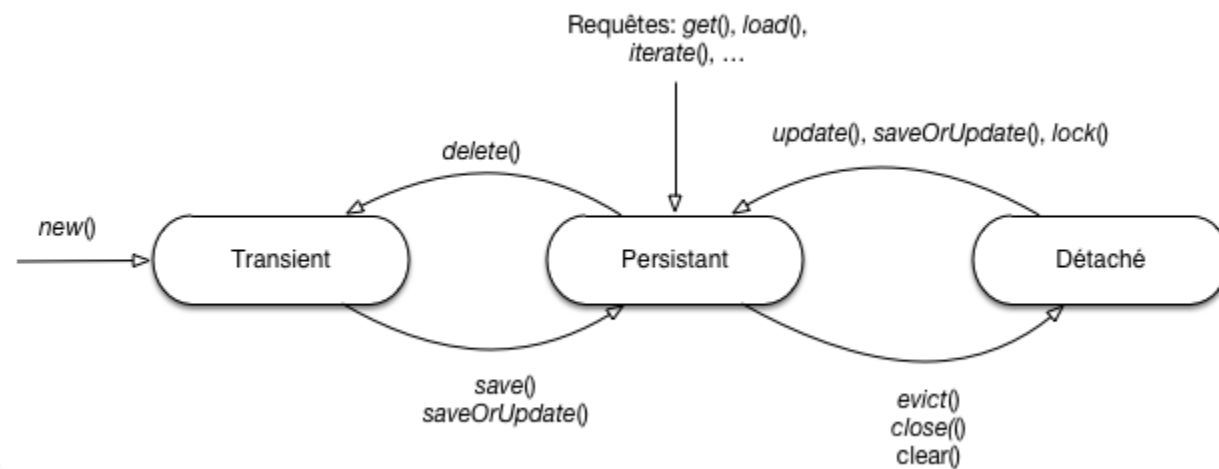
Il faut donc créer la base de données à la main.



Methode Persist



Cycle de vie d'un objet





Cycle de vie d'un objet





```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Personne personne = new Personne("Doe", "John", 25);
    session.persist(personne);
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



Persistant



Un objet persistant est :



-  **Un objet qui est attaché à une session**
-  **Un objet qui a été inséré dans la base de données**



Transient



Un objet transient est :




-  **Un objet qui n'est pas attaché à une session**
-  **Un objet qui n'a pas été inséré dans la base de données**



Detached



Un objet detached est :




-  **Un objet qui est attaché à une session**
-  **Un objet qui a été inséré dans la base de données**
-  **Un objet qui n'est plus attaché à une session**



Methode persist



La methode persist permet :

-  **D'insérer un objet dans la base de données**
-  **D'attacher l'objet à la session**
-  **Faire passer l'objet de transient à persistant**
si l'objet est deja persistant, la methode persist ne fait rien.



Quel différence avec save ?

`session.save` permet :

- ☕ **Retourne l'identifiant généré après la sauvegarde.**
- ☕ **Sauvegarde les changements dans la base de données en dehors de la transaction ;**
- ☕ **Session.save() pour un objet détaché créera une nouvelle ligne dans la table.**

`session.persist` permet :

- ☕ **Ne renvoie pas l'identifiant généré après la sauvegarde.**
- ☕ **Ne sauvegarde pas les changements dans la base de données en dehors de la transaction ;**
- ☕ **Session.persist() pour un objet détaché lèvera une `PersistentObjectException`, car cela n'est pas autorisé.**





Exemple de persist



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Personne personne = new Personne("Doe", "John", 25);
    session.persist(personne);
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



TP : Save Et Persist



Methode Get



Qu'est-ce que la methode get ?



La methode get permet de récupérer un objet à partir de sa clé primaire.

Nous avons vu que la clé primaire est générée automatiquement par la base de données.

Nous utiliserons donc la methode get pour récupérer un objet à partir de son identifiant.



Exemple de get



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Personne personne = session.get(Personne.class, 1);
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



TP : Get



Methode Update



Qu'est-ce que la methode update ?



Update permet de mettre à jour un objet dans la base de données.

Lorsque l'on fait un update, l'objet doit être persistant.

Une fois l'objet mis a jour, il est toujours persistant.



Exemple de update



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Personne personne = session.get(Personne.class, 1);
    personne.setNom("Doe");
    personne.setPrenom("Jane");
    personne.setAge(30);
    session.update(personne);
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



Methode SaveOrUpdate



Qu'est-ce que la methode saveOrUpdate ?



La methode `saveOrUpdate` permet :

☕ De mettre à jour un objet dans la base de données

☕ D'insérer un objet dans la base de données

en fonction de l'état de l'objet.

Si l'objet est persistant, la methode `saveOrUpdate` fait un update.

Si l'objet est transient, la methode `saveOrUpdate` fait un insert.



Exemple de saveOrUpdate



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Personne personne = new Personne("Doe", "John", 25);
    session.saveOrUpdate(personne);
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```




TP : Update Et SaveOrUpdate



Methode Delete



Qu'est-ce que la methode delete ?



La methode delete permet de supprimer un objet de la base de données.

L'objet doit être persistant.

Si l'objet n'est pas persistant, la methode delete ne fait rien.



Exemple de delete



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Personne personne = session.get(Personne.class, 1);
    session.delete(personne);
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



TP : Delete



Exemple de CRUD



```
create(new Personne("Doe", "John", 25));  
Personne personne = read(1);  
personne.setNom("Doe");  
personne.setPrenom("Jane");  
personne.setAge(30);  
update(personne);  
delete(personne);
```



TP : CRUD



Dao Pattern



Qu'est-ce que le dao pattern ?



Le dao pattern permet d'isoler les accès à la base de données dans des classes dédiées.

Sa logique est la suivante, nous créons une classe 'DAO' par entité.



Exemple de dao pattern



```
public class Personne
{
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    // constructeur, getters, setters
}
```



```
public class PersonneDao
{
    public void create(Personne personne)
    {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();
        try
        {
            session.persist(personne);
            transaction.commit();
        }
        catch (Exception e)
        {
            transaction.rollback();
        }
    }
    public Personne read(Integer id)
    {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();
        try
        {
            Personne personne = session.get(Personne.class, id);
            transaction.commit();
            return personne;
        }
        catch (Exception e)
        {
            transaction.rollback();
            return null;
        }
    }
    public void update(Personne personne)
    {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();
        try
        {
            session.update(personne);
            transaction.commit();
        }
        catch (Exception e)
        {
            transaction.rollback();
        }
    }
    public void delete(Personne personne)
    {
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();
        try
        {
            session.delete(personne);
            transaction.commit();
        }
        catch (Exception e)
        {
            transaction.rollback();
        }
    }
}
```



TP : Dao Pattern



Les Requetes Queries



Qu'est-ce que les requetes queries ?



Les requetes queries permettent de faire des requetes SQL en utilisant des objets Java.

Nous pouvons faire des requetes en :

 **SQL**

 **JPQL (Java Persistence Query Language)**



Le langage JPQL et la methode createQuery



Pour faire des requetes en utilisant JPQL, on utilise la methode

`createQuery` **de la classe** `Session`.

La methode `createQuery` **prend en paramètre une requete JPQL.**



Le langage JPQL



Le langage JPQL est un langage similaire au SQL.

Sa syntaxe est :

```
SELECT <attribut> FROM <entité> WHERE <condition>
DELETE FROM <entité> WHERE <condition>
CREATE TABLE <entité> (<attribut> <type>, <attribut> <type>, ...)
UPDATE <entité> SET <attribut> = <valeur>, <attribut> = <valeur>, ... WHERE <condition>
```




Methode createQuery



La methode `createQuery` retourne un objet de type `Query` .

L'objet `Query` permet d'executer la requete.

Il faut utiliser la methode `executeUpdate` pour les requetes de type `INSERT` , `UPDATE` et `DELETE` .

Il faut utiliser la methode `getResultList` pour les requetes de type `SELECT` .

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Query query = session.createQuery("SELECT p FROM Personne p WHERE p.age > 20");
    List<Personne> personnes = query.getResultList();
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



Methode createSQLQuery



La methode `createSQLQuery` permet de faire des requetes SQL.

La methode `createSQLQuery` s'utilise depuis la classe

`Session` , et prend en paramètre une requete SQL.

```
// requete SQL qui selectionne tout de la table personne  
Query query = session.createSQLQuery("SELECT * FROM personne");
```



Exemple de createSQLQuery



```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
try
{
    Query query = session.createSQLQuery("SELECT * FROM personne");
    List<Object[]> personnes = query.list();
    transaction.commit();
}
catch (Exception e)
{
    transaction.rollback();
}
```



TP : Requetes Queries



Les Relations Entre Les Entités



Relation entre les entités

Une relation entre les entités est une association entre deux entités.

Par exemple :

☕ Une personne peut avoir plusieurs adresses

☕ Une adresse appartient à une personne

Il existe 4 types de relations entre les entités :

☕ OneToOne : relation 1-1

☕ OneToMany : relation 1-n

☕ ManyToOne : relation n-1

☕ ManyToMany : relation n-n





Configuration de relations



En plus de la relation entre les entités, il faut configurer la relation dans les deux entités.

Il y a deux types de relations :

 **Relation unidirectionnelle**

 **Relation bidirectionnelle**



Relation one to one



La relation one to one est une relation 1-1.

C'est-à-dire qu'une entité peut avoir une seule relation avec une autre entité.

Et l'autre entité peut avoir une seule relation avec l'entité.

Exemple :

 **Une personne peut avoir une seule adresse**

 **Une adresse appartient à une seule personne**



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToOne
    private Adresse adresse;
    // constructeur, getters, setters
}
```



Fetch type



Le fetch type est un attribut de la relation, il se définit avec l'annotation `@OneToOne(fetch = FetchType.EAGER)` ou `@OneToOne(fetch = FetchType.LAZY)`.

Par défaut, le fetch type est `EAGER`.

Le fetch type `EAGER` permet de charger les données de la relation lors de la création de l'objet.

Le fetch type `LAZY` permet de charger les données de la relation lors de l'appel de l'objet.

Dans l'exemple ci-dessous, le fetch type est `EAGER`, donc les données de la relation sont chargées lors de la création de l'objet.



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToOne(fetch = FetchType.EAGER)
    private Adresse adresse;
    // constructeur, getters, setters
}

@Entity
public class Adresse
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String rue;
    private String ville;
    private String codePostal;
    @OneToOne(mappedBy = "adresse")
    private Personne personne;
    // constructeur, getters, setters
}
```



Attribut cascade



L'attribut cascade permet de définir les actions à effectuer sur les entités liées.

Cet attribut se définit avec l'annotation `@OneToOne(cascade = CascadeType.ALL)` ou `@OneToOne(cascade = CascadeType.PERSIST)`.

Dans l'exemple ci-dessous, l'attribut cascade est `ALL`, donc toutes les actions sur l'entité sont effectuées sur les entités liées.



Exemple de cascade



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToOne(cascade = CascadeType.ALL)
    private Adresse adresse;
    // constructeur, getters, setters
}

@Entity
public class Adresse
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String rue;
    private String ville;
    private String codePostal;
    @OneToOne(mappedBy = "adresse")
    private Personne personne;
    // constructeur, getters, setters
}
```



Attribut override



On peut utiliser l'attribut override pour modifier le nom de la colonne dans la table.

Cet attribut se définit avec l'annotation `@OneToOne(override = @AttributeOverride(name = "nomColonne", column = @Column(name = "nomColonne")))`.



Exemple de override



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToOne(override = @AttributeOverride(name = "nomColonne", column = @Column(name = "newNomColonne")))
    private Adresse adresse;
    // constructeur, getters, setters
}
```



Attribut embedded



L'attribut embedded permet de définir une relation 1 - 1, entre deux entités.

Cet attribut se définit avec l'annotation `@Embedded`.

Ces deux entités sont liées par une relation one to one.

Ils sont stockés dans la même table, dans la même ligne.

id	nom	prenom	age	adresse
---	-----	-----	---	-----
1	Dupont	Jean	25	1 rue de la paix 75000 Paris
2	Durand	Marie	30	2 rue de la paix 75000 Paris
3	Martin	Paul	35	3 rue de la paix 75000 Paris



Exemple de embedded



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @Embedded(override = @AttributeOverride(name = nb, column = @Column(name = "nbRue")))
    private Adresse adresse;
    // constructeur, getters, setters
}

@Entity
public class Adresse
{
    @column(name = "nb")
    private String rue;
    @column(name = "ville")
    private String ville;
    @column(name = "codePostal")
    private String codePostal;
    // constructeur, getters, setters
}
```



Relation one to many



La relation one to many permet de définir une relation entre deux entités.

L'entité A peut avoir plusieurs entités B.

L'entité B appartient à une seule entité A.

Exemple : Une femme peut avoir plusieurs enfants, Un enfant n'aura qu'une seule mère.



Exemple one to many



```
@Entity
public class Mother
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToMany
    private List<Child> children;
    // constructeur, getters, setters
}

@Entity
public class Child
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @ManyToOne
    private Mother mother;
    // constructeur, getters, setters
}
```



Relation many to many



La relation many to many permet de définir une relation entre deux entités.

Chaque entité peut avoir plusieurs entités.

Exemple : Un étudiant peut avoir plusieurs cours, Un cours peut avoir plusieurs étudiants.



Exemple many to many



```
@Entity
public class Student
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @ManyToMany
    private List<Course> courses;
    // constructeur, getters, setters
}

@Entity
public class Course
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    @ManyToMany
    private List<Student> students;
    // constructeur, getters, setters
}
```



Relation unidirectionnelle



Pour configurer une relation unidirectionnelle, il faut utiliser l'annotation `@OneToMany` dans l'entité qui possède la relation. L'annotation `@OneToMany` prend en paramètre la relation de l'autre entité.



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToMany(mappedBy = "personne")
    private List<Adresse> addresses;
    // constructeur, getters, setters
}

@Entity
public class Adresse
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String rue;
    private String ville;
    private String codePostal;
    @ManyToOne
    private Personne personne;
    // constructeur, getters, setters
}
```



Relation bidirectionnelle



Pour configurer une relation bidirectionnelle, il faut utiliser l'annotation `@OneToMany` et `@ManyToOne` dans les deux entités. L'annotation `@OneToMany` prend en paramètre la relation de l'autre entité, et l'annotation `@ManyToOne` prend en paramètre la relation de cette entité.



```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String nom;
    private String prenom;
    private Integer age;
    @OneToMany(mappedBy = "personne")
    private List<Adresse> addresses;
    // constructeur, getters, setters
}

@Entity
public class Adresse
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String rue;
    private String ville;
    private String codePostal;
    @ManyToOne
    private Personne personne;
    // constructeur, getters, setters
}
```



Tp One To Many