JAVA ORM JPA
Hibernate

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JAVA

ORM JPA Hibernate

Sommaire

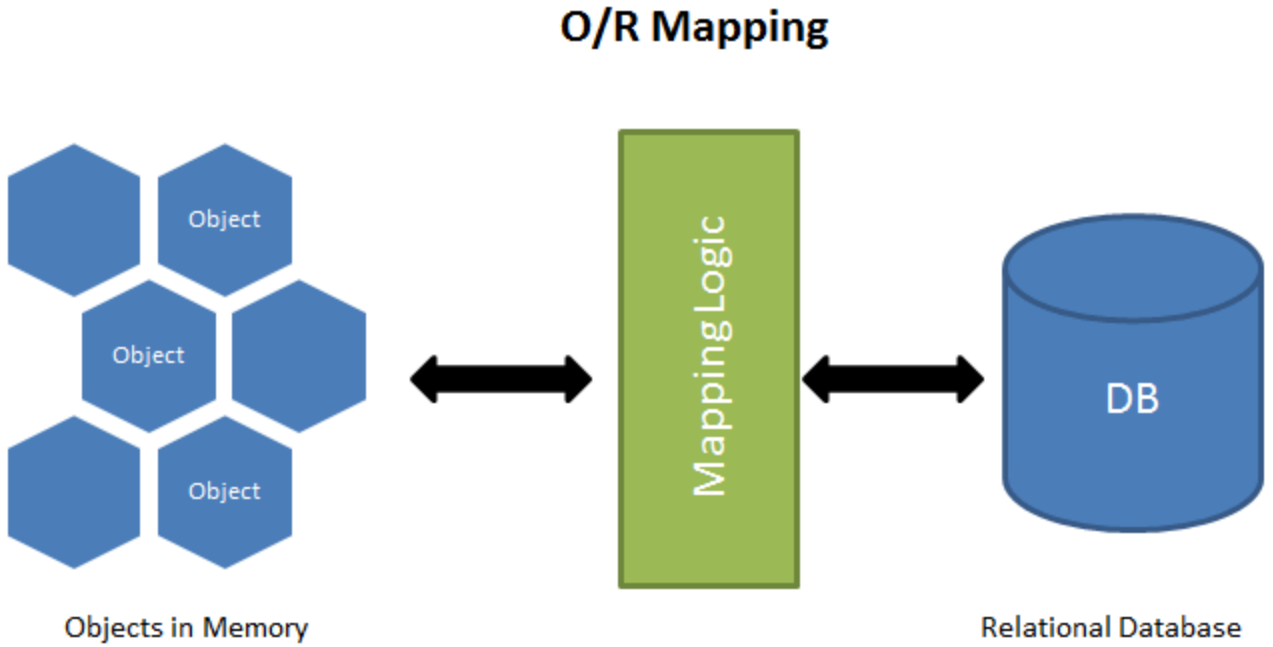
ORM , Hibernate, JPA ?

La **persistance de données** en Java est un **concept fondamental** dans le **développement d'applications**, particulièrement lorsque l'on travaille avec des **bases de données** relationnelles. Pour **simplifier** la gestion et l'**interaction avec la base de données**, les développeurs utilisent souvent des outils tels que **Hibernate** et **JPA (Java Persistence API)** en conjonction avec des **ORM (Object-Relational Mapping)**.

Commençons par définir ces trois termes.

ORM

ORM (Object-Relational Mapping) : Il s'agit d'une technique de programmation qui permet de **mapper des objets de type orienté objet** sur des **données stockées dans une base de données relationnelle**. En d'autres termes, cela permet de **manipuler des objets en Java comme s'ils étaient des lignes de données dans une table de base de données**, et vice versa, **sans avoir à écrire de requêtes SQL manuellement**.



ORM

Il existe plusieurs ORM disponibles.

Hibernate

Hibernate est un **framework ORM populaire** en Java qui **simplifie la manipulation des données** en fournissant une interface orientée objet pour **interagir avec la base de données**. Il permet de **représenter des objets Java comme des enregistrements dans la base de données**, et inversement, sans avoir besoin de coder des requêtes SQL complexes.

JPA

JPA (Java Persistence API) : **JPA** est une **spécification Java** qui définit une **interface commune** pour la **gestion de la persistance** des données. Elle fournit un **ensemble de classes et d'annotations standardisées** pour **mapper des objets** Java sur une **base de données** relationnelle. **Hibernate est l'une des implémentations de JPA les plus populaires**, mais d'autres implémentations existent également, telles que EclipseLink et Apache OpenJPA.

ORM , Hibernate, JPA conclusion

En résumé **Hibernate est une implémentation d'ORM qui se conforme à la spécification JPA**. Cela signifie que **Hibernate** implémente les **interfaces et les annotations définies par JPA**, ce qui permet aux développeurs d'utiliser les fonctionnalités standardisées de **JPA** pour **gérer la persistance des données** dans leurs applications Java. En utilisant **Hibernate** en tant qu'**implémentation JPA**, les développeurs bénéficient à la fois des fonctionnalités spécifiques de **Hibernate** et de la portabilité offerte par **JPA**, ce qui **facilite le développement d'applications robustes et évolutives**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JPA

Java Persistence API

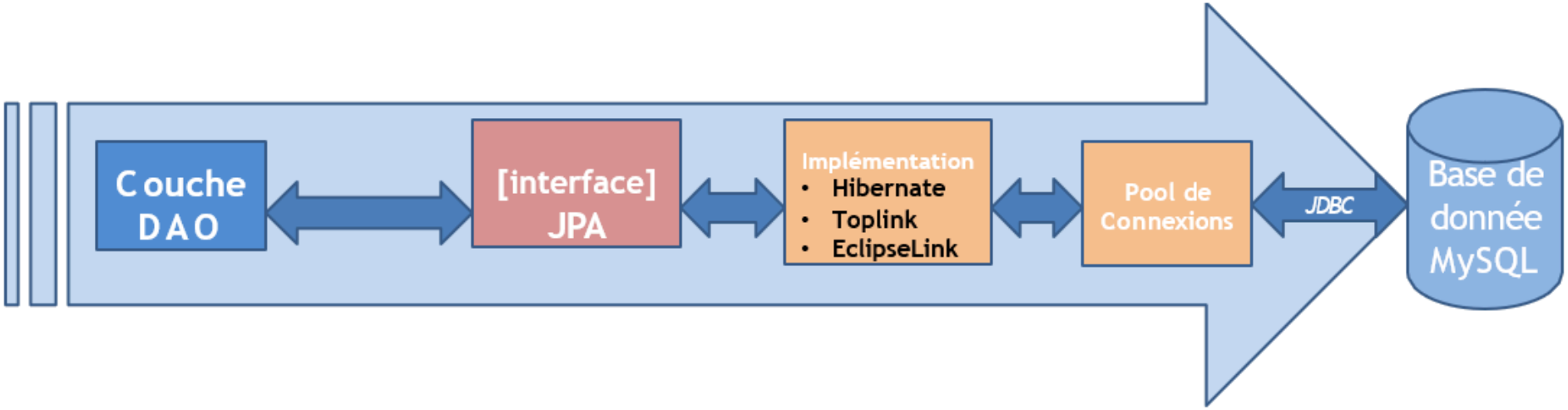
Introduction JPA

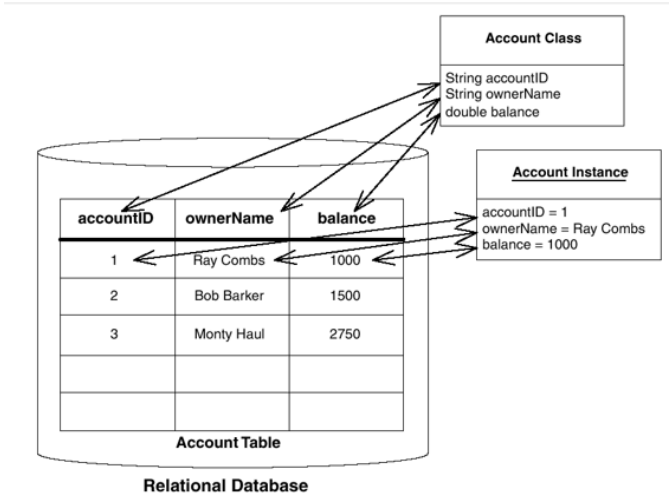
Devant le **succès** des **Frameworks ORM**, Sun a décidé de **standardiser une couche ORM** via une **spécification** appelée **JPA** apparue en même temps que **Java 5**.

La **spécification JPA** est un **ensemble d'interface** du **package javax.persistence** qui contient entre autre :

- javax.persistence.Entity;
- javax.persistence.EntityManagerFactory;
- javax.persistence.EntityManager;
- javax.persistence.EntityTransaction;

Introduction JPA





Introduction JPA

Ses **principaux avantages** sont les suivants :

- **JPA** peut être utilisé par **toutes les applications Java, Java SE ou Java EE**.
- **Mapping O/R (objet-relationnel)** avec les **tables de la BD**, **facilitée** par les **Annotations**.
- Un **langage de requête objet standard JPQL** pour la **récupération des objets**,

Les Entités / Entity

Les **classes** dont les **instances** peuvent être **persistantes** sont appelées des **entités** dans la **spécification de JPA**.

Le développeur indique qu'**une classe est une entité** en lui associant l'**annotation @Entity**.

Il faut importer **javax.persistence.Entity** dans les classes entités.

Les **entités** dans les spécifications de l'**API Java Persistence** permettent d'**encapsuler les données d'une occurrence d'une ou plusieurs tables**.

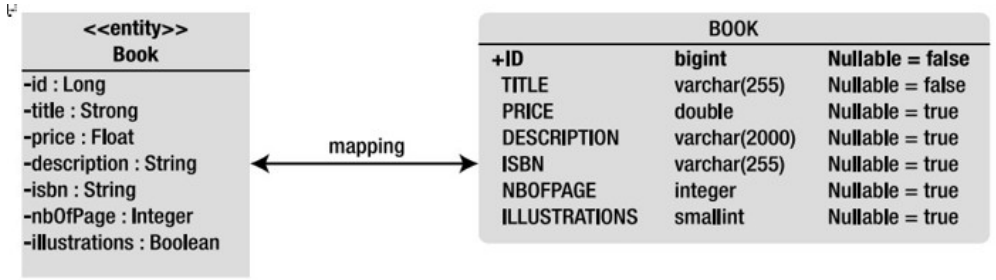
Ce sont de simples **POJO**.

Un **POJO** est une **classe Java** qui n'implémente aucune interface particulière ni n'hérite d'aucune classe mère spécifique.

Classes entités : Entity Class

- Une **classe entité** doit **posséder** un **attribut** qui représente la **clé primaire** dans la BD (**annotation @Id**)
- Elle doit avoir un **constructeur sans paramètre** protected ou public **sans argument** et la classe de l'entité doit **obligatoirement** être **marquée** avec **l'annotation @Entity**. Cette annotation possède un attribut optionnel nommé name qui permet de préciser le nom de l'entité dans les requêtes. Par défaut, ce nom est celui de la classe de l'entité.
- Elle ne doit pas être final
- Aucune méthode ou champ persistant ne doit être final
- Une entité peut être une classe abstraite mais elle ne peut pas être une interface

Entity / Entités exemple :




```java
@Entity
@Table(name = "BOOK")
public class Book {
  @Id  @GeneratedValue
  private Long id;
  @Column(name = "TITLE",nullable = false)
  private String title;
  private Float price;
  @Basic(fetch = FetchType.LAZY)
  @Column(length = 2000)
  private String description;
  private String isbn;
  private Integer nbOfPage;
  private Boolean illustrations;

//Les Getters et les Setters
}
```

Les Annotations

Annotation	Rôle
@Table	Préciser le nom de la table concernée par le mapping
@Column	Associé à un getter, il permet d'associer un champ de la table à la propriété
@Id	Associé à un getter, il permet d'associer un champ de la table à la propriété en tant que clé primaire
@GeneratedValue	Demander la génération automatique de la clé primaire au besoin
@Basic	Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
@Transient	Demander de ne pas tenir compte du champ lors du mapping

L'annotation @Id

- Il faut obligatoirement définir une des propriétés de la classe avec l'annotation @Id pour la déclarer comme étant la clé primaire de la table.
- Cette annotation peut marquer soit le champ de la classe concernée soit le getter de la propriété.
- L'utilisation de l'un ou l'autre précise au gestionnaire s'il doit se baser sur les champs ou les getter pour déterminer les associations entre l'entité et les champs de la table. La clé primaire peut être constituée d'une seule propriété ou composées de plusieurs propriétés qui peuvent être de type primitif ou chaîne de caractères

L'annotation @GeneratedValue

L'annotation @GeneratedValue définit la stratégie de génération automatique de l'identifiant :

- **Strategy = GenerationType.AUTO** : Hibernate produit lui même la valeur des identifiants grâce à une table hibernate_sequence . Le générateur AUTO est le type préféré pour les applications portables.
- **Strategy GenerationType.IDENTITY** Hibernate s'appuie alors sur le mécanisme propre au SGBD pour la production de l'identifiant.l'identifiant.(Dans le cas de MySQL, c'est l'option AUTO INCREMENT, dans le cas de postgres ou Oracle, c'est une séquence).
- **Strategy GenerationType.TABLE**: Hibernate utilise une table dédiée qui stocke
  les clés des tables générées, l'utilisation de cette stratégie nécessite l'utilisation de l'annotation @TableGenerator.
- **Strategy GenerationType.SEQUENCE**: Hibernate utilise un mécanisme nommé séquence proposé par certaines bases de données notamment celles d'Oracle. L'utilisation de la stratégie « SEQUENCE » nécessite l'utilisation de l'annotation @SequenceGenerator.

L'annotation @Table

- Permet de lier l'entité à une table de la base de données. Par défaut, l'entité est liée à la table de la base de données correspondant au nom de la classe de l'entité. Si ce nom est différent alors l'utilisation de l'annotation @Table est obligatoire.

Attributs	Rôle
name	Nom de la table
catalog	Catalogue de la table

L'annotation @Column

- Permet d'associer un membre de l'entité à une colonne de la table. Par défaut, les champs de l'entité sont liés aux champs de la table dont les noms correspondent. Si ces noms sont différents alors l'utilisation de l'annotation @Column est obligatoire.

Attributs	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi−table
unique	Indique si la colonne est unique
Nullable	Indique si la colonne est nullable
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type insert
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type update

Contexte de persistance

Le **contexte de persistance** contient **l'ensemble des instances d'entités** gérées a un instant donné.

- Gérées par qui ?
  - Le **gestionnaire d'entités** : **EntityManager**

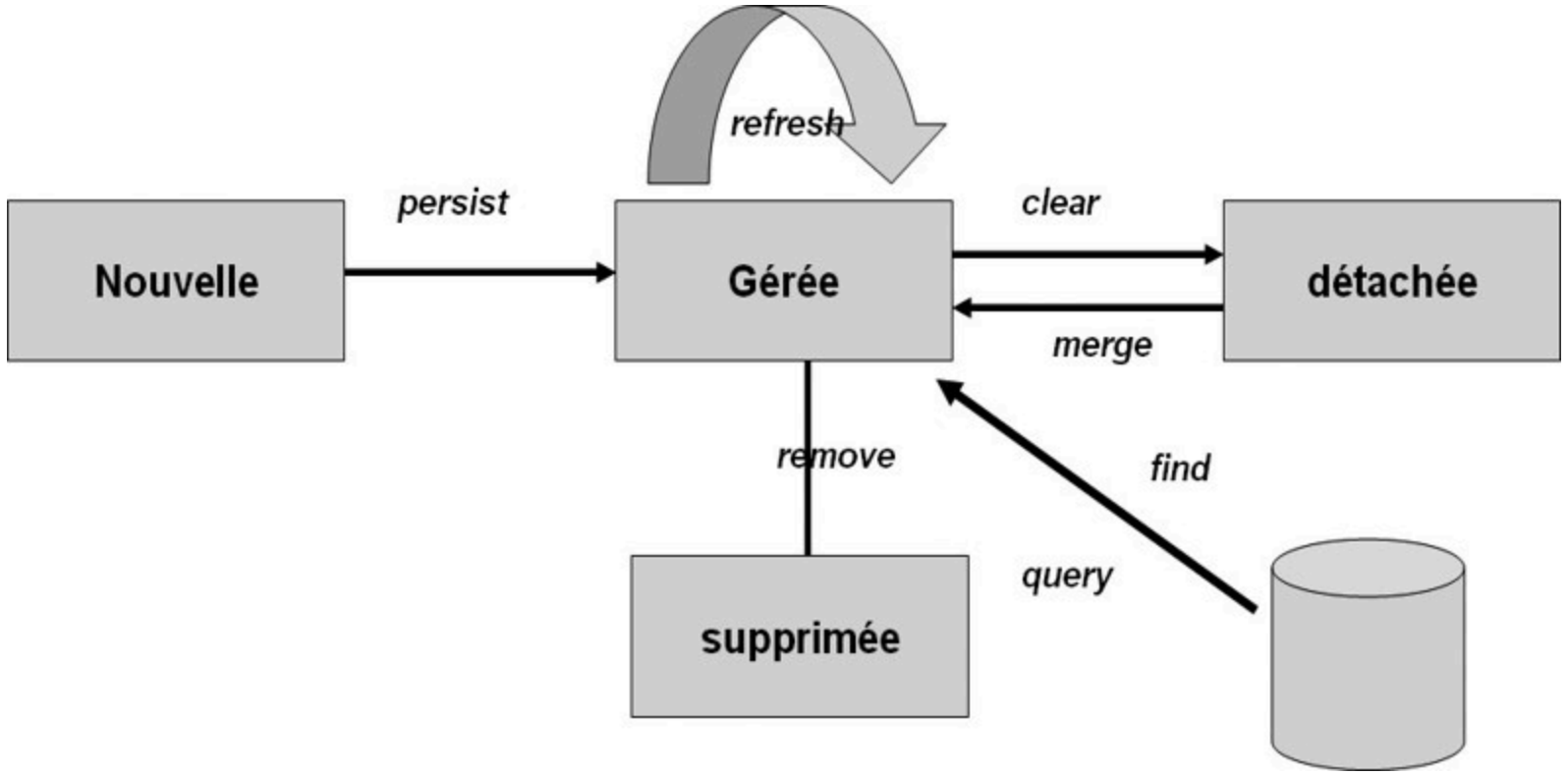
Ce **contexte** peut donc être considéré comme un cache de premier niveau : c'est un espace réduit ou le **gestionnaire stocke les entités avant d'écrire son contenu dans la base de données**

Opérations prises en charge par le gestionnaire d'entité

Opération	Description
persist()	Insére l'état d'une entité dans la base de données. Cette novelle entité devient alors une entité gérée.
remove()	Supprime l'état de l'entité géréé est ses données correspondante da la base
refresh()	Synchronise l'état de l'entité à partir de la base, les données de la BD étant copiées dans l'entité
merge()	Synchronise les états des entités "détachées" avec le PC. La méthode retourne une entité gérée qui a la même identité dans la base que l'entité passée en paramètre, bien que ce ne soit pas le même objet.
find()	Exécute une requête simple de recherche de clé
CreateQuery()	Crée une instance de requête en utilisant le langage JPQL.
createNamedQuery()	Crée une instance de requête spécifique.
createNativeQuery()	Crée une instance de requête SQL.
contains()	Spécifie si l'entité est managée par le PC.
flush()	Toutes les modifications effectuées sur les entités du contexte de persistance gérées par le gestionnaire d'entités sont enregistrées dans la BD lors d'un flush du gestionnaire.

NB : Un flush() est automatiquement effectué au moins à chaque commit de la transaction en cours.

Cycle de vie d'une instance d'entité



Obtention d'une fabrique EntityManagerFactory

L' interface EntityManagerFactory permet d'obtenir une instance de l'objet EntityManager.

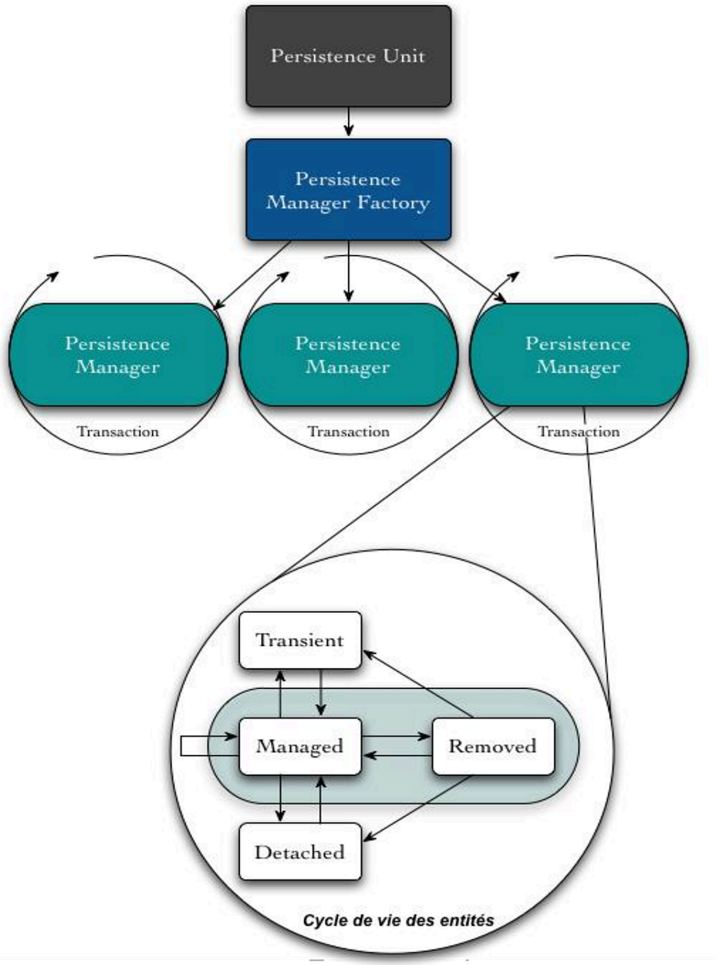
```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
```

« jpa » est le nom de l'unité de persistance, définie dans le fichier de configuration de la couche JPA META-INF/persistence.xml.

- Création du gestionnaire d'entité EntityManager

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("jpa");
EntityManager em = emf.createEntityManager() ;
```

Fonctionnement de JPA



- La **Persistence Unit** : organise les meta données qui définissent le mapping entre les entités et la base de donnée dans le **fichier de configuration METAINF/persistence.xml**
- La **Persistence Manager Factory** récupère ces metas données de la Persistence Unit et les interprètent pour créer des Persistence Manager (EntityManager).
- Le **Persistence Mangager (EntiyManager)** gère les échanges entre le code et la base de donnée, c'est à dire le cycle de vie des **entités**.
- Enfin, les opérations du EntityManager est englobé dans une **Transaction**.

Ajout de dépendances

Pour pouvoir utiliser JPA fourni par Hibernate, nous devrons ajouter les dépendances nécessaires. Pour un projet maven :

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
    <!-- pour mysql -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
    <!-- pour postgresql -->
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.6.0</version>
    </dependency>
</dependencies>
```

persistence.xml

Le fichier persistence.xml est un élément crucial pour configurer et déployer des applications Java utilisant JPA pour la persistance des données.Il est une composante essentielle dans le contexte de Java Persistence API (JPA). Voici une brève description de son rôle :

- **Définition des paramètres de persistance** : Le fichier persistence.xml est utilisé pour définir les paramètres de persistance pour une unité de persistance dans une application Java EE ou SE.
- **Configuration des entités persistantes** : Il permet de déclarer les classes d'entités (entités Java) qui doivent être persistées dans la base de données. Ces classes sont souvent annotées avec des annotations JPA telles que @Entity, @Table, etc.
- **Configuration du fournisseur de persistance** : On y spécifie le fournisseur de persistance JPA que l'application utilisera. Cela peut être Hibernate, EclipseLink, OpenJPA, etc. Ces fournisseurs se chargent de la gestion de la persistance des objets Java dans la base de données.
- **Gestion des transactions** : Le fichier persistence.xml peut également contenir des configurations relatives à la gestion des transactions, telles que le type de transaction (JTA ou resource-local), le gestionnaire de transactions à utiliser, etc.

Exemple persistence.xml

```xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="jpa_demo">
        <!-- balise description optionnelle mais utile-->
        <description>
            Ici je peux mettre la description de mon fichier persistence.xml
        </description>
        <!-- <provider>: Cet élément facultatif spécifie le fournisseur de persistance JPA à utiliser.
      Si cet élément est omis, le fournisseur par défaut configuré dans l'environnement sera utilisé.-->
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <!--        Ici spécifier les classes d'entités qui seront gérées par cette unité de persistance-->
        <class>org.example.entity.Person</class>
        <class>org.example.entity.Adress</class>

        <properties>
            <!--            Spécifie le nom de la classe du pilote JDBC utilisé pour la connexion à la base de données.-->
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
            <!--            Spécifie l'URL de connexion à la base de données.-->
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/jpa_demo3" />
            <!--            Spécifie le nom d'utilisateur utilisé pour se connecter à la base de données.-->
            <property name="javax.persistence.jdbc.user" value="root" />
            <!--            Spécifie le mot de passe utilisé pour se connecter à la base de données.-->
            <property name="javax.persistence.jdbc.password" value="test" />
            <property name="javax.persistence.schema-generation.database.action" value="update"/>

        </properties>

    </persistence-unit>

</persistence>
```

EntityManager

- Le gestionnaire d'entités via la classe EntityManager est l'interlocuteur principal pour le développeur.
- Les interactions entre la base de données et les beans entité sont assurées par un objet de type javax.persistence.EntityManager : il permet de lire et rechercher des données mais aussi de les mettre à jour (ajout, modification, suppression).
- L'EntityManager est donc au coeur de toutes les actions de persistance. Il fournit les méthodes pour gérer les entités :
  - les rendre persistantes,
  - les supprimer de la base de données
  - retrouver leurs valeurs dans la base
  - etc...

Cycle de vie d'un EntityManager

- La méthode **createEntityManager()** de la classe EntityManagerFactory créé un Entity Manager
- L'Entity Manager est supprimé avec la méthode **close()** de la classe EntityManager, il ne sera plus possible de l'utiliser ensuite.

Fabrique de l'EntityManager

- La classe Persistence permet d'obtenir une fabrique de gestionnaire d'entités par la méthode createEntityManagerFactory
- 2 variantes surchargées de cette méthode :
  - 1 seul paramètre qui donne le nom de l'unité de persistance (définie dans le fichier persistence.xml)
  - Un 2ème paramètre de type Map qui contient des valeurs qui vont écraser les propriétés par défaut contenues dans persistence.xml

Méthodes de EntityManager

- **void flush()**

Toutes les modifications effectuées sur les entités du contexte de persistance gérées par l'EntityManager sont enregistrées dans la BD

- **void persist(Object entité)**

Une entité nouvelle devient une entité gérée, l'état de l'entité sera sauvegardé dans la BD au prochain flush ou commit.

- **void remove(Object entité)**

Une entité gérée devient supprimée, les données correspondantes seront supprimées de la BD

Méthodes de EntityManager

- **void lock(Object entité, LockModeType lockMode)**

Le fournisseur de persistance gère les accès concurrents aux données de la BD représentées par les entités avec une stratégie optimiste, lock permet de modifier la manière de gérer les accès concurrents à une entité

- **void refresh(Object entité)**

L'EntityManager peut synchroniser avec la BD une entité qu'il gère en rafraichissant son état en mémoire avec les données actuellement dans la BD.Utiliser cette méthode pour s'assurer que l'entité a les mêmes données que la BD.

- **<T> T find(Class<T> classeEntité,Object cléPrimaire)**

La recherche est polymorphe : l'entité récupérée peut être de la classe passée en paramètre ou d'une sous-classe (renvoie null si aucune entité n'a l'identificateur passé en paramètre)

Utilisation de EntityManager pour la création d'une occurrence

Pour insérer une nouvelle entité dans la base de données, il faut :

- Instancier une occurrence de la classe de l'entité
- Initialiser les propriétés de l'entité
- Définir les relations de l'entité avec d'autres entités, et au besoin
- Utiliser la méthode persist() de l'EntityManager en passant en paramètre l'entité

Exemple

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

public class TestJPA {
    public static void main(String[] args) {
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transac = em.getTransaction()
        transac.begin();
        Personne nouvellePersonne = new Person();
        nouvellePersonne.setId(4);
        nouvellePersonne.setNom("nom4");
        nouvellePersonne.setPrenom("prenom4");
        em.persist(nouvellePersonne);
        transac.commit();
        em.close();
        emf.close();
    }
}
```

Utilisation de EntityManager pour rechercher des occurrences

Pour effectuer des recherches de données, l'EntityManager propose deux mécanismes :

- La **recherche** à partir de la **clé primaire**
- La **recherche** à partir d'une **requête utilisant une syntaxe dédiée**

Pour la recherche par **clé primaire**, la classe EntityManager possède les méthodes **find()** et **getReference()** qui attendent toutes les deux en paramètres un objet de type Class représentant la classe de l'entité et un objet qui contient la valeur de la **clé primaire**.

exemple avec find()

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

public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        Personne personne = em.find(Personne.class, 4);
        if(personne != null){
          System.out.println("Personne.nom = "+ personne.getNom());
        }
        em.close();
        emf.close();
    }
}
```

exemple avec getReference()

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

public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        try {
          Personne personne = em.find(Personne.class, 5);
          System.out.println("Personne.nom = "+ personne.getNom());
        } catch (EntityNotFoundException e) {
          System.out.println("Personne non trouvée");
        }
        em.close();
        emf.close();
    }
}
```

Queries

Il est possible de rechercher des données sur des critères plus complexes que la simple identité. Les étapes sont alors les suivantes :

- **Décrire ce qui est recherché (langage JPQL)**
- **Créer une instance de type Query**
- **Initialiser la requête (paramètres, pagination)**
- **Lancer l'exécution de la requête**

JPA introduit le **JPQL (Java Persistence Query Language)**, qui est, tout comme le EJBQL ou encore le HQL, un **langage de requête** du modèle objet, **basé sur SQL**.

Interface Query

Pour représenter une requête, on utilise une instance de Query obtenue via la classe EntityManager à l'aide de méthodes dédiées :

- **createQuery()**
- **createNativeQuery()**
- **createNamedQuery()**

L'objet Query

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

public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createQuery("select p from Personne p where p.nom='nom2'");
        Personne personne = (Personne) query.getSingleResult();
        if(personne ==  null){
          System.out.println("Personne non trouvée");
        }else {
            System.out.println("Personne.nom = "+ personne.getNom());
        }
        em.close();
        emf.close();
    }
}
```

L'objet Query et la méthode getResultList()

```
import java.util.list;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;
import javax.persistence.Persistence;

public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createQuery("select p.nom from Personne p where p.id > 2");
        List noms = query.getResultList();
        for(Object nom : noms){
          System.out.println("nom = "+ nom);
        }
        em.close();
        emf.close();
    }
}
```

L'objet Query et les paramètres nommés avec setParameter()

```
import java.util.list;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;
import javax.persistence.Persistence;

public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createQuery("select p.nom from Personne p where p.id > :id");
        query.setParameter("id",1)
        List noms = query.getResultList();
        for(Object nom : noms){
          System.out.println("nom = "+ nom);
        }
        em.close();
        emf.close();
    }
}
```

Modifier une occurence avec le EntityManager

Pour modifier une entité existante dans la base de données, il faut :

- Obtenir une instance de l'entité à modifier (par recherche sur la clé primaire ou l'exécution d'une requête)
- Modifier les propriétés de l'entité
- Selon le mode de synchronisation des données de l'EntityManager, il peut être nécessaire d'appeler la méthode flush() explicitement

Exemple Modifier une occurence avec le EntityManager

```
public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transac = em.getTransaction();
        transac.begin();
        Query query = em.createQuery("select p.nom from Personne p where p.nom='nom2'");
        Personne personne = (Personne) query.getSingleResult();
        if(personne == null){
          System.out.println("Personne non trouvée");
        }else{
          System.out.println("Personne.prenom = "+ personne.getPrenom());
          personne.setPrenom("prenom2 modifié");
          em.flush();
          personne = (Personne) query.getSingleResult();
          System.out.println("Personne.prenom = "+ personne.getPrenom());
        }
        transac.commit();
    }
}
```

Supprimer une occurence avec le EntityManager

Pour supprimer une entité existante dans la base de données:

- Obtenir une instance de l'entité à supprimer (par recherche sur la clé primaire ou l'exécution d'une requête)
- Appeler la méthode remove() de l'EntityManager en lui passant en paramètre l'instance de l'entité

Exemple Supprimer une occurence avec le EntityManager

```
public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transac = em.getTransaction();
        transac.begin();
        Personne personne = em.find(Personne.class,4);
        if(personne == null){
          System.out.println("Personne non trouvée");
        }
        else {
          em.remove(personne);
        }
        transac.commit();
        em.close();
        emf.close();
    }
}
```

Exemple rafraîchir les données d'une occurence

```
public class TestJPA {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
        EntityManager em = emf.createEntityManager();
        EntityTransaction transac = em.getTransaction();
        transac.begin();
        Personne personne = em.find(Personne.class,4);
        if(personne == null){
          System.out.println("Personne non trouvée");
        }
        else {
          em.refresh(personne);
        }
        transac.commit();
        em.close();
        emf.close();
    }
}
```

Queries : NamedQueries

Peut être mise dans n'importe quelle entité, mais on choisira le plus souvent l'entité qui correspond à ce qui est renvoyé par la requête.
On peut sauvegarder des gabarits de requête dans nos entités. Ceci permet :

- La réutilisation de la requête
- D'externaliser les requête du code.

Exemple Queries : NamedQueries

```
@Entity
@NamedQueries({
    @NamedQuery(name = "User.findAllUsers", query = "SELECT u FROM User u")
})
public class User {
    // attributs et méthodes de l'entité User
}
```

```
import javax.persistence.EntityManager;
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.util.List;

public class ExampleDAO {

    EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
      EntityManager em = emf.createEntityManager();

    public List<User> findAllUsers() {
        Query query = em.createNamedQuery("User.findAllUsers");
        return query.getResultList();
    }
}
```

Queries : NativeQueries

Les NativeQueries sont une façon de faire des requête en SQL natif. Elle sert principalement à avoir plus de contrôle sur les requêtes à la base de donnée.

```
public class MyService {
  public void myMethod(){
    ...
    List results = em.createNativeQuery("SELECT * FROM MyPojo",MyPojo.class).getResultList();
    ...
  }
}
```

Relations

- Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations.
- Ces relations sont transposées dans les liaisons que peuvent avoir les différentes entités correspondantes.
- **4 types de relations** à définir entre les entités de la JPA :
  - **1–1 (One to One)**
  - **1–n (Many to One)**
  - **1–n (One to Many)**
  - **n–n (Many to Many)**

Relations

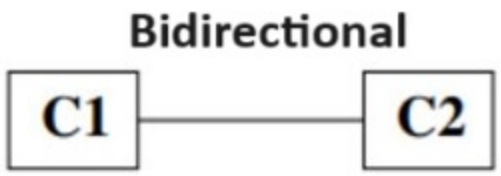
Les relations utilisables dans le monde relationnel et le monde objet sont cependant différentes:

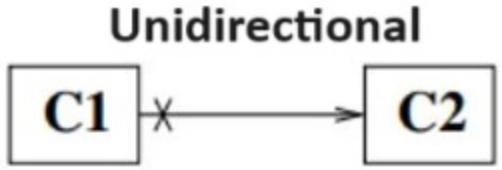
- Les relations du monde objets :
  - Elles sont réalisées via des références entre objets
  - Elles peuvent mettre en œuvre l'héritage et le polymorphisme
- Les relations du monde relationnel :
  - Elles sont gérées par des clés étrangères et des jointures entre tables.

La navigabilité

Il s'agit de la capacité à naviguer facilement entre les objets associés à l'aide des relations définies dans le modèle objet-relationnel. En JPA, il existe deux types de navigabilité :

- **Bidirectional** : Navigabilité dans les deux sens, les objets peuvent traverser les deux coté de la relation:
  - C1 a un attribut de type C2 et C2 a un attribut de type C1
- **Unidirectional** : Spécification de la navigabilité : Orientation de la navigabilité, les objets peuvent uniquement traverser d'un côté de la relation, par exemple:
  - C1 a un attribut du type de C2, mais pas l'inverse





La direction d’une associations

La réduction de la portée de l'association est souvent réalisée en phase d'implémentation. C ’est le développeur qui est responsable de la gestion de la navigabilité des objets, c’est à dire les deux bouts de l’association.
Un des bouts est le propriétaire de l'association (Master-Slave) :

- pour les associations autres que many to many, **le bout propriétaire** correspond à **la table qui contient la clé étrangère** qui traduit l’association.
- pour les associations many to many, le développeur choisit le bout propriétaire (de manière arbitraire).
- l’autre bout (**non propriétaire**) est qualifié par **l’attribut mappedBy** qui donne le nom de l’association correspondante dans le bout propriétaire :
  - @annotation(mappedBy= "ObjetAutreBout").

Récupérations d'associations

Dans la définition d'une association en JPA, "fetch" est une caractéristique qui spécifie comment les données associées doivent être récupérées de la base de données lorsqu'une requête est exécutée sur l'entité principale.

- immédiatement ("eager") fetch = FetchType.EAGER
- à la demande ("lazy") fetch=FetchType.LAZY

Le paramètre fetch peut être positionné à FetchType.LAZY ou à FetchType.EAGER.Cela permet de contrôler le comportement de chargement des données liées à l'entité principale et peut avoir un impact sur les performances de l'application.

Relations un à un (One to One)

Cette relation peut se traduire de plusieurs manières dans la base de données :

- Une seule table qui contient les données de l'étudiant et ses coordonnés.
- Deux tables, une pour les étudiants et une pour les coordonnés avec une clé primaire partagée
- Deux tables, une pour les étudiants et une pour les coordonnés avec une seule clé étrangère

Il y a plusieurs façons de traiter ce cas avec une ou deux tables dans la base de données et Hibernate :

- Deux tables et une relation One-to-One d'Hibernate
- Une seule table avec un Component d'Hibernate

Exemple Relations un à un (One to One) (deux tables avec une seul clé étrangére)

Dans l'exemple ci-dessous, chaque « étudiant » ne peut avoir qu'une seule «Coordonnées» et une « coordonnées » ne peut appartenir qu‘a un seul « étudiant ».

- C ’est la table étudiant qui contient la clé étrangère (Id_Crd)
- On va faire un mapping bidirectionnelle avec l’annotation @OneToOne et l’attribut mappedBy

Exemple Relations un à un (One to One) (deux tables avec une seul clé étrangére)

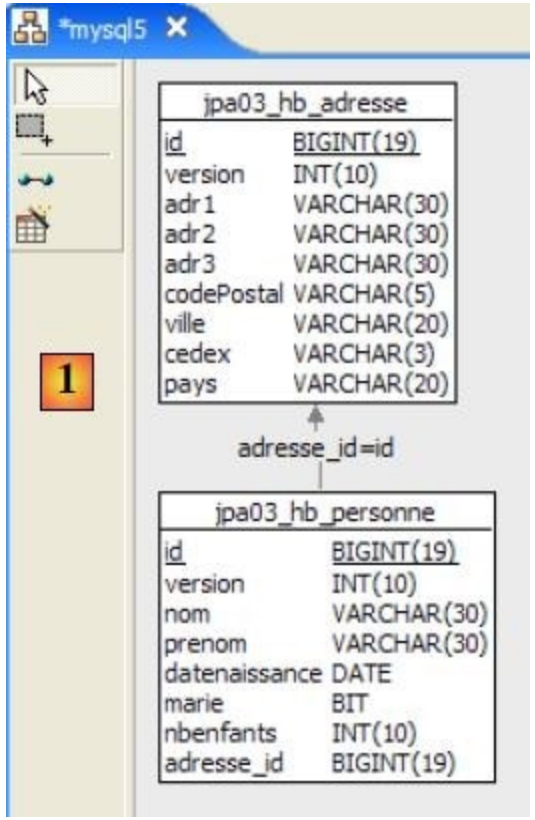
```java
@Entity
@Table(name="coordonnees")
public class Coordonnees {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="Id_Crd")
  private int id;
  @Column(name="adresse_Crd")
  private String adresse;
  @Column(name="ville_Crd")
  private String ville;
  @Column(name="email_Crd")
  private String email;
  @Column(name="tel_Crd")
  private String tel;
  @OneToOne(mappedBy="coordonnees")
  private Etudiant etudiant;
  //les constructeurs, les getters et les setters
}
```

```java
@Entity
@Table(name="etudiant")
public class Etudiant {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="Apoee_Etd",length=5)
  private int apogee;
  @Column(name="Nom_Etd", length=30)
  private String nom;
  @Column(name="Nom_Prenom", length=30)
  private String prenom;
  @OneToOne
  @JoinColumn(name="Id_Crd")
  private Coordonnees coordonnees;
  //les constructeurs, les getters et les setters
}
```

Exemple Relations un à un (One to One) (deux tables avec une seul clé étrangére) mise en pratique

```
public class MappingTest {
public static void main(String[] args) {
  Session session=HibernateUtil.getSessionFactory().openSession();
  Transaction tx=session.getTransaction(); tx.begin();
  Coordonnees crd=new Coordonnees("ENSA-agadir", "Agadir","0525008800", "yboukouchi@gmail.com");
  Etudiant etd1=new Etudiant(2020, "BOUKOUCHI","Youness",crd);
  session.save(crd);session.save(etd1);
  tx.commit();}
}
```

Dernier exemple Relations One to One




```java
@Entity
@Table(name = "jpa03_hb_personne")
public class Personne {
...
  @OneToOne(cascade = CascadeType.ALL, fetch=FetchType.LAZY)
  /* @OneToOne(cascade = {CascadeType.MERGE, CascadeType.PERSIST,
     CascadeType.REFRESH, CascadeType.REMOVE}, fetch=FetchType.LAZY) */
  @JoinColumn(name = "adresse_id", unique = true, nullable = false)
  private Adresse adresse;
...
}
```

```java
@Entity
@Table(name = "jpa03_hb_adresse")
public class Adresse{
...
/*n'est pas obligatoire*/
@OneToOne(mappedBy= "adresse", fetch = FetchType.EAGER)
private Personne personne;
...
}
```

Les composants (component)

Une entité existe par elle-même indépendamment de toute autre entité, et peut être rendue persistante par insertion dans la base de données, avec un identifiant propre.
Un composant est un objet sans identifiant, qui ne peut être persistant que par rattachement à une entité.
La notion de composant résulte du constat qu'une ligne dans une base de données peut parfois être décomposée en plusieurs sous-ensemble dotés chacun d'une logique autonome. Cette décomposition mène à une granularité fine de la représentation objet, dans laquelle on associe plusieurs objets à une ligne de la table.

- **@Embedded** nous indique que ce composant est embarqué.
- **@Embeddable** nous indique que cette classe sera utilisée comme composant.

Les composants (component) exemple

```java
@Entity
public class Agence {
  @Id
  private String code;
  private String libelle;
  @Embedded
  private Adresse adresse;
  public Agence() { }
  public Agence(String code, String libelle,
  Adresse adresse) {
  super();
  this.code = code;
  this.libelle = libelle;
  this.adresse = adresse;
  }
  //getters et setters
}
```

```java
@Embeddable
public class Adresse {
  private String ville;
  private String pays;
  private int codePostal;
  public Adresse() {}
  public Adresse() {}
  public Adresse(String ville, String pays, int codePostal){
    super();
    this.ville = ville;
    this.pays = pays;
    this.codePostal = codePostal;
  }
  // getters et setters
}
```

Associations 1-N et N-1

- les associations “un à plusieurs” et “plusieurs à un” annotations @OneToMany et @ManyToOne
- Représentée par une clé étrangère dans la table qui correspond au côté propriétaire (obligatoirement le côté Many)
- Exemple : Enseignant-Département, cette association peut se représenter de 3 manières :
  - dans une classe Enseignant, on place un lien vers le Département (unidirectionnel): @ManyToOne
  - dans une classe Département , on place des liens vers les Enseignants qu’il lui sont affectés(unidirectionnel): @OneToMany
  - Ou bien, on représente les liens des deux côtés (bidirectionnel) : mappedBy

Association Unidirectionnelle.

- **Enseignant est le maître de l'association.**

Dans la classe Enseignant, **@ManyToOne** nous indique qu'un objet lié nommé departement (clé étrangère) est de la classe Departement.
L'annotation **@JoinColumn** indique la clé étrangère dans la table Enseignant qui permet de rechercher le Département concerné.

```
@Entity
@Table(name="enseignant")
public class Enseignant {
  @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="Code_Ens")
  private int code;
  @Column(name="Nom_Ens", length=30)
  private String nom;
  @Column(name="Prenom_Ens", length=30)
  private String prenom;
  @ManyToOne
  @JoinColumn(name="Code_Dep")
  private Departement departement;
  public Enseignant() {}
  public Departement getDepartement() {
  return departement;}
  public void setDepartement(Departement departement)
  {this.departement = departement;}
  // les getters et les setters
}
```

Association Unidirectionnelle mise en pratique.

```
public class MappingTest {
public static void main(String[] args) {
  // TODO Auto-generated method stub
  Session session=HibernateUtil.getSessionFactory().openSession();
  Transaction tx=session.getTransaction();
  tx.begin();
  Enseignant ens = new Enseignant("Youness","Boukouchi");
  ens.setDepartement(dep);
  session.save(dep);
  tx.commit();
  }
}
```

Association Unidirectionnelle.

- **Département est le maître de l'association.**

Dans la classe Département, **@OneToMany** nous indique que la collection des objets liés nommé enseignats, est de la classe Ensaignant.

L'annotation **@JoinColumn** indique la clé étrangère dans la table Enseignant qui permet de rechercher le Département concerné.

On ajoute le getter et le setter de la collection.

On ajoute la méthode ajouterEnseignant pour lier un objet Enseignant au département.

```
@Entity
@Table(name="departement")
public class Departement {
  @Id
  @Column(name="Code_Dep", length=6)
  private String code;
  @Column(name="Libelle_Dep", length=50)
  private String libelle;
  @OneToMany
  @JoinColumn(name="Code_Dep")
  Set<Enseignant> enseignants =new HashSet<>();
  public Set<Enseignant> getEnseignants() {
  return enseignants;
  }
  public void setEnseignants(Set<Enseignant> enseignants) {
  this.enseignants = enseignants;
  }
  public void ajouterEnseignant(Enseignant ens){
  this.enseignants.add(ens);
  }
  //Les constructeurs
  //les getters et les setters
}
```

Association Unidirectionnelle mise en pratique.

```
public class MappingTest {
public static void main(String[] args) {
  Session session=HibernateUtil.getSessionFactory().openSession();
  Transaction tx=session.getTransaction();
  tx.begin();
  Enseignant ens=new Enseignant("Youness","Boukouchi");
  session.save(ens);
  Departement dep=new Departement("GINFO","Génie Informatique");
  dep.ajouterEnseignant(ens);
  session.save(dep);
  tx.commit();
  }
}
```

Associations bidirectionelle

Enseignant sera le propriétaire de l'association.

Dans la classe Enseignant, @ManyToOne nous indique qu'un objet lié nommé departement est de la classe Departement.

L'annotation @JoinColumn indique la clé étrangère dans la table Enseignant qui permet de rechercher le Département concerné.

Dans la classe Département, @OneToMany nous indique que la collection des objets liés nommé enseignants, est de la classe Enseignant.

L'attribut mappedBy nous indique l'attribut de mapping dans la classe Enseignant.

Associations bidirectionelle (entity)

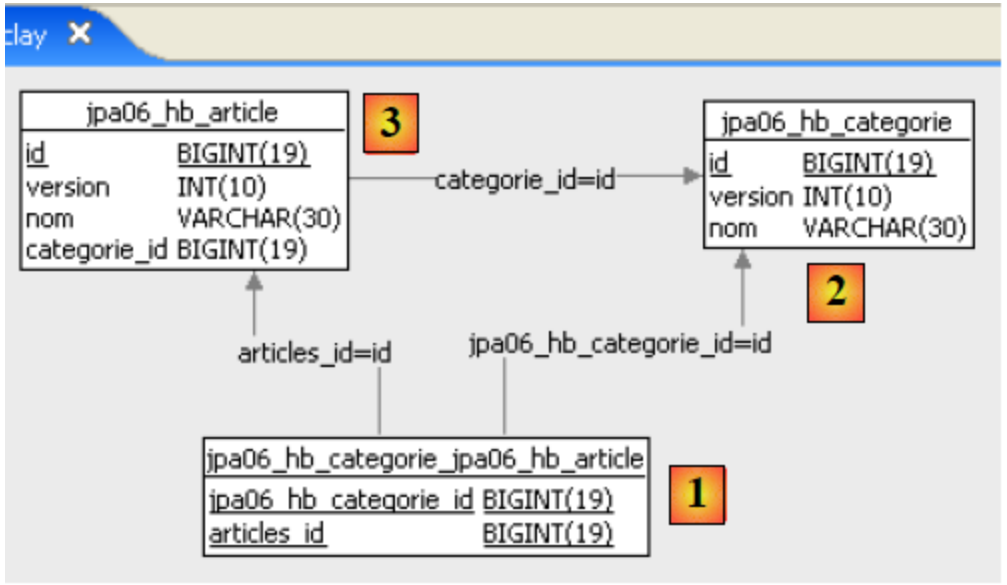
```java
@Entity
@Table(name="departement")
public class Departement {
  // les champs
  @OneToMany(mappedBy="departement")
  Set<Enseignant> enseignants =new HashSet<>();
  public Set<Enseignant> getEnseignants() {
  return enseignants;
  }
  public void setEnseignants(Set<Enseignant> enseignants) {
  this.enseignants = enseignants;
  }
  public Departement() {}
  // les getters et les setters
}
```

```java
@Entity
@Table(name="enseignant")
public class Enseignant {
  //les champs
  @ManyToOne
  @JoinColumn(name="Code_Dep")
  private Departement departement;
  public Departement getDepartement() {
  return departement;
  }
  public void setDepartement(Departement departement) {
    this.departement = departement;
  }
  public Enseignant() {}
  //les getters et les setters
}
```

Associations bidirectionelle mise en pratique

```
public class MappingTest {
  public static void main(String[] args) {
    Session session=HibernateUtil.getSessionFactory().openSession();
    Transaction tx=session.getTransaction();
    tx.begin();
    Enseignant ens1=new Enseignant("Youness","Boukouchi");
    session.save(ens1);
    Enseignant ens2=new Enseignant("Mohammed","Ahmed");
    session.save(ens2);
    Departement dep=new Departement("GINFO","Génie Informatique");
    ens1.setDepartement(dep);
    ens2.setDepartement(dep);
    session.save(dep);
    tx.commit();
    session.refresh(dep);
    System.out.println("size : "+dep.getEnseignants().size());
    session.close();
  }
```

Dernier exemple Relations One to Many




```java
@Entity
@Table(name = "jpa06_hb_categorie")
public class Personne {
...
// relation OneToMany non inverse (absence de mappedby) Categorie (one) -> Article (many)
// implémentée par une table de jointure Categorie_Article pour qu'a partir d'une catégorie
// on puisse atteindre plusieurs articles
  @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
  private Set<Article> articles = new HashSet<Article>();
...
}
```

Association many to many (plusieurs à plusieurs)

Quand on représente une association plusieurs-plusieurs en relationnel, l'association devient une table dont la clé est la concaténation des clés des deux entités de l'association.

La valeur par défaut du nom de la table association est la concaténation des 2 tables, séparées par _

Annotation @ManyToMany est représentée par une table association.

Deux cas se présentent :

- l'association n'est pas porteuse d'aucun attribut.
- l'association est porteuse des attributs (traitée comme @OneToMany dans les deux cotés de l'association).

Association many to many (plusieurs à plusieurs) unidirectionnelle

Exemple :

- Un enseignant enseigne plusieurs matières.
- Une matière est enseignée par plusieurs enseignants.

La classe Matière est le maître de l'association.

Dans la classe Matière, **@ManyToMany** nous indique que la collection des objets liés nommé enseignants, est de la classe Enseignant.

```
@Entity
@Table(name="matiere")
public class Matiere {
  @Id
  @Column(name="Code_Mat", length=6)
  private String code;
  @Column(name="Libelle_Mat", length=100)
  private String libelle;
  @ManyToMany
  private Set<Enseignant> enseignants=new HashSet<>();
  public Set<Enseignant> getEnseignants() {
  return enseignants;
  }
  public void setEnseignants(Set<Enseignant> enseignants) {
  this.enseignants = enseignants;
  }
  public void ajouterEnseigant(Enseignant ens){
  this.enseignants.add(ens);
  }
  //les constructeurs
  //les getters et les setters
}
```

Association many to many (plusieurs à plusieurs) unidirectionnelle mise en pratique

```
public class MappingTest {
  public static void main(String[] args) {
    Session session=HibernateUtil.getSessionFactory().openSession();
    Transaction tx=session.getTransaction();
    tx.begin();
    Enseignant ens1=new Enseignant("Youness","Boukouchi");
    session.save(ens1);
    Matiere mat1 = new Matiere("JEE","Java Entreprise Edition");
    mat1.ajouterEnseignant(ens1);
    Matiere mat2 = new Matiere("SOA","Architecture Orintée Services");
    mat2.ajouterEnseignant(ens1);
    session.save(mat1);
    session.save(mat2);
    tx.commit();
    System.out.println("size :"+mat1.getEnseignants().size());
    session.close();
  }
}
```

Association many to many (plusieurs à plusieurs) l'annotation @JoinTable

@JoinTable est une annotation utilisée en JPA pour spécifier une table d'association personnalisée entre deux entités dans une relation ManyToMany, et elle est généralement nécessaire pour définir la table de jointure.

@JoinTable à plusieurs attributs :

- attribut **name** donne le nom de la table association.
- attribut **joinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté propriétaire de l'association.
- attribut **inverseJoinColumns** donne les noms des attributs de la table qui référencent les clés primaires du côté qui n'est pas propriétaire de l'association.

```
@Entity
@Table(name="matiere")
public class Matiere {
  @Id
  @Column(name="Code_Mat", length=6)
  private String code;
  @Column(name="Libelle_Mat", length=100)
  private String libelle;
  @ManyToMany
  @JoinTable(name="enseigner",
  joinColumns=@JoinColumn(name="Code_Mat"),
  inverseJoinColumns=@JoinColumn(name="Code_Ens"))
  private Set<Enseignant> enseignants=new HashSet<>();
  public Set<Enseignant> getEnseignants() {
  return enseignants;
  }
  public void setEnseignants(Set<Enseignant> enseignants) {
  this.enseignants = enseignants;
  }
  public void ajouterEnseigant(Enseignant ens){
  this.enseignants.add(ens);
  }
  //les constructeurs
  //les getters et les setters
}
```

Association many to many bidirectionnelle

- Dans la classe Matière, @ManyToMany nous indiquons que la collection des objets liés nommé enseignants, est de la classe Enseignant.
- Dans la classe Enseignant, @ManyToMany avec l'attribut mappedBy qui indiqué le mapping dans la classe Enseignant.

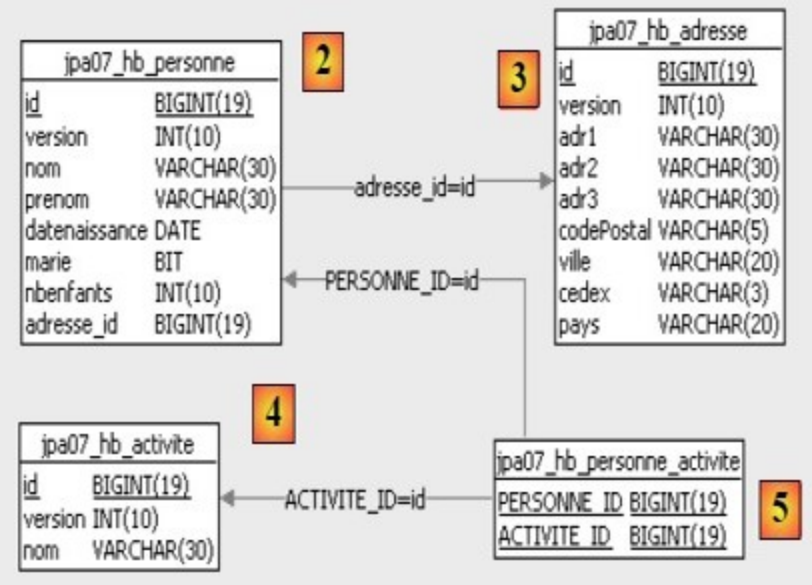
```
@Entity
@Table(name="enseignant")
public class Enseignant {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="Code_Ens")
  private int code;
  @Column(name="Nom_Ens", length=30)
  private String nom;
  @Column(name="Prenom_Ens", length=30)
  private String prenom;
  @ManyToMany(mappedBy="enseignants")
  private Set<Matiere> matieres=new HashSet<>();
  public Set<Matiere> getMatieres() {
  return matieres;}
  public void setMatieres(Set<Matiere> matieres) {
  this.matieres = matieres;}
  //les getters et les setters
}
```

```
@Entity
@Table(name="matiere")
public class Matiere {
  @Id
  @Column(name="Code_Mat", length=6)
  private String code;
  @Column(name="Libelle_Mat",length=100)
  private String libelle;
  @ManyToMany
  @JoinTable(name="enseigner",
  joinColumns=@JoinColumn(name="Code_Mat"),
  inverseJoinColumns=@JoinColumn(name="Code_Ens"))
  private Set<Enseignant> enseignants = new Hashet<>();
  public Set<Enseignant> getEnseignants() {
    return enseignants;}
  public void setEnseignants(Set<Enseignant> enseignants) {
    this.enseignants = enseignants;}
  public void ajouterEnseigant(Enseignant ens){
    this.enseignants.add(ens);}
  //les getters et les setters
}
```

Association many to many bidirectionnelle mise en pratique

```
public class MappingTest {
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    Session session = HibernateUtil.getSessionFactory().openSession();
    Transaction tx=session.getTransaction();
    tx.begin();
    Enseignant ens1=new Enseignant("Youness","Boukouchi");
    session.save(ens1);
    Matiere mat1=new Matiere("JEE","Java Entreprise Edition");
    mat1.ajouterEnseigant(ens1);
    Matiere mat2=new Matiere("SOA","Architecture Orientée Services");
    mat2.ajouterEnseigant(ens1);
    session.save(mat1);
    session.save(mat2);
    tx.commit();
    session.refresh(ens1);
    System.out.println("size :"+ens1.getMatieres().size());
    session.close();
  }
}
```

Dernier exemple Relations Many to Many




```java
@Entity
@Table(name = "jpa07_hb_personne")
public class Personne {
...
// relation Personne (many) -> Activite (many) via une table de jointure personne_activite
  @ManyToMany(cascade={CascadeType.PERSIST})
  @JoinTable(name="jpa07_hb_personne_activite", joinColumns= @JoinColumn(name = "PERSONNE_ID"),
  inverseJoinColumns = @JoinColumn(name = "ACTIVITE_ID"))
  private Set<Activite> activites = new HashSet<Activite>();
...
}
```

```java
@Entity
@Table(name = "jpa07_hb_activite")
public class Activite {
...
// relation inverse Activite -> Personne @ManyToMany(mappedBy = "activites")
  private Set<Personne> personnes = new HashSet<Personne>();
...
}
```

Clé primaire composée

Le modèle de base de données relationnelle permet la définition d'une clé primaire composée de plusieurs colonnes.

Nous avons pour cela 3 possibilités :

- **@Id** et **@Embeddable** : un seul attribut Id dans la classe entité
- **@IdClass** : correspond à plusieurs attributs Id dans la classe entité
- **@EmbeddedId** : un seul attribut Id dans la classe entité

Dans les trois cas, la clé doit être représentée par une classe Java :

- les attributs correspondent aux composants de la clé
- la classe doit être publique
- la classe doit avoir un constructeur sans paramètre
- la classe doit être sérialisable.
- La classe doit redéfinir equals() et hashcode().

Exemple @IdClass

```java
@Entity
@IdClass(CommandePK.class)
public class Commande {
  @Id
  private int num;
  @Id
  private int code;
  private int quantite;
  private double prix;
  public Commande() {}
  // les getters et les setters
}
```

```java
public class CommandePK  implements  Serializable {
  private int num;
  private int code;
  public CommandePK() {}
  // les getters et les setters
  public int hashCode() {
  ...
  }
  public boolean equals(Object obj) {
  ...
  }
}
```

Exemple @Id et @Embeddable

```java
@Entity
public class Commande {
  @Id
  private CommandePK pk;

  private int quantite;
  private double prix;
  public Commande() {}
  // les getters et les setters
}
```

```java
@Embeddable
public class CommandePK  implements  Serializable {
  private int num;
  private int code;

  public CommandePK() {}
  // les getters et les setters
  public int hashCode() {
  ...
  }
  public boolean equals(Object obj) {
  ...
  }
}
```

Exemple @EmbeddedId

```java
@Entity
public class Commande {
  @EmbeddedId
  private CommandePK pk;

  private int quantite;
  private double prix;
  public Commande() {}
  // les getters et les setters
}
```

```java
public class CommandePK  implements  Serializable {
  private int num;
  private int code;
  public CommandePK() {}
  // les getters et les setters
  public int hashCode() {
  …
  }
  public boolean equals(Object obj) {
  ...
  }
}
```

Gestion de l'héritage avec JPA

JPA offre plusieurs stratégies pour modéliser l'héritage, telles que l'héritage par table, l'héritage par classe concrète et l'héritage par table par classe. Chacune de ces stratégies a ses propres avantages et inconvénients en termes de performances, de lisibilité du code et de maintenabilité. En utilisant les annotations fournies par JPA, telles que @Inheritance et @DiscriminatorColumn, les développeurs peuvent définir efficacement la stratégie d'héritage souhaitée et mapper les entités héritées sur des tables relationnelles tout en maintenant l'intégrité des données et les performances de l'application.

Héritage par table (Table per class)

- Chaque classe dans la hiérarchie d'héritage est mappée sur une table distincte dans la base de données.
- Les colonnes spécifiques à chaque classe sont stockées dans la table correspondante.
- **Avantages** : Facile à comprendre et à implémenter, évite la redondance de données.
- **Inconvénients** : Peut entraîner des jointures complexes et des performances médiocres avec des hiérarchies profondes.

Héritage par classe concrète (Class per concrete class)

- Chaque classe concrète est mappée sur une table distincte dans la base de données.
- Les attributs de la classe parente et des sous-classes sont tous stockés dans la même table.
- **Avantages** : Performances potentiellement meilleures que l'héritage par table, pas de jointures nécessaires.
- **Inconvénients** : Peut entraîner une redondance de données si les sous-classes ont des attributs distincts.

Héritage par table par classe (Table per concrete class)

- Chaque classe, y compris la classe parente, est mappée sur une table distincte dans la base de données.
- Les attributs de chaque classe sont stockés dans leur propre table, même les attributs hérités.
- **Avantages** : Évite la redondance de données, performances potentiellement meilleures que l'héritage par table.
- **Inconvénients** : Peut nécessiter des jointures dans certaines requêtes, peut être complexe à maintenir.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

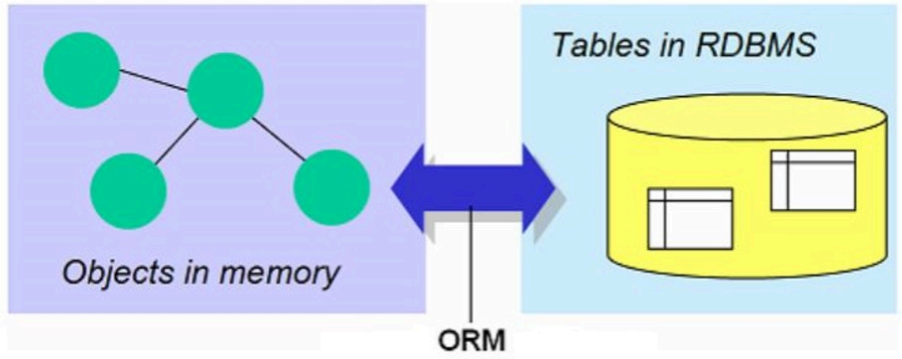
HIBERNATE

Introduction HIBERNATE

Rappel : Le mapping Objet/Relationnel (Object-Relational Mapping : **ORM**) consiste à réaliser la **correspondance** entre une application, modélisé en conception **orientée objet**, et une **base de données relationnelle**.

L' **ORM** a pour but d'établir la **correspondance** entre : une **table de la base de données** et une **classe du modèle objet**.

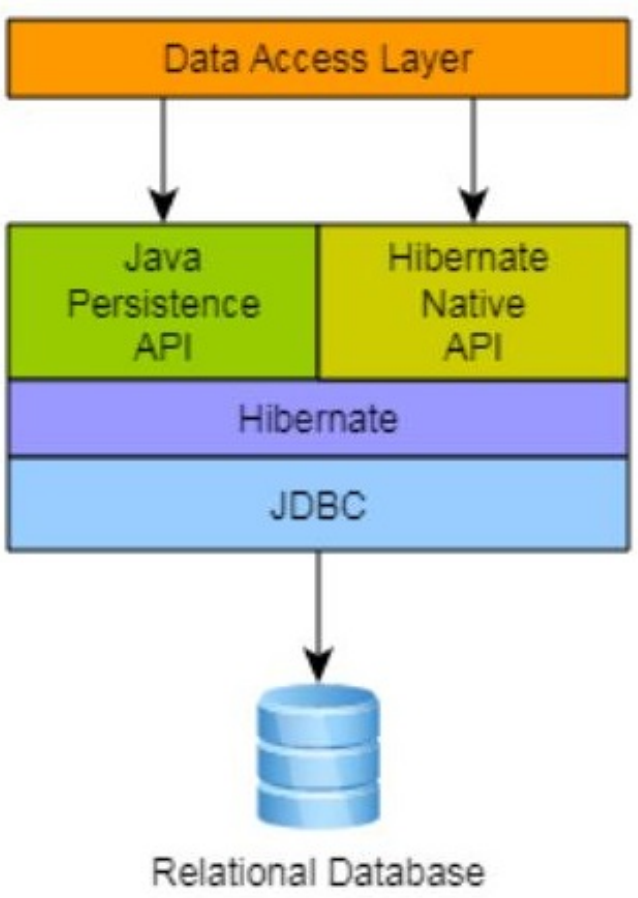
Hibernate est un **framework open source** gérant la persistance des objets en base de données relationnelle, développé par l'entreprise JBoss.



Introduction HIBERNATE

Hibernate est un framework de mapping objet-relationnel (ORM) pour Java, offrant une implémentation de la spécification Java Persistence API (JPA). En utilisant Hibernate, les développeurs peuvent mapper des objets Java à des tables de base de données et vice versa, facilitant ainsi le stockage et la récupération des données dans une application Java. En intégrant JPA, Hibernate fournit une couche d'abstraction supplémentaire pour la persistance des données, simplifiant le développement et offrant une portabilité accrue entre les différents fournisseurs de bases de données.

Introduction HIBERNATE



- **Hibernate** propose son **propre langage d'interrogation HQL** et a **largement inspiré les concepteurs de l'API JPA**.
- **Hibernate** est **très populaire** notamment à cause de ses **bonnes performances** et de son ouverture à de nombreuses bases de données.
- Les **bases de données supportées** sont les principales du marché : HSQL Database Engine, DB2/NT, **MySQL**, **PostgreSQL**, FrontBase, Oracle, **Microsoft SQL Server Database**, Sybase SQL Server, Informix Dynamic Server, etc.

Configuration hibernate

- **Hibernate** a besoin de plusieurs éléments pour fonctioner :
  - **une classe de type** javabean qui encapsule les données d'une occurrence d'une table (**Entité**)
  - Un **fichier de la configuration** pour configurer les informations de la BD et les informations du mapping (**hibernate.properties** ou **hibernate.cfg.xml**)
  - un fichier de correspondance qui configure la correspondance entre la classe et la table (.hbm.xml) ou bien l'utilisation **des annotations**.
- Une fois ces éléments correctement définis, il est possible d'utiliser Hibernate dans le code des traitements à réaliser.

hibernate.properties

Le fichier hibernate.properties va contenir les principales propriétés pour configurer la connexion JDBC mais aussi d'autres propriétés :

- hibernate.connection.driver_class : nom pleinement qualifié de classe du pilote JDBC
- hibernate.connection.url : URL JDBC désignant la base de données
- hibernate.connection.username : nom de l'utilisateur pour la connexion
- hibernate.connection.password : mot de passe de l'utilisateur
- hibernate.connection.pool_size : nombre maximum de connexions dans le pool
- hibernate.dialect : nom de la classe pleinement qualifiée qui assure le dialogue avec la base de données
- hibernate.show_sql : booléen qui précise si les requêtes SQL générées par Hibernate sont affichées dans la console
- hibernate.hbm2ddl.auto : Exporte automatiquement le schéma DDL vers la base de données lorsque la SessionFactory est créée:
    - validate: valider la structure du schéma sans faire de modification dans la base de données
    - update: mettre à jour le schéma
    - create: créer le schéma en supprimant celui existant
    - create-drop: créer le schéma et le supprimer lorsque la sessionFactory est fermée

Exemple hibernate.properties

```
## Avec une BDD Postgresql

# Hibernate Configuration
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost:5432/demo_hibernate
hibernate.connection.username = postgres
hibernate.connection.password = test
hibernate.hbm2ddl.auto = update
hibernate.show_sql = true
```

hibernate.cfg.xml

Le fichier `hibernate.cfg.xml` est un fichier de configuration optionnel utilisé par Hibernate pour définir des paramètres supplémentaires et des configurations avancées lors de la mise en place d'une session Hibernate dans une application Java. Bien que la plupart des configurations de base puissent être spécifiées dans le fichier `hibernate.properties`, le fichier `hibernate.cfg.xml` offre une flexibilité supplémentaire en permettant de définir des configurations plus détaillées, telles que les classes mappées, les stratégies de cache avancées, les gestionnaires d'événements d'entité, etc. Il agit comme un complément au fichier `hibernate.properties`, permettant aux développeurs de personnaliser davantage le comportement de Hibernate dans leur application.

Exemple hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <mapping class="org.example.entities.Personne"/>
        <mapping class="org.example.entities.Entreprise"/>
    </session-factory>
</hibernate-configuration>
```

Deux formes de mapping

Lorsque vous utilisez Hibernate, vous avez le choix entre deux approches pour effectuer le mapping objet-relationnel (ORM) : l'utilisation de fichiers XML de configuration ou l'utilisation d'annotations.

1. **Fichiers XML de configuration** : Historiquement, Hibernate utilisait des fichiers XML de configuration pour définir le mapping entre les classes Java et les tables de la base de données, ainsi que d'autres paramètres de configuration. Bien que cette approche soit toujours prise en charge par Hibernate, elle est de moins en moins utilisée en raison de sa complexité et de la préférence croissante des développeurs pour une configuration plus concise et intégrée.
2. **Annotations** : Les annotations sont devenues une méthode populaire pour spécifier le mapping ORM directement dans le code source des classes Java. Hibernate prend en charge un ensemble complet d'annotations pour définir le mapping des entités, les associations, les clés primaires, les clés étrangères, etc. L'utilisation d'annotations simplifie la configuration en regroupant les informations de mapping directement avec le code des entités, ce qui rend le code plus lisible et maintenable.

Bien que les deux approches soient possibles, l'utilisation d'annotations est devenue la norme pour la plupart des nouveaux projets Hibernate en raison de sa simplicité et de sa meilleure intégration avec le code Java. Cependant, les fichiers XML de configuration restent disponibles pour ceux qui préfèrent cette approche ou pour les projets existants qui utilisent déjà des configurations XML.

Les Annotations

Pour travailler avec Hibernate, il suffit de maîtriser les annotations de JPA, car Hibernate est une implémentation de JPA. En utilisant des annotations standardisées telles que @Entity, @Table, @Column, @ManyToOne, @OneToMany, etc., vous pouvez définir le mapping objet-relationnel de manière portable et efficace, profitant ainsi des fonctionnalités offertes à la fois par JPA et par Hibernate.

Conclusion configuration hibernate

Dans notre projet nous aurons 2 fichiers de configurations :

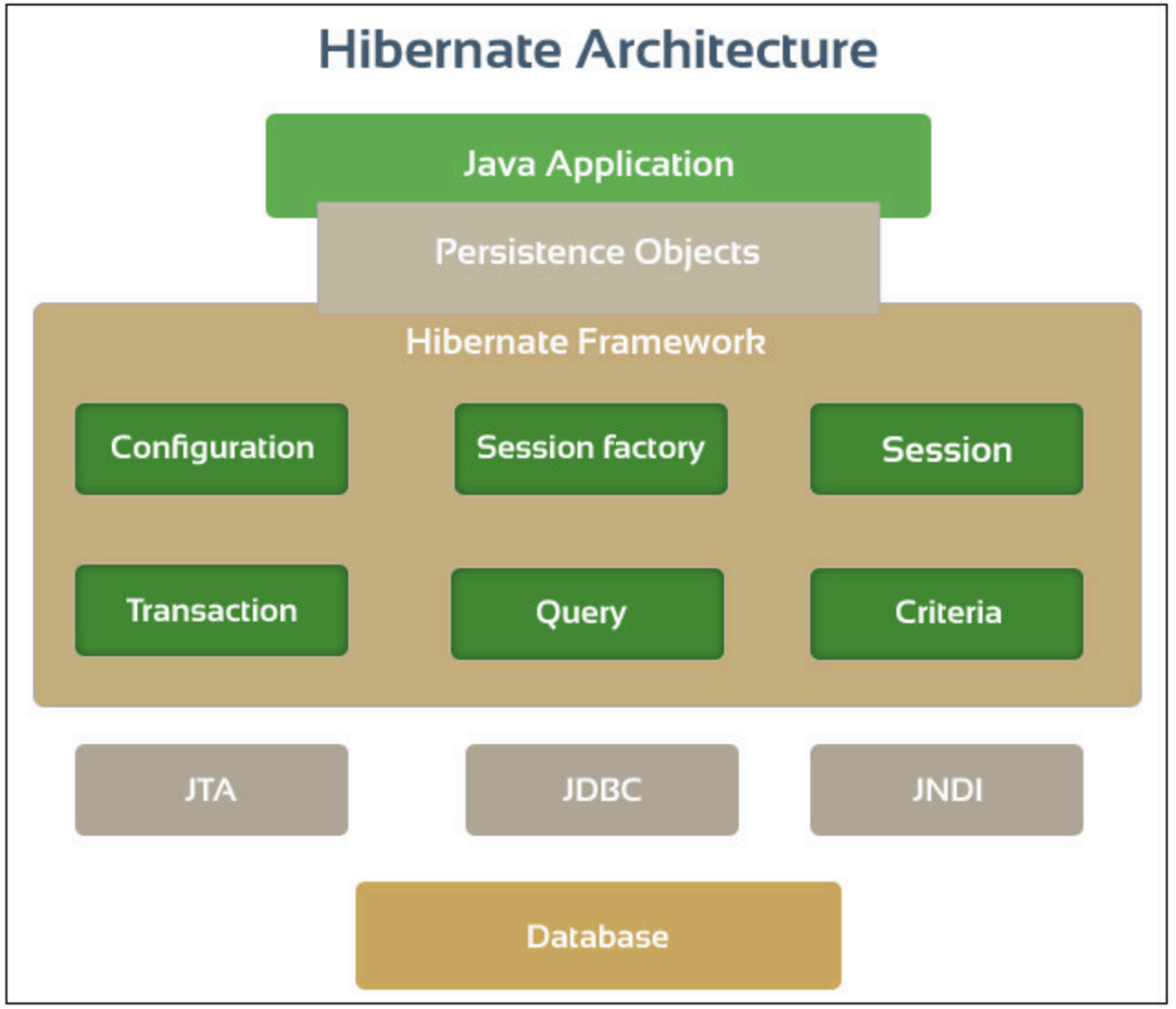
- hibernate.cfg.xml et/ou hibernate.properties dans le répertoire src/main/ressources de notre projet.
- Nous utiliserons les @Annotations afin de réaliser le mapping objet-relationnel.
- Il faudra bien évidemment rajouter les dépendances nécessaires à notre projet. (Voir l'exemple page suivante pour un projet Maven.)

Dependances hibernate maven

```
<dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-entitymanager</artifactId>
            <version>5.6.14.Final</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.28</version>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.6.0</version>
        </dependency>
 </dependencies>
```

Utilisation de Hibernate

- Pour utiliser Hibernate dans le code, il est nécessaire de réaliser plusieurs opérations :
  - création d'une instance de la classe Configuration (les propriétés sont définies dans le fichier hibernate.cfx.xml).
  - création d'une instance de la classe SessionFactory, cet objet est obtenu à partir de l'instance du type Configuration en utilisant la méthode buildSessionFactory().
  - création d'une instance de la classe Session qui va permettre d'utiliser les services d'Hibernate, la classe Session est obtenu à partir d'une fabrique de type SessionFactory à l'aide de sa méthode openSession().
  - Par défaut, la méthode openSession() ouvre une connexion vers la base de données en utilisant les informations fournies par les propriétés de configuration.


Hibernate Architecture
Java Application
Persistence Objects
Hibernate Framework
Configuration
Session factory
Session
Transaction
Query
Criteria
JTA
JDBC
JNDI
Database

Exemple

```java
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
public class Demo1 {
    public static void main(String[] args) {
      // Création d'un registre pour charger la configuration à partir de notre fichier de configuration
        StandardServiceRegistry registre = new StandardServiceRegistryBuilder().configure().build();
        SessionFactory sessionFactory = new MetadataSources(registre).buildMetadata().buildSessionFactory();

        // Création de la session
        Session session = sessionFactory.openSession();
        // Dés l'ouverture de la session, et en fonction de la propriété hibernate.hbm2ddl.auto hibernate va agir sur la base de donnée

        // ... manipulation/interaction avec les entites / BDD

        //Fermeture de la session et la sessionfactory
        session.close();
        sessionFactory.close();
    }
```

Utilisation de Hibernate

- Pour créer une nouvelle occurrence dans la source de données, il suffit de :
  - créer une nouvelle instance de la classe encapsulant les données, de valoriser ces propriétés, et d'appeler la méthode save() de la session en lui passant en paramètre l'objet encapsulant les données.
  - La méthode save() n'a aucune action directe sur la base de données.
  - Pour enregistrer les données dans la base, il faut réaliser un commit sur la transaction de la classe Session.

```
// Ajout d'une personne
        session.getTransaction().begin();
        Personne pe = new Personne();
        pe.setNom("toto");
        pe.setPrenom("tata");
        session.save(pe);
        System.out.println("ID de la personne : " + pe.getId());
        session.getTransaction().commit();
```

Utilisation de Hibernate

- La méthode load() ou get() de la classe Session permet d'obtenir une instance de la classe des données encapsulant les données de l'occurrence de la base dont l'identifiant est fourni en paramètre.
  - Personne perChercher=session.load(Personne.class, 1); // lève une exception si n'est pas trouvé
  - Personne perChercher=session.get(Personne.class, 1); // retourne null si n'est pas trouvé
  - System.out.println("Nom :"+ perChercher.getNomPersonne());
- Pour supprimer une occurrence encapsulée dans une classe, il suffit d'invoquer la méthode delete() de la classe Session en lui passant en paramètre l'instance de la classe.
  - session.delete(perChercher);
- Pour mettre à jour une occurrence dans la source de données, il suffit d'appeler la méthode update() de la session en lui passant en paramètre l'objet encapsulant les données. La méthode saveOrUpdate() laisse Hibernate choisir entre l'utilisation de la méthode save() ou update() en fonction de la valeur de l'identifiant dans la classe encapsulant les données. Exemple :
  - perChercher.setNomPersonne("Amal");
  - session.update(perChercher);

Exemple 1

```
import org.example.entities.Personne;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
public class Demo1 {
    public static void main(String[] args) {
        // Création d'un registre pour charger la configuration à partir de notre fichier de configuration
        StandardServiceRegistry registre = new StandardServiceRegistryBuilder().configure().build();
        SessionFactory sessionFactory = new MetadataSources(registre).buildMetadata().buildSessionFactory();
        // Création de la session
        Session session = sessionFactory.openSession();
        // Dés l'ouverture de la session, et en fonction de la propriété hibernate.hbm2ddl.auto hibernate va agir sur la base de donnée
        // Récupérer une personne
        session.getTransaction().begin();
        Personne p = session.load(Personne.class,1L);
        System.out.println(p.getNom());
        // Modification Attention il est important d'être dans la même transaction si on souhaite modifier ou supprimer
        // on met à jour avec la methode update
        p.setPrenom("titi");
        session.update(p);
        //On supprimme avec la methode delete
        session.delete(p);
        session.getTransaction().commit();
        //Fermeture de la session et la sessionfactory
        session.close();
        sessionFactory.close();
    }
}
```

Exemple 2

```
public static void main(String[] args) {
  SessionFactory factory=HibernateUtil.getFactory()
  Session session=factory.openSession();
  session.getTransaction().begin();
  Personne persRechercher=session.load(Personne.class, 4);
  persRechercher.setAge(20);
  persRechercher.setNom("Mohamed");
  session.update(persRechercher);
  Personne persSupprimer=session.load(Personne.class, 5);
  session.delete(persSupprimer);
  session.getTransaction().commit();
  session.close();
}
```

```
public class HibernateUtil {

  private static SessionFactory factory=null;

  private HibernateUtil(){}

  public static SessionFactory getFactory(){
    if (factory==null){
      StandardServiceRegistry registre= new StandardServiceRegistryBuilder().configure().build();
      factory=new MetadataSources(registre).buildMetadata().buildSessionFactory();
    }
    return factory;
  }
}
```

HQL Hibernate Query Language

Hibernate utilise plusieurs moyens pour obtenir des données de la base de données :

- Hibernate Query Language (HQL)
- API Criteria // deprecated par Hibernate 5, utiliser CriteriaQuery de API JPA .
- SQL natives

HQL Hibernate Query Language

Hibernate propose son propre langage nommé HQL (Hibernate Query Language) pour interroger les bases de données.

- Le langage HQL est proche de SQL avec une utilisation sous forme d'objets des noms de certaines entités.
- Il n'y a aucune référence aux tables ou aux champs car ceux-ci sont référencés respectivement par leur classe et leurs propriétés.
- Hibernate qui se charge de générer la requête SQL à partir de la requête HQL en tenant compte du contexte.

HQL Hibernate Query Language

- Une requête HQL peut être composée :
  - de clauses (from, select, where, ...)
  - de fonctions d'agrégation (Max, Count, Min, ...)
  - de sous requêtes

HQL Hibernate Query Language VS JDBC SQL

```
List<Movie> movies = new ArrayList<Movie>();
Movie m = null;
try {
  Statement st = getConnection(). createStatement();
  ResultSet rs = st. executeQuery("SELECT * FROM MOVIES");
  while (rs. next()) {
    m = new Movie();
    m.setId(rs.getInt("ID"));
    m.setTitle(rs.getString("TITLE "));
    movies.add(m);
  }
} catch (SQLException ex) {
  System. err. println(ex. getMessage());
}
```

```
List<Movie> movies=session.createQuery("from Movie").list(); // Hibernate HQL
```

La classe Query : createQuery

- La mise en œuvre de HQL peut se faire de plusieurs manières.La plus courante est d'obtenir une instance de la classe Query en invoquant la méthode createQuery() de la session Hibernate courante.
- **createQuery()** attend en paramètre la requête HQL qui devra être exécutée.

```
// Get the current session
Session session = sessionFactory.getSesion();
// Instance of the query is created from this session
Query query = session.createQuery("from Personne");
```

La classe Query : liste() & iterator()

- Pour parcourir la collection des occurrences trouvées, l'interface Query propose:
  - la méthode list() qui renvoie une collection de type List
  - la méthode list().iterator() qui renvoie un itérateur sur la collection

```
// L'utilisation d'un iterator()
Query query = session.createQuery("from Personne");
Iterator personnes= query.iterate();
while(personnes.hasNext()){
  Personne per = (Personne) personnes.next();
  System.out.println("Personne :"+ per);
}
```

```
// L'utilisation d'une liste()
Query query = session.createQuery("from Personne");
List<Personne> personnes= query.list();
for (Personne per : personnes) {
  System.out.println("Personne:"+ per);
}
```

La classe Query : les clauses

Les clauses sont les mots clés HQL qui sont utilisés pour définir la requête :

- **where condition** : précise une condition qui permet de filtrer les occurrences retournées. Doit être utilisé avec une clause select et/ou from
  - **from Personne where nom ='Ali'**
- **order by propriété [asc|desc]**: précise un ordre de tri sur une ou plusieurs propriétés. L'ordre par défaut est ascendant
  - **from Personne where nom='ginfo' order by prenom desc, age asc**
- **group by propriete** : précise un critère de regroupement pour les résultats retournés. Doit être utilisé avec une clause select et/ou from
  - **from Personne order group by age**

La classe Query : uniqueResult

Dans le cas où on a attend qu'un seul occurrence (uniqueResult):

```
Query query = session. createQuery(" from Personne where id=1 ");
Personne per = (Personne) query.uniqueResult();
```

La classe Query : Paramètre nommé

- Il est également possible de définir des requêtes utilisant des paramètres nommés.
- Dans ces requêtes, les paramètres sont précisés avec un caractère « : » suivi d'un nom unique.
- L'interface Query propose de nombreuses méthodes setXXX() pour associer à chaque paramètre une valeur en fonction du type de la valeur (XXX représente le type). // deprecated dans Hibernate 5, utiliser setParameter(para,valeur)
- Chacune de ces méthodes possède deux surcharges permettant de préciser le paramètre à partir de son nom dans la requête et sa valeur.

```
Query query = session.createQuery("from Personne where id=:idP and nom=:nomP");
query.setInteger("idP", 1 );
query.setString("nomP", "Ali");
```

La classe Query : Paramètre positionné

- Il est également possible de définir des requêtes utilisant des paramètres positionné.
- Dans ces requêtes, les paramètres sont précisés avec un caractère « ? ».
- les méthodes setXXX() ou setParameter() possède deux surcharges permettant de:
  - préciser l'index de paramètre dans la requête(à partir de 0)
  - la valeur du paramétre

```
Query query = session.createQuery("from Personne where id=? and nom=?");
query.setInteger("idP", 1 );
query.setString("nomP", "Ali");
```

La classe Query : Alias

- Parfois, nous souhaitons donner un nom à l'objet que nous interrogeons. Les noms donnés à ces objets sont appelés alias, et ils sont particulièrement utiles lorsque nous construisons des jointures ou des sous-requêtes.

```
// The per is the alias to the object Personne
Query query = session. createQuery( "from Personne as per where per.nom=:nom and per.id=:id" );
```

La classe Query : Select

```
// Une seule colonne
Query query = session.createQuery("SELECT nom from Personne");
List<String> listNoms= query.list();
System.out.println("Nom des personnes:");
// Loop through all of the result columns
for (String nom: listNoms) {
  System.out.println("\t"+ nom);
}
```

```
String QUERY = "SELECT new Chef(id, nom ) from Personne";
List<Chef> chefs = session.createQuery(QUERY).list();
for (Chef chef : chefs) {
  System.out.println("Les chefs: " +chef);
}
```

```
String qr = "SELECT nom, prenom from Personne";
Query query = session.createQuery(qr);
Iterator personnes= query.list().iterator();
while(personnes.hasNext()){
  Object[] p = (Object[])personnes.next();
  System.out.print("Nom: "+p[0]+ \t );
  System.out.println("Prenom: " +p[1]);
}
```

La classe Query : Fonctions d'agrégation

Les fonctions d'agrégation HQL ont un rôle similaire à celles de SQL, elles permettent de calculer des valeurs agrégeant des valeurs de propriétés issues du résultat de la requête.

- **count([distinct|all|*] object | object.property)**
- **sum([distinct|all] object.property)**
- **avg([distinct|all] object.property)**
- **max([distinct|all] object.property)**
- **min([distinct|all] object.property)**

```
// Fetching the max age
int ageMax= (int)session.createQuery("select max(age) from Personne").uniqueResult();
System.out.println("Age max:"+ageMax);
// Getting the average age of persons
double ageMoyen= (double)session.createQuery("select avg(age) from Personne").uniqueResult();
System.out.println("Age Moyenne:"+ageMoyen);
```

La classe Query : IN {ensemble}

Nous utilisons la clause IN pour extraire des données avec des critères correspondant à une liste sélective.

```
// Define a list and populate with our criteria
List villeList = new ArrayList();
villeList.add("Casablanca");
villeList.add("Agadir");
Query query = session.createQuery("from Personne where ville in (:villes) ");
query.setParameterList("villes", villeList);
List<Personne> personnes = query.list();
...
```

La classe Query : Updates et Deletes

- Il existe un autre moyen de mettre à jour et de supprimer des données en utilisant la méthode executeUpdate de la requête.
- La méthode attend une chaîne de requête avec des paramètres de liaison.

```
// To update the record
String update_query="update Personne set nom=:nomP where id=2";
Query query = session. createQuery(update_query);
query.setParameter("nomP", "Sirina");
int success = query.executeUpdate();
```

```
// To delete a record
String delete_query="delete Personne where id=5";
Query query = session.createQuery(delete_query);
int success = query.executeUpdate();
```

API Criteria

L'API Criteria était auparavant une option populaire pour construire des requêtes dynamiques avec Hibernate. Cependant, à partir de Hibernate 5, l'API Criteria est considérée comme obsolète et est remplacée par l'API JPA Criteria. Cette dernière est une spécification de JPA qui fournit une API standardisée pour la construction de requêtes dynamiques, indépendamment du fournisseur JPA utilisé (comme Hibernate).

Bien que l'API Criteria de Hibernate soit encore fonctionnelle et utilisable, il est recommandé d'utiliser l'API JPA Criteria.

Exemple API JPA Criteria

```
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.CriteriaQuery;

// Supposons que entityManager est une instance de l'EntityManager

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
CriteriaQuery<Person> criteriaQuery = criteriaBuilder.createQuery(Person.class);

Root<Person> root = criteriaQuery.from(Person.class);
criteriaQuery.select(root);

// Création du prédicat pour filtrer les personnes dont le prénom commence par "John"
Predicate predicate = criteriaBuilder.like(root.get("firstName"), "John%");

// Ajout du prédicat à la requête
criteriaQuery.where(predicate);

// Exécution de la requête
List<Person> results = entityManager.createQuery(criteriaQuery).getResultList();

// Affichage des résultats
for (Person person : results) {
    System.out.println(person.getFirstName() + " " + person.getLastName());
}
```

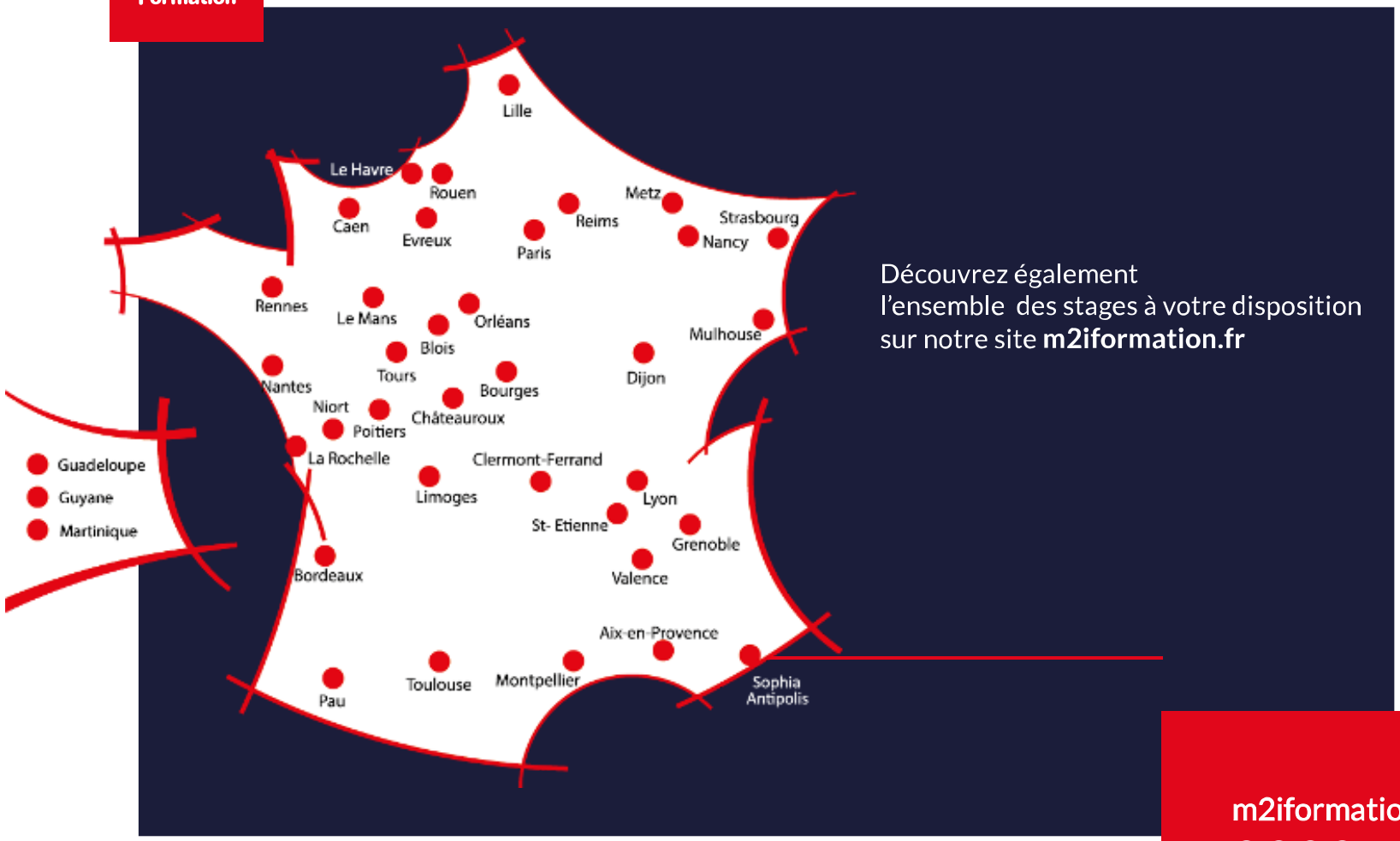
Native SQL

- Hibernate fournit également une fonctionnalité pour exécuter des requêtes SQL natives.
- La méthode session.createSQLQuery renvoie un objet SQLQuery, similaire à la méthode utilisée par createQuery pour renvoyer un objet Query.
- Cette classe étend la classe Query vu précédemment.

```
SQLQuery query = session. createSQLQuery("select * from PersonneTable");
List employees = query. list();
```

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr