

Framework de persistance

HJPA : Hibernate Java Persistance Api

A. MADANI

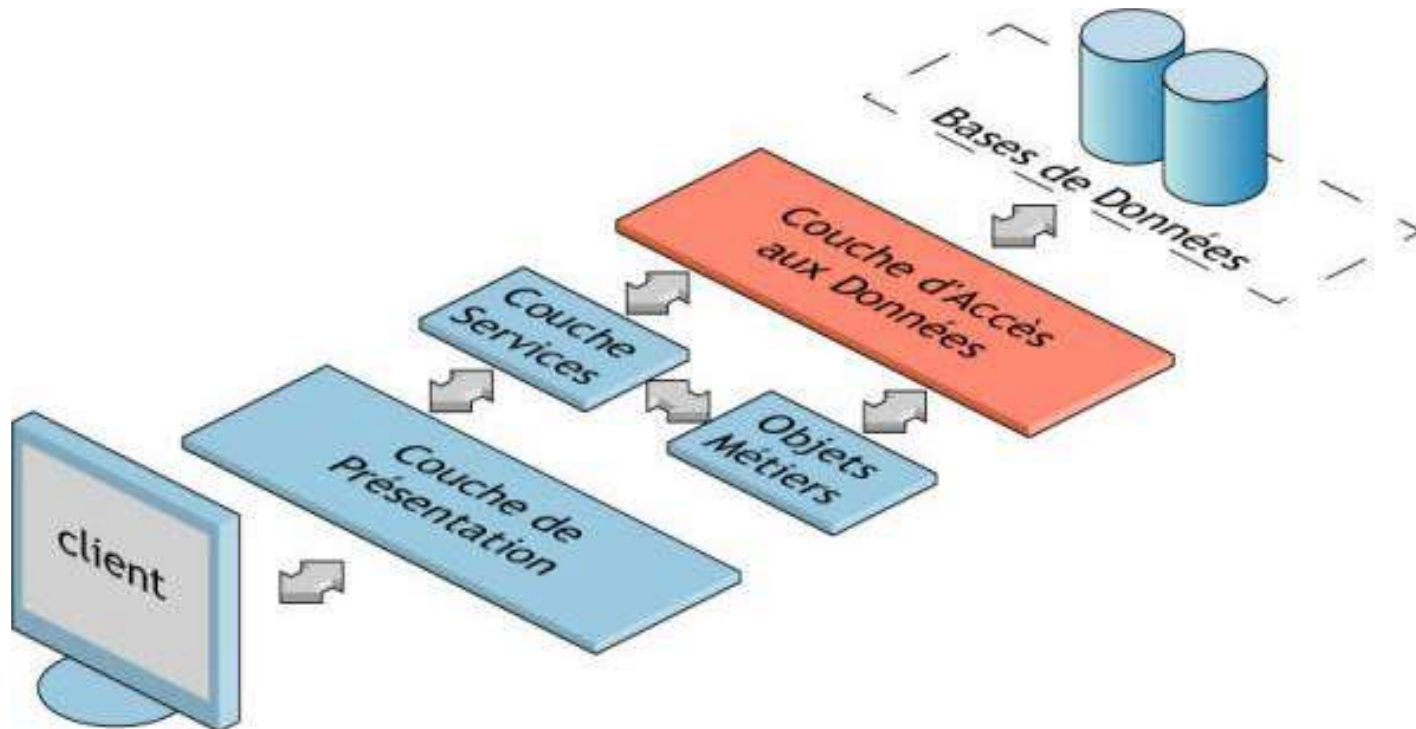
Madani.a@ucd.ac.ma

ORM avec Hibernate JPA

- Introduction/Rappel
- Gestion de la correspondance objet/relationnel
- ORM: Object Relationnel Mapping
- Framework: Hibernate, Tooplik,...
- JPA: Java persistance API
- Mapping Objet Relationnel Avec JPA (entités, annotations, fichier de configuration)
- Gestion d'une entité
- Requêtes : JP QL, Creteria API, SQL Native
- Mapping des relations

Introduction (Rappel)

En général, une architecture multi-couches présente ces couches :



Introduction (Rappel)

Avantages

- ❑ Couches indépendantes ➔ modification d'une couche n'affecte pas les autres
- ❑ Le développement pourra se faire en parallèle
- ❑ La maintenance de l'application n'en sera que plus aisée.
- ❑ Rendre l'accès aux données complètement indépendant du SGBD utilisé. Il devient donc très simple de changer de SGBD au cours du développement de l'application

Introduction (Rappel)

- Couche de présentation :
 - ❑ constitue la partie qui est en contact direct les médias de sortie
 - ❑ correspond à l'interface avec laquelle l'utilisateur interagit
 - ❑ responsable de la représentation externe sous forme textuelle et graphique des objets internes et abstraits de l'application
 - ❑ **correspond à la couche Vue du modèle MVC**

Introduction (Rappel)

- couche métier :
 - ❑ stocke les données de l'application
 - ❑ contient les objets métiers de l'application
 - ❑ avant son implémentation, cette couche fait l'objet d'une conception approfondie en élaborant les différents diagrammes UML, notamment le diagramme de classes
 - ❑ **correspond à la couche Modèle du modèle MVC**

Introduction (Rappel)

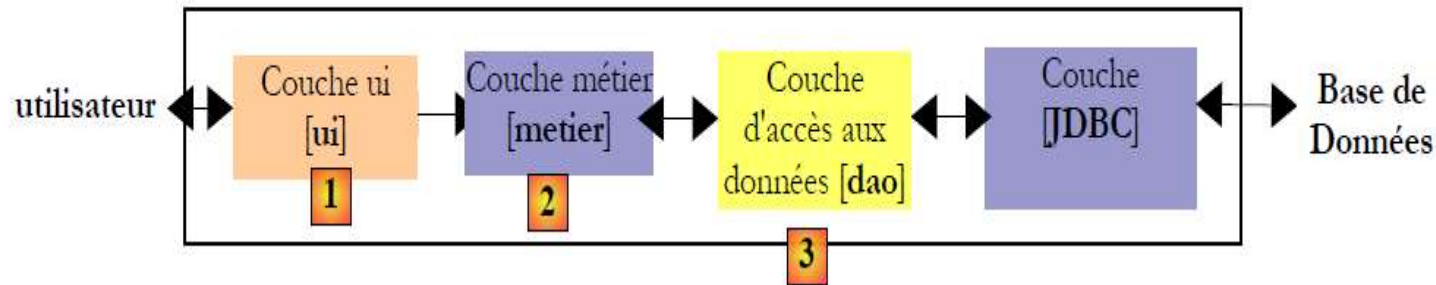
- Couche service
 - ❑ Traite les entrées de l'utilisateur
 - ❑ Il informe la couche métier des interactions faites par l'utilisateur
 - ❑ La couche métier modifie alors son état et informe la couche présentation du nouvel aspect qu'elle doit prendre suite à cette modification
 - ❑ **Correspond à la couche Contrôleur du modèle MVC**

Introduction (Rappel)

- Couche dao :
 - La couche d'accès aux données doit prendre en charge toutes les interactions entre l'application et la base de données.
 - la création
 - la lecture
 - la modification
 - la suppression
 - des enregistrements dans chacune des tables de la base de données.
 - **Généralement**, on crée pour chaque classe du modèle métier une classe correspondante dans la couche d'accès aux données

Introduction (Rappel)

- En résumé, une architecture multi couches se présente comment suit :



- La couche [1] contient les interfaces utilisateur (UI : User Interfaces)
- La couche [2] est la partie métier : Classes métiers
- La couche [3], appelée [dao] (Data Access Object) est la couche qui fournit à la couche [2] des données pré-enregistrées (fichiers, bases de données, ...) et qui enregistre certains des résultats fournis par la couche [2].

Introduction

- La couche DAO gère des données provenant de deux mondes:
 - Monde relationnel (BDR)
 - Monde Objet (Couche métier)
- Comment gérer la persistance des objets provenant de la couche métier?
- Il faut faire la correspondance entre les deux mondes.
- Solutions:
 - Manuelle (solution 1 : c'est ce que nous avons fait jusqu'ici)
 - Utilisation d'un framework de mapping O/R (solution 2)

Gestion de la correspondance objet/relationnel (solution 1)

- 30% du code java est consacré à gérer la mise en rapport SQL/JDBC et la non correspondance entre le modèle objet et le modèle relationnel
 - Gestion des classes, objets, attributs, ..
 - Correspondances de types
 - Gestion des associations
- Code dépend du SGBDR utilisé. Si on change le SGBDR, il faut revoir le code Java

Example

```
String driver="com.mysql.jdbc.Driver";  
String strcon="jdbc:mysql://localhost:3306/smi_2024_2025"; String  
req="insert into Users(name, profile) values(?,?)";  
try {  
    Class.forName(driver);  
    Connection con = DriverManager.getConnection(strcon,"root","");  
    PreparedStatement st = con.prepareStatement(req);  
    st.setString(1, name);  
    st.setString(2, profile);  
    st.execute();  
} catch(Exception e) {  
    System.out.println(e.getMessage());  
}
```

Exemple : différents drivers

- **Mysql**

String url = "jdbc:mysql://localhost:3306/DB";

- **SQL Server**

String url = "jdbc:sqlserver://localhost:1433;databaseName=DB";

- **Oracle**

String url = "jdbc:oracle:thin:@localhost:1521:xe";

- **PostgreSQL**

String url = "jdbc:postgresql://localhost:5432/DB";

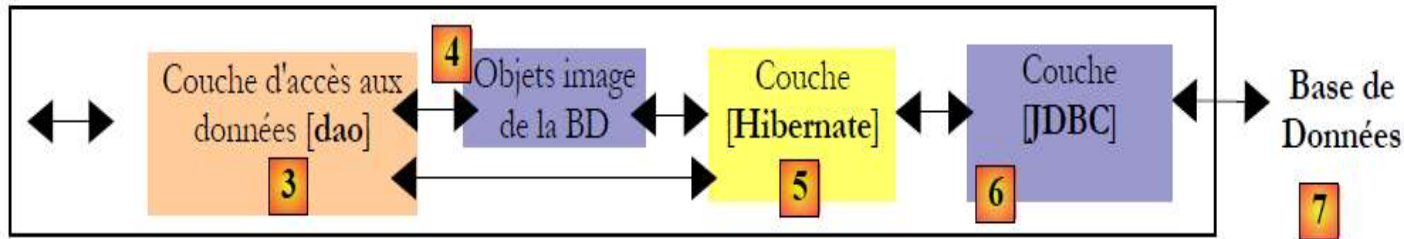
Gestion de la correspondance objet/relationnel (solution 2)

- Le mapping Objet/Relationnel (ORM) désigne la persistance automatisée et transparente d'objets dans une application Java vers les tables d'une base de données relationnelle à l'aide des méta données décrivant le mapping entre les objets et la base de données.
- Les outils et frameworks ORM réduisent grandement le code qui permet cette correspondance et facilitent donc les adaptations mutuelles des deux modèles.
- Exemple de frameworks:
 - Hibernate
 - Toplink
 - EclipseLink
 - MyBatis
 - etc

ORM : Object Relationnel Mapping

- C'est une technique de mapping d'une représentation des données d'un modèle objet vers un modèle relationnel de base de données et inversement.
- Quatre éléments constituant une solution ORM :
 1. Une API pour effectuer les opérations **CRUD** (**Create, Read, Update, delete**) de base sur les objets des classes persistantes.
 2. Un langage ou une API pour spécifier des requêtes qui se réfèrent aux classes et aux propriétés des classes.
 3. Un système pour spécifier des **métadonnées de mapping**.
 4. Une technique pour l'implémentation du mapping objet/relationnel permettant d'interagir avec des objets

Solutions ORM: Hibernate



- Hibernate est un ORM (Object Relational Mapping) qui fait le pont entre le monde relationnel des bases de données et celui des objets manipulés par Java.
- Le développeur de la couche [dao] ne voit plus la couche [Jdbc] ni les tables de la base de données.
 - Il ne voit que l'image objet de la base de données, image objet fournie par la couche [Hibernate].
- XML constitue le format de description de la correspondance entre les tables relationnelles et les classes Java.

Hibernate (suite)

- Hibernate fournit au développeur, un langage HQL (Hibernate Query Language) pour interroger le contexte de persistance [4] et non la BD elle-même.

Limites :

- Hibernate est populaire mais complexe à maîtriser:
 - Dès qu'on a une base de données avec des tables ayant des relations un-à-plusieurs ou plusieurs-à-plusieurs, la configuration du pont relationnel/objets devient complexe.
 - Des erreurs de configuration peuvent alors conduire à des applications peu performantes.
- Apparition des plusieurs ORMs
 - Toplink
 - JDO
 - Eclipse Link
 - Ibatis
 - etc

Exemple de *.hbm.xml

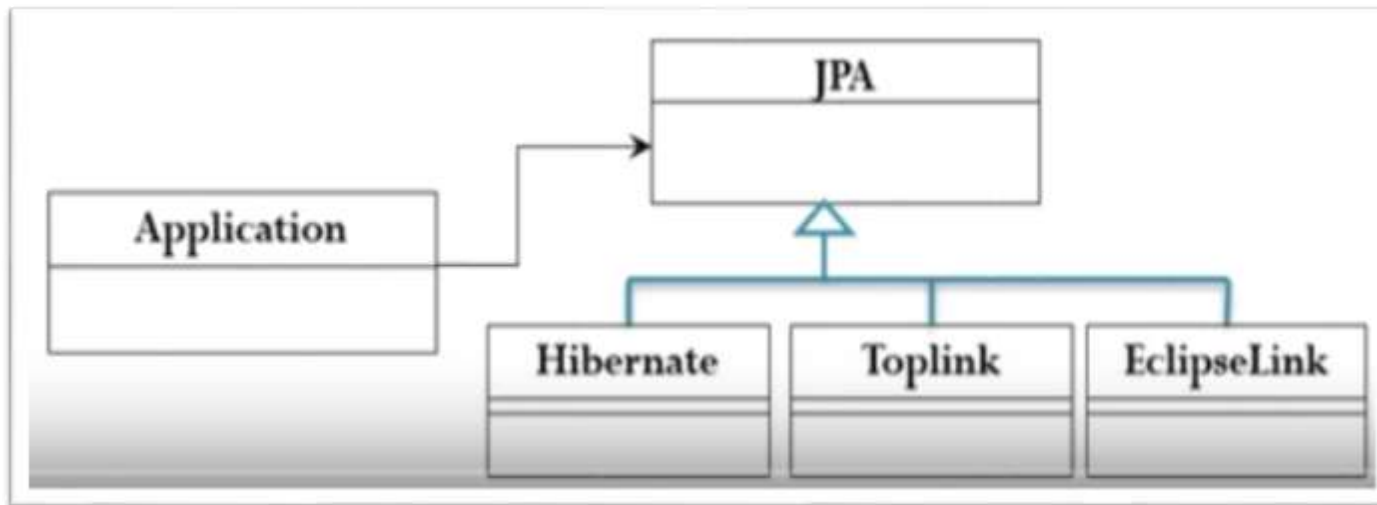
```
<class name="User" table="utilisateurs">
  <id name="id" column="id" type="long">
    <generator class="identity"/>
  </id>
  <property name="name" column="nom" type="string"/>
  <property name="address" column="adresse" type="string"/>
  <property name="city" column="ville" type="string"/>
</class>
```

Inconvénients

- Plusieurs Framework ORM (Hibernate, TopLink, etc.), chacun avait son langage d'interrogation de la BD, sa propre API et ses propres conventions.
- Cela rendait difficile :
 - L'interopérabilité entre les différents Framework.
 - Le transfert de compétences d'un projet à un autre.
 - La migration d'une application d'un ORM à un autre.

JPA: Java persistence API (1)

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



JPA: Java persistence API (1)

- JPA a été introduit comme une spécification standard pour la persistance des données en Java.
- Elle définit un ensemble commun d'APIs et de fonctionnalités que tous les frameworks ORM doivent implémenter.
- Ainsi, les développeurs peuvent écrire du code conforme à JPA et basculer facilement entre différentes implémentations (par exemple, passer de Hibernate à EclipseLink) sans réécrire leur logique métier.

Récapitulatif

- Un mapping objet-relationnel (ORM) est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet.
- Une classe pour laquelle il existe une correspondance en base de données est appelée une **entité**.

Principe de mise en correspondance	
Programmation OO	Base de données
Classe	Table
Attribut d'une classe	Champ dans une table
Référence entre classes	Relation entre table
Un objet	Un enregistrement dans une table

Récapitulatif

- Définition

Java Persistence API (JPA) est une spécification d'interface de programmation pour la gestion de données relationnelles.

- Possibilités offertes

Manipulation de données et de leurs relations directement en Java.

- Différentes implémentations de la spécification :

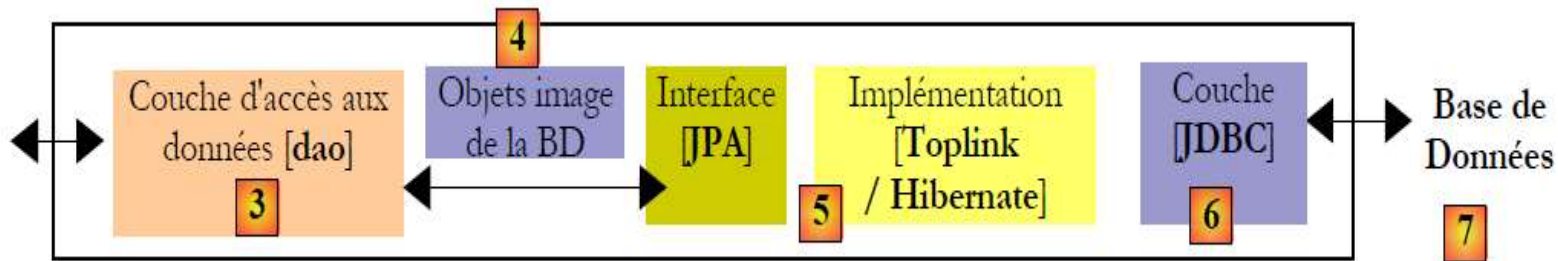
- Hibernate,
- TopLink,
- EclipseLink,
- Apache OpenJPA,
- ...

- Concrètement

- Les correspondances se font via des annotations JPA dans les classes,
- une implémentation fournit une API pour manipuler les données,
- JPA s'appuie sur JDBC pour communiquer avec une base de données.

JPA: Java persistence API (1)

- Avec JPA, l'architecture précédente devient la suivante :



Avantages:

- Avant, la couche [dao] est très dépendante du Framework ORM choisi.
 - Si le développeur décide de changer sa couche ORM, il doit également changer sa couche [dao] qui avait été écrite pour dialoguer avec un ORM spécifique
- Maintenant, le développeur va écrire une couche [dao] qui va dialoguer avec une couche JPA.
 - Quelque soit le produit qui implémente celle-ci, l'interface de la couche JPA présentée à la couche [dao] reste la même

JPA: Java persistence API (1)


- Le rôle de JPA est de faire abstraction du schéma relationnel qui sert à stocker les entités.
- Les métadonnées permettent de définir le lien entre votre conception orientée objet et le schéma de base de données relationnelles.
- Avec JPA le développeur d'application n'a plus à se soucier a priori de la base de données.
- Il lui suffit d'utiliser JPA, notamment **EntityManager**, et les interfaces de récupération d'objets persistants.

Mapping Objet Relationnel Avec JPA

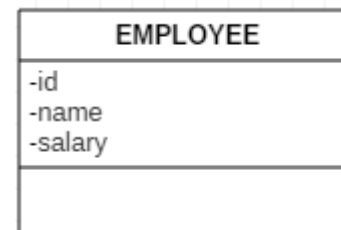
- La communication entre les mondes objet et relationnel suppose une transformation pour adapter la structure des données relationnelles au modèle objet.
- Les Entités
 - Les classes dont les instances peuvent être persistantes sont appelées des entités dans la spécification de JPA
 - Le développeur indique qu'une classe est une entité en lui associant l'annotation **@Entity**

Les entités: Exemple1

- Considérons une base de données ayant une unique table [**EMP**] dont le rôle est de mémoriser quelques informations sur des employés que nous voulons mapper avec la classe [**EMPLOYEE**] :
- La table **EMP** contient les champs suivants:
 - **ID**: clé primaire de la table
 - **EMP_NAME** : nom de la personne
 - **SALARY** : son salaire

Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
ID 	int(11)			Non	Aucun(e)		AUTO_INCREMENT
EMP_NAME	varchar(20)	utf8mb4_general_ci		Non	Aucun(e)		
SALARY	float			Non	Aucun(e)		

- La classe EMPLOYEE est caractérisée par les attributs suivants :
 - **id** : Identifiant de l'employé
 - **name** : nom de l'employé
 - **salary** : son salaire



Les entités: Exemple1 (suite)

- L'objet [EMPLOYEE] image de la table [EMP] présentée précédemment pourrait être le suivant :

```
@Entity
@Table(name = "EMP")
public class EMPLOYEE {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(name = "EMP_NAME")
    private String name;
    private double salary;
    public EMPLOYEE() {
        super();
    }
    public int getId() {return id;}
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public double getSalary() {return salary;}
    public void setSalary(double salary) {this.salary = salary;}
}
```

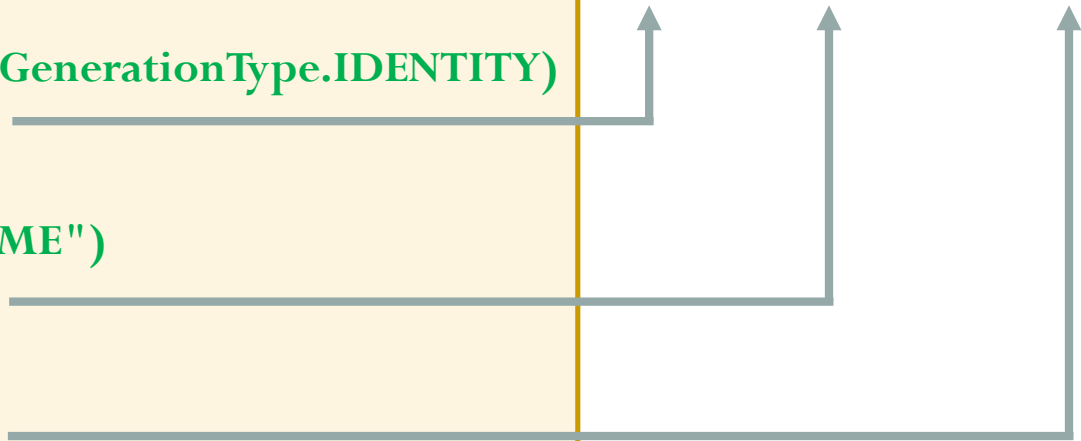
madani.a@ucd.ac.ma

Java Persistence API: *Mappings de base*

```
@Entity  
@Table(name = "EMP")  
public class EMPLOYEE {  
  @Id  
  @GeneratedValue(strategy = GenerationType.IDENTITY)  
  private int id;  
  
  @Column(name = "EMP_NAME")  
  private String name;  
  
  private double salary;  
  
  ....  
}
```

EMP

ID	EMP_NAME	SALARY



Annotation JPA

- L'annotation **@Entity** est la première annotation indispensable. Elle se place avant la ligne qui déclare la classe et indique que la classe en question doit être gérée par la couche de persistance JPA.
 - En l'absence de cette annotation, toutes les autres annotations JPA seraient ignorées.
- L'annotation **@Table** désigne la table de la base de données dont la classe est une représentation. Son principal argument est name qui désigne le nom de la table.
 - En l'absence de cet argument, la table portera le nom de la classe
- L'annotation **@Id** sert à désigner le champ dans la classe qui est image de la clé primaire de la table. Cette annotation est obligatoire.

Annotation JPA (suite)

- L'annotation **@Column** sert à faire le lien entre un attribut de la classe et la colonne de la table dont le champ est l'image.
 - L'attribut **name** indique le nom de la colonne dans la table. En l'absence de cet attribut, la colonne porte le même nom que le champ.
 - L'argument **nullable=false** indique que la colonne associée au champ ne peut avoir la valeur NULL et que donc le champ doit avoir nécessairement une valeur.
- L'annotation **@GeneratedValue** indique comment est générée la clé primaire lorsqu'elle est générée automatiquement par le SGBD.
 - GenerationType.AUTO (par défaut)
 - GenerationType.IDENTITY
(ne fonctionne pas bien avec le batch processing)
 - GenerationType.SEQUENCE
 - GenerationType.TABLE

Exemples de choix :

MySQL → IDENTITY ou AUTO

PostgreSQL / Oracle → SEQUENCE

Portabilité entre SGBD → AUTO

Performance importante → SEQUENCE (meilleur pour le batch processing)

Annotation JPA (suite : Exemples)

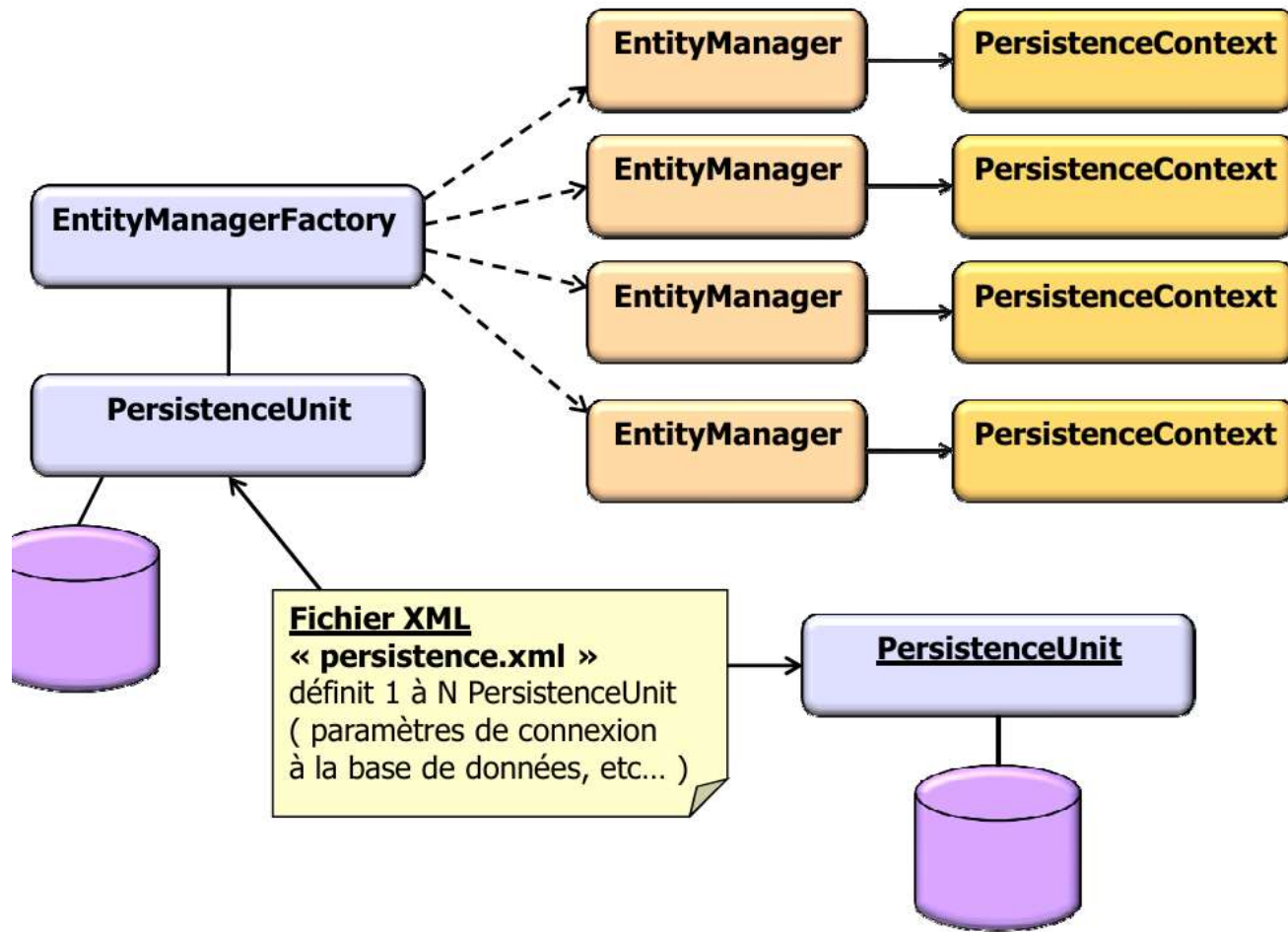
- `@Entity`
- `@Table`
- `@Id`
- `@GeneratedValue`
- `@SequenceGenerator`
- `@TableGenerator`
- `@Column`
- `@Transient`
- `@Lob`
- `@Temporal`
- `@Enumerated`
- `@Version`
- `@OneToOne`
- `@OneToMany`

- `@ManyToOne`
- `@ManyToMany`
- `@JoinColumn`
- `@JoinTable`
- `@MappedBy`
- `@NamedQuery`
- `@NamedNativeQuery`
- `@Transactional`
- `@Lock`
- `@Embeddable`
- `@Embedded`
- `@AttributeOverride`
- `@ElementCollection`

JPA : Quelques objets pour tout gérer

- Chaque base de données est une «Persistence Unit» décrite dans un fichier de configuration XML
- A chaque «Persistence Unit» correspond un «EntityManagerFactory»
- Les opérations de persistance reposent sur un objet central : «EntityManager» qui est fourni par l' «EntityManagerFactory» de la base de données à adresser

JPA : Quelques objets pour tout gérer



Exemple avec une base de données

```
<persistence-unit name="Cinema0">
<class>model.Film</class>
<properties>
    <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:8889/Cinema"/>
    <property name="javax.persistence.jdbc.user"
value="root"/>
    <property name="javax.persistence.jdbc.password"
value="root"/>
    <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
    <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5Dialect"/>
    <property name="hibernate.hbm2ddl.auto"
value="update"/>
    <property name="hibernate.show_sql" value="true"/>
</properties>
</persistence-unit>
</persistence>
```

Exemple avec 2 bases de données

<!-- Unité de persistance pour MySQL -->

<persistence-unit name="mysqlPU">

<class>com.example.entity.MySQLTable</class>

<properties>

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

<property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/mysql_db"/>

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

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

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

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

</properties>

</persistence-unit>

<!-- Unité de persistance pour PostgreSQL -->

<persistence-unit name="postgresPU">

<class>com.example.entity.PostgresTable</class>

<properties>

<property name="javax.persistence.jdbc.driver"
value="org.postgresql.Driver"/>

<property name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/postgres_db"/>

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

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

<property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect"/>

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

</properties>

</persistence-unit>

EntityManager : principales méthodes

- **persist(entity)** ajout d'une nouvelle entité
- **merge(entity)** mise à jour d'un entité (ajout si inex.)
- **remove(entity)** suppression d'une entité
- **find(type, key)** recherche d'une entité par son id
- **getReference(type, key)** idem mais renvoie un «proxy»
- **refresh(entity)** rafraichissement (DB entité)
- **lock(entity, mode)** verrouillage
- **contains(entity)** entité présente dans le contexte ?
- **flush()** force la mise à jour en base
- **clear()** vide le PersistenceContext
- **getTransaction()** récupère la transaction courante
- **close()** fin d'utilisation (ne commit pas)

EntityManager : principales méthodes

-find()-

Récupère une entité par son identifiant (ID).

```
User user = entityManager.find(User.class, 1); //
```

Récupère l'utilisateur avec l'ID 1

Cette méthode retourne l'entité correspondante, ou null si elle n'existe pas.

EntityManager : principales méthodes

-persist()-

Insère une nouvelle entité dans la base de données.

```
User user = new User();  
user.setName("John Doe");  
user.setEmail("john@example.com");  
entityManager.getTransaction().begin();  
entityManager.persist(user); // Insère l'entité  
entityManager.getTransaction().commit();
```

EntityManager : principales méthodes

-merge()-

Met à jour une entité existante ou en insère une nouvelle si elle n'existe pas.

```
User user = entityManager.find(User.class, 1);  
user.setEmail("new-email@example.com");  
entityManager.getTransaction().begin();  
entityManager.merge(user); // Met à jour l'entité  
entityManager.getTransaction().commit();
```

EntityManager : principales méthodes

-remove()-

Supprime une entité de la base de données.

```
User user = entityManager.find(User.class, 1L);  
entityManager.getTransaction().begin();  
entityManager.remove(user); // Supprime l'entité  
entityManager.getTransaction().commit();
```


EntityManager : principales méthodes

-createQuery()-

Crée une requête JPQL (Java Persistence Query Language) pour récupérer des données.

```
String req= "SELECT u FROM User u";  
//String req= "FROM User u";  
Query query = entityManager.createQuery(req)  
List<User> users = query.getResultList();
```

Fichier de configuration

- **Un seul fichier de configuration XML situé dans META-INF**
- **IL est nécessaire d'indiquer au fournisseur de persistance comment il peut se connecter à la base de données**
- **Ce fichier peut aussi comporter d'autres informations**

Fichier de configuration

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

```
<persistence ....>
```

```
<!-- Define a name used to get an entity manager. Define that you will complete transactions with the DB -->
```

```
<persistence-unit name="Test_JPA">
```

```
<!-- Define the class for Hibernate which implements JPA -->
```

```
<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

```
<!-- Define the object that should be persisted in the database -->
```

```
<class>Metier.Etudiant</class>
```

```
<properties>
```

```
...
```

```
</properties>
```

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

```
</persistence>
```

Fichier de configuration

```
<properties>
```

```
<!-- URL for DB -->
```

```
<property name="javax.persistence.jdbc.url"  
value="jdbc:mysql://localhost:3306/base1" />
```

```
<!-- Username -->
```

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

```
<!-- Password -->
```

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

```
<!-- Driver for DB database -->
```

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

```
<property name="hibernate.dialect"  
value="org.hibernate.dialect.MySQL5Dialect" />
```

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

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

```
</properties>
```

Modèle d'une application Desktop

```
public static void main(String[] args) {  
    EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("Test_JPA");  
    // Récupération d'une instance de EntityManager  
    EntityManager em = emf.createEntityManager();  
    // Utilisation de EntityManager  
    ...  
    //Fermeture de l'EntityManager  
    em.close();  
    emf.close();  
}
```

Exemple de JPA Project

Dans cet exemple, nous allons voir une utilisation des méthodes suivantes :

- Objet EntityManager
 - persist()
 - merge()
 - find()
 - remove()
- Objet Query
 - getResultList()
 - getSingleResult()