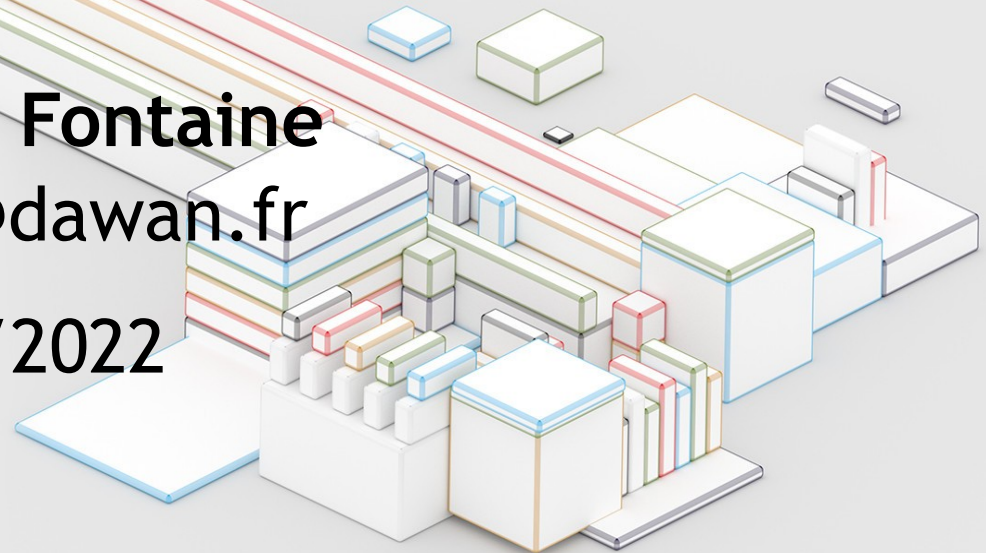


# JPA / Hibernate

**Christophe Fontaine**  
[cfontaine@dawan.fr](mailto:cfontaine@dawan.fr)

**28/11/2022**



# Objectifs

---



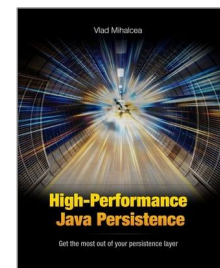
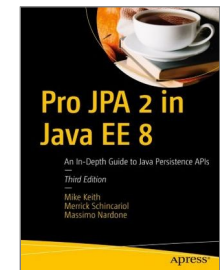
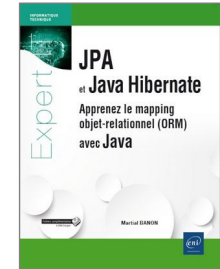
- Implémenter une couche de persistance performante avec JPA/Hibernate

**Durée :** 5 jours

**Pré-requis :** Maîtrise de la programmation orientée objets en JAVA  
Notions de SQL

# Bibliographie

- **JPA et Java Hibernate**  
Apprenez le mapping objet-relationnel (ORM) avec Java  
Martial Banon  
Éditions ENI - janvier 2017
- **Pro JPA 2 in Java EE 8 (3rd édition)**  
An In-Depth Guide to Java Persistence APIs  
Mike Keith, Merrick Schincariol, Massimo Nardone  
Apress - 3<sup>rd</sup> édition - février 2018
- **Hibernate Tips**  
More than 70 solutions to common Hibernate  
Thorben Janssen - mars 2017
- **High-Performance Java Persistence**  
Get the most out of your persistence layer  
Vlad Mihalcea - octobre 2016



# Bibliographie



- **JPA 2.2 spécification**

[https://download.oracle.com/otndocs/jcp/persistence-2\\_2-mrel-eval-spec/index.html](https://download.oracle.com/otndocs/jcp/persistence-2_2-mrel-eval-spec/index.html)

- **JPA tutorial**

<https://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html>

- **Documentation Hibernate**

[https://docs.jboss.org/hibernate/orm/current/userguide/html\\_single/Hibernate\\_User\\_Guide.html](https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html)

- **Tutoriaux**

- <https://thorben-janssen.com/tutorials/>
- <https://vladmihalcea.com/tutorials/hibernate/>

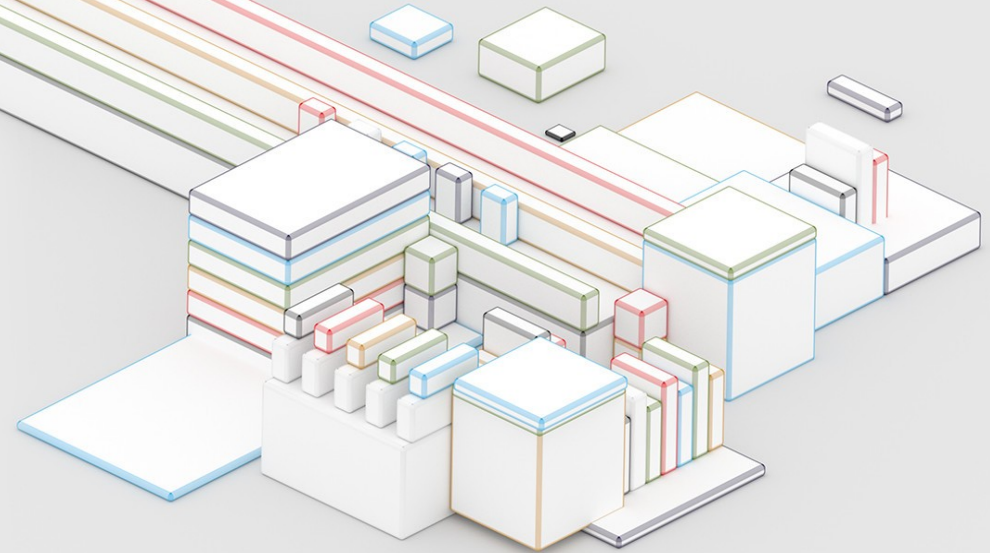
# Plan

---



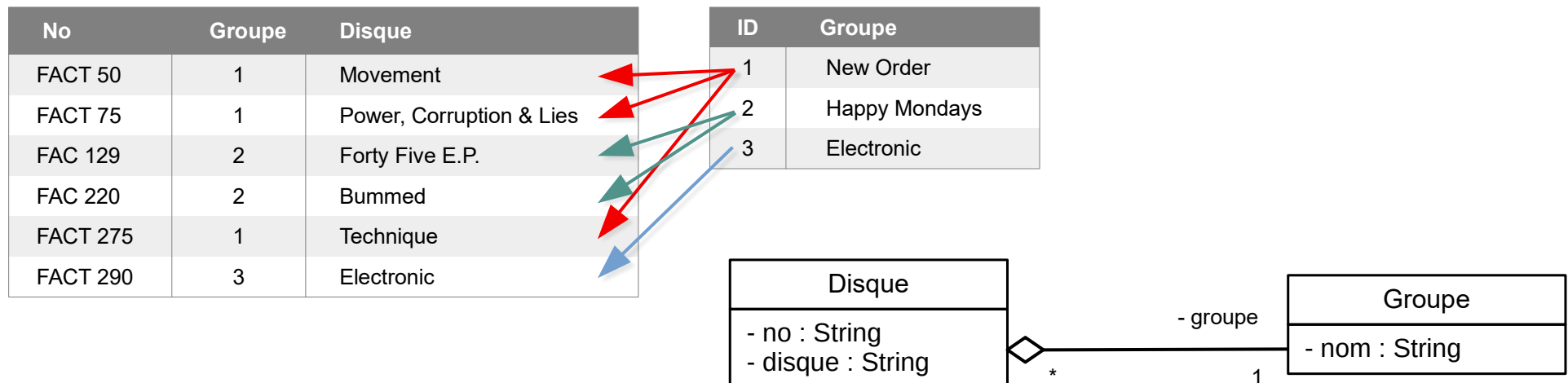
- Découvrir JPA Hibernate
- Mapping des entités
- Relations entre entités
- Maîtriser le langage à requêtes objet
  - Requête JPQL
  - API Criteria
  - Requêtes native SQL
- Éléments avancés de Hibernate
  - Intercepteurs, event-listeners
  - Gestion des caches

# Découvrir JPA Hibernate



# Correspondance des modèles " Relationnel - Objet "

- Le modèle objet propose plus de fonctionnalités :
  - L'héritage
  - Le polymorphisme
- Les relations entre deux entités sont différentes
- De nombreux types de données sont différents
- Les objets ne possèdent pas d'identifiant unique contrairement au modèle relationnel



# Accès aux Bdd en Java

---



- JDBC (Java DataBase Connectivity)
- Inconvénients :
  - Nécessite l'écriture de nombreuses lignes de codes répétitives
  - La liaison entre les objets et les tables est un travail de bas niveau
- Exemple de code + pattern DAO



# Mapping relationnel-objet

---



Concept permettant de connecter un modèle objet à un modèle relationnel

Couche qui va interagir entre l'application et la base de données

- **Pourquoi utiliser ce concept ?**
  - Pas besoin de connaître l'ensemble des tables et des champs de la base de données
  - Faire abstraction de toute la partie SQL d'une application

# Mapping relationnel-objet

Développeur

Objet

Personne

id  
nom  
prenom

ORM

Système

SGBD

personnes

Id	Nom	Prenom
...	...	...

# Mapping relationnel-objet

---



- **Avantages :**
  - Gain de temps au niveau du développement d'une application
  - Abstraction de toute la partie SQL
  - La portabilité de l'application d'un point de vue SGBD
- **Inconvénients :**
  - L'optimisation des frameworks/outils proposés
  - La difficulté à maîtriser les frameworks/outils

# Critères de choix d'un ORM

---



- La facilité du mapping des tables avec les classes, des champs avec les attributs
- Les fonctionnalités de bases des modèles relationnel et objet
- Les performances et optimisations proposées : gestion du cache, chargement différé
- Les fonctionnalités avancées : gestion des sessions, des transactions
- Intégration IDE : outils graphiques
- La maturité

# JPA

- Une API (Java Persistence API)
- Des implémentations



- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

# Concepts vs Classes

Concept	JDBC	Hibernate	JPA
<b>Ressource</b>	Connection	Session	EntityManager
<b>Fabrique de ressources</b>	DataSource	SessionFactory	EntityManagerFactory
<b>Exception</b>	SQLException	HibernateException	PersistenceException

# Objets Hibernate (JPA)



- **SessionFactory (EntityManagerFactory)**
  - Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données
  - Une factory (fabrique) de Session et un client de ConnectionProvider
  - Peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au niveau processus ou au niveau cluster
- **Session (EntityManager)**
  - Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance
  - Encapsule une connexion JDBC, une Factory (fabrique) des objets Transaction

# Objets Hibernate (JPA)



- Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire
- Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant
- **Objets et Collections persistants**
  - Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier
  - Ceux-ci sont en général les objets métier  
La seule particularité est qu'ils sont associés avec une (et une seule) Session
- **Objets et collections transitoires (Transient)**
  - Instances de classes persistantes qui ne sont actuellement pas associées à une Session



# Objets Hibernate (JPA)



- Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elles n'ont pu être instanciées par une Session fermée
- **Transaction (EntityTransaction)** (Optionnel)
  - Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique
  - Abstrait l'application des transactions sous-jacentes
  - Une Session peut fournir plusieurs Transactions dans certains cas
- **TransactionFactory** : (Optionnel)
  - Une fabrique d'instances de Transaction
  - Non exposé à l'application, mais peut être étendu/implémenté par le développeur

# Configuration de Spring Tool Suite 4



- À partir de eclipse 4.18, le JRE qui va exécuter eclipse est intégrée sous forme de plugin (openjdk 17)
- Il n'est plus nécessaire de le configurer dans le fichier **eclipse.ini** avec l'option **-vm**

**-vm**

C:\Program Files\Java\jdk1.8.0\_351

- **Ajout des plug-ins**
  - Menu → Help → Install new software
  - Work with : Choisir → All Available Sites
  - **Web, XML,Java EE and OSGI Enterprise Development**
  - Choisir :
    - Eclipse Enterprise Java and Web Developer Tools 3.27

# Configuration de Spring Tool Suite 4



- Dali Java Persistence Tools - JPA Support
- Dali JavaPersistence Tools - JPA Diagram Editor
- m2e-wtp-JPA configurator for WTP

## – Database développement

Choisir tout et décocher le support des bdd non utilisés

- Pour désinstaller les plugins :  
menu → about spring tools suite 4 → installation détails
- Dans windows → préférence

## – filtre sur **jre**

Installed JREs → Add → choisir :

- Standard VM
- JRE home : C:\Program Files\Java\jdk1.8.0\_351\jre
- JRE Name : jdk1.8.0\_351

# Configuration de Spring Tool Suite 4

---



- filtre sur **compiler**  
JDK Compliance → Compiler compliance level → 1.8
- filtre sur **text editors**  
cocher : Insert spaces for tabs  
cocher : remove multiple spaces and backspace/delete
- filtre sur **spelling**  
décocher : enable spelling
- filtre sur **encoding**  
CSS Files, HTML Files, JSP Files : sélectionner UTF-8
- filtre sur **formatter**  
java → code style → Formatter  
new → profil name : Eclipse  
indentation : tab policy choisir **space only**

# Configuration du projet



- Créer un projet Maven avec l'archetype :  
**maven-archetype-quickstart**
- Pour compiler en java 8 avec maven, ajouter dans l'élément **<properties>** de pom.xml :

```
<maven.compiler.target>1.8</maven.compiler.target>  
<maven.compiler.source>1.8</maven.compiler.source>
```

- **Ajouter le dossier resources**
  - Ajouter dans le projet un dossier qui a pour chemin :  
**src/main/resources**
- **Maven update**  
projet → maven → update project

# Configuration du projet



- Ajouter les dépendances dans **pom.xml**

- **Hibernate**

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.14.Final</version>
</dependency>
```

- **Driver JDBC du SGBD (MySQL)**

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.31</version>
</dependency>
```

# Configuration du projet

- **project facet**


- projet → propriétés: filtre sur **project facet**  
cliquer sur le lien: [convert to faceted form...](#)

- Java → 1.8
- Jpa → 2.2

- cliquer sur le lien: [further configuration available](#)

**JPA implementation type** → choisir : Disable Library Configuration

**connection** : cliquer sur le lien: [add connection](#)

- filtre sur **MySQL**
- name : MySQL 8.0
- cliquer sur  → new driver definition

# Configuration du projet



- onglet **Name/Type**
  - Driver name : MySQL JDBC 8.0
- onglet **Jar Files**
  - remplacer le driver existant par :  
`C:\Users\compte utilisateur windows\.m2\repository\mysql\mysql-connector-java\8.0.31\mysql-connector-java-8.0.31.jar`
- onglet **properties**
  - **connection URL**: `jdbc:mysql://localhost:3306/nom_bdd`
  - **database Name**: `nom_bdd`
  - **driver class**: `com.mysql.cj.jdbc.Driver`
  - **user Id**: `root`
  - **password** :



# Paramétrage de l'ORM



- Le fichier **persistence.xml**, permet de configurer l'ORM  
Il se trouve dans le dossier **META-INF** du projet  
(Il faut déplacer ce dossier dans src/main/resources)
- **L'unité de persistance <persistence-unit>**  
Elle représente la configuration à charger, elle pour attribut:
  - **name** : nom pour l'appeler depuis le code source
  - **transaction-type** : le type de transaction
    - **RESOURCE\_LOCAL**: l'application est responsable des transaction (création, suivie...)
    - **JTA**: Le **serveur d'application** est responsable de la gestion des transaction

```
<persistence-unit name="formationjpa" transaction-type="RESOURCE_LOCAL">  
...  
</persistence-unit>
```

# Paramétrage de l'ORM



- **L'implémentation de JPA utilisée <provider>**  
pour hibernate:

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

- **Les entités à mapper**

On définit les classes qui vont être mappées

- La balise <mapping-file> permet d'indiquer le fichier de mapping

```
<mapping-file>\META-INF\orm.xml</mapping-file>
```

- La balise <class> permet d'indiquer les classes des entités

```
<class>fr.dawan.projetjpa.beans.Personne</class>
```

# Paramétrage de l'ORM



- Les propriétés du fichier de persistance se trouve dans la balise **<properties>**

Une propriété a pour balise **<property>** avec 2 attributs:

- **name**: nom de la propriété
- **value**: valeur de la propriété

- **Les propriétés de la base de données (obligatoires) :**

- **javax.persistence.jdbc.driver** → le driver de connexion à la base de données

pour MySQL 8: `com.mysql.cj.jdbc.Driver`

- **javax.persistence.jdbc.url** → l'URL de la base de données

pour MySQL 8: `jdbc:mysql://localhost:3306/Nom_bdd`

# Paramétrage de l'ORM



- `javax.persistence.jdbc.user` → nom du compte pour se connecter à la bdd
- `javax.persistence.jdbc.password` → mot de passe du compte
- **Les propriétés d'hiberbate**
  - `hbm2ddl.auto` → création automatique des tables de base de données  
a pour valeur :
    - **validate** (valeur par défaut) → hibernate valide la structure de la table  
si la table n'existe pas, hibernate lève une exception
    - **create** → hibernate supprime les tables existantes (données et structure) et crée les nouvelles tables

# Paramétrage de l'ORM



- **update** → hibernate vérifie la table et les colonnes
  - si une table n'existe pas, elle crée de nouvelles tables
  - si une colonne n'existait pas, elle crée de nouvelles colonnes

hibernate ne supprime aucune table existante, il n'y a aucune perte de données dans les tables existantes

- **create-drop** → hibernate commence par rechercher une table, puis effectue les opérations nécessaires, puis supprime la table une fois toutes les opérations terminées
- **none** → création automatique des tables de base de données

# Paramétrage de l'ORM



- **hibernate.dialect** → permet à Hibernate de générer du SQL optimisé pour une base de données relationnelle (**liste**)

MySql 8                      `org.hibernate.dialect.MySQL8Dialect`

MariaDb                    `org.hibernate.dialect.MariaDB106Dialect`

H2                            `org.hibernate.dialect.H2Dialect`

SQLServer 2016   `org.hibernate.dialect.SQLServer2016Dialect`

PostgreSQL 10    `org.hibernate.dialect.PostgreSQL10Dialect`

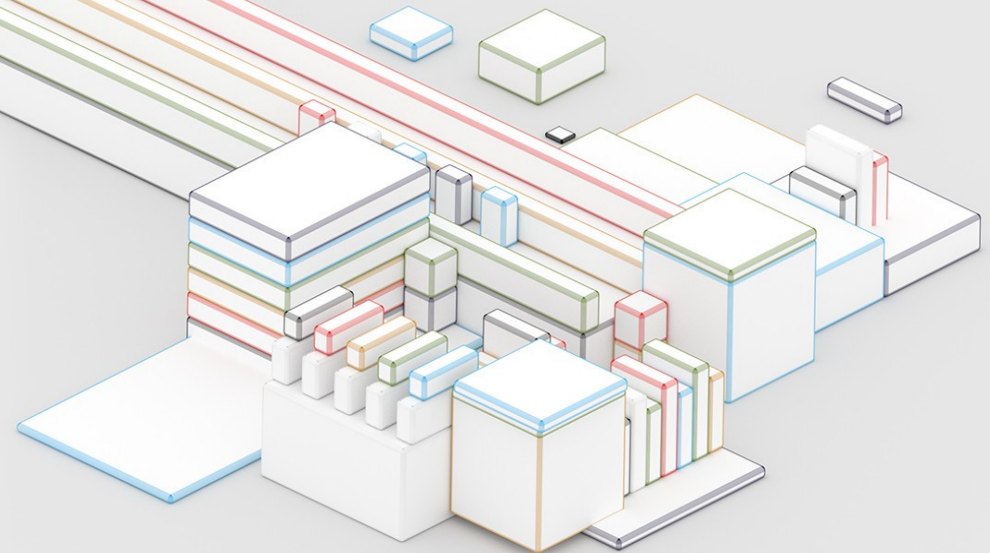
- **hibernate.show\_sql** → écrit toutes les requêtes SQL dans la console
- **hibernate.format\_sql** → formate le SQL dans la console et dans les log

# persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
  <persistence-unit name="formationjpa">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>fr.dawan.formation.entities.Personne</class>
    <properties>
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/formationjpa"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.cj.jdbc.Driver"/>

      <property name="hibernate.hbm2ddl.auto" value="create"/>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQL8Dialect"/>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

# Mapping des entités





# EntityManager



- L'objet **EntityManager** (Session) possède un ensemble de méthodes permettant d'effectuer des opérations en Bdd :
  - **recherche** : find, get
  - **sauvegarde** : persist, merge
  - **suppression** : remove
  - **requêtage** : langage JPQL, createQuery
  - **transaction** : begin, commit, rollback  
(support des transactions par annotations)

# EntityManager et Transaction



- L'objet **EntityManagerTransaction** permet de :
  - valider une action de la session : **commit()**;
  - annuler une action de la session : **rollback()**;

```
EntityManagerTransaction tx=null;
try{
    //...
    tx = em.getTransaction().begin();
    //...
    tx.Commit();
} catch(Exception ex){
    tx.Rollback();
}
em.Close();
```

# Entité



- Dans la spécification JPA, Une entité est une classe dont les instances peuvent être persistées en base de données
- Une entité est une classe Java standard qui doit :
  - être identifiée comme une entité avec l'annotation **@Entity**
  - avoir un attribut qui joue le rôle d'identifiant annoté avec **@Id** (représente la clé primaire de la table)
  - avoir un **constructeur sans argument**
  - Implémenter l'interface **Serializable**
  - ne doit pas être **final**
  - aucune de ses méthodes ne peut être **final**

# Entité

```
@Entity
public class Personne implements Serializable {
    @Id
    private long id;
    private String firstName;
    private String name;
    public Personne() {
    }
    // Getters / Setters
}
```

- **Nom de l'entité**

Par défaut, le nom de l'entité est le nom de la classe

L'attribut **name** de l'annotation **@Entity** permet de donner un autre nom à l'entité

```
@Entity(name="dept")
public class Departement { //... }
```

- **Nom de la table**

Par défaut, le nom de table associé est le nom de la classe

On peut le changer avec l'annotation :

**@Table**(name = "Nom\_table")

```
@Entity
@Table(name="personnes")
public class Personne {

    // ...

}
```

# Attributs persistants

- Par défaut, tous les attributs d'une entité sont persistants
- Les attribut qui ne sont pas persister sont ceux :
  - qui ont pour annotation **@Transient**
  - dont la variable de l'attribut est **transient**
  - qui sont **final** et/ou **static**

```
@Entity
public class NoPersist{
    @Transient
    int attr1;
    transient int attr2;
    static int attr3;
    final int attr4=0;
}
```

# Énumération

- Une énumération est stockée par défaut sous forme numérique (0,1,... n)
- L'annotation `@Enumerated` permet de définir comment l'énumération sera stockée
  - `EnumType.ORDINAL` stockée sous forme numérique
  - `EnumType.STRING` le nom de l'énumération est stocké

```
public enum Civilite{  
    MADemoiselle, MADame, MONSIEUR  
}  
  
@Entity  
public class Personne{  
    @Enumerated(EnumType.STRING)  
    private Civilite civ; // Stocke MADemoiselle,  
                           MADame ou MONSIEUR  
                           dans la base de données  
    // ...  
}
```

# Attribut temporel

- Lorsqu'un attribut est de type temporel de `java.util` (`Calendar` ou `Date`), il faut indiquer le type temporel avec l'annotation `@Temporal`
- Elle peut prendre comme paramètre:
  - `TemporalType.DATE` : ne stocke que le jour
  - `TemporalType.TIME`: ne stocke que l'heure
  - `TemporalType.TIMESTAMP`: stocke le jour et l'heure

```
@Temporal(TemporalType.DATE)  
private Date birthday;
```

- JPA 2.2 et hibernate 5 supportent les nouveaux type Date Java 8 du package `java.time`

```
private LocalDateTime localTime;  
private LocalDate localDate;  
private LocalDateTime localDateTime;
```



# Large OBject

- L'annotation **@Lob** indique que l'attribut de l'entité est un type de données de longueur variable pour stocker des objets volumineux (Large OBject)
- Le type de données peut être un :
  - **CLOB** (Character Large Object) pour stocker du texte
  - **BLOB** (Binary Large Object) pour stocker des données binaires (images, audio ...)
- Un BLOB sera stocké dans un tableau d'octet

```
@Entity
public class User {
    @Id
    private long id;
    @Lob
    private byte[] photo;
```

# Propriétés de la colonne



- Par défaut, une colonne de la table aura le nom de l'attribut correspondant
- L'annotation **@Column** permet pour définir plus précisément la colonne, avec les attributs suivant :
  - **name**: permet de définir le nom de la colonne
  - **unique**: permet de définir si le champs doit être unique (par défaut à **false**)
  - **nullable**: permet de définir si le champ peut être **null** (par défaut à **true**)
  - **length**: pour les chaînes de caractères, permet de définir la longueur (par défaut 255)
  - **precision**: permet de définir la précision pour un nombre décimal (par défaut 0)

# Propriétés de la colonne



- **scale**: permet de définir l'échelle d'un nombre décimal (par défaut 0)
- **insertable**: permet de définir, si la colonne est prise en compte pour une requête insert (par défaut à true)
- **updatable**: permet de définir si la colonne est prise en compte pour une requête update (par défaut à true)
- **ColumnDefinition**: permet de donner en SQL, le code de création d'une colonne (DDL). En général, à éviter

Permet de donner un valeur par défaut à une colonne :

```
@Column(columnDefinition=" default '10' ")
```

```
@Column(name="family_name", length=50)  
private String nom ;  
@Column(name="first_name", length=50)  
private String prenom ;
```

# Classe Intégrable

- Une classe intégrable va stocker ses données dans la table de l'entité mère ce qui va créer des colonnes supplémentaires
- La classe intégrable est annotée avec **@Embeddable**
- L'attribut de l'objet dans la classe mère doit utiliser l'annotation **@Embedded**

```
@Embeddable
public class PersonneDetail{
    private LocalDate birthday;
}
```

```
@Entity
public class Personne{
    @Id
    private long id;
    private String prenom;
    private String nom;
    @Embedded
    private PersonneDetail detail;
    //...
}
```

id	prenom	nom	birthday

# Utilisation multiple d'une classe intégrable

---



- Une classe entité peut référencer plusieurs instances d'une même classe intégrable
- Les noms des colonnes dans la table de l'entité ne peuvent être les mêmes pour chacune des utilisations
- Un champ annoté par `@Embedded` peut être complété par une annotation `@AttributeOverride`, ou plusieurs insérées dans une annotation `@AttributeOverrides`
- Elles permettent d'indiquer le nom d'une ou de plusieurs colonnes dans la table de l'entité

# Utilisation multiple d'une classe intégrable

```
@Embeddable
public class Adresse {
    private String rue;
    private String ville;
    private int codePostal;
    ...
}
```

```
@Entity
public class Employe {
    @Embedded
    private Adresse adresse;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(
            name="ville",
            column=
            @column(name="ville_travail")),
        @AttributeOverride( ... )
    })
    // Adresse du travail
    private Adresse adresseTravail;
    // ...
}
```

# Clé primaire

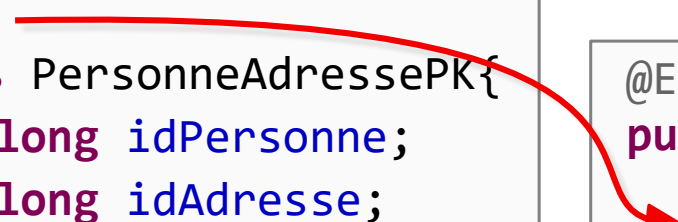


- Une entité doit avoir un attribut qui correspond à la clé primaire dans la table associée
- **Clé primaire simple**  
Une entité a un attribut unique qui sert de clé primaire  
L'attribut clé primaire est désigné par l'annotation **@Id**
- **Clé primaire composée**  
Une clé primaire peut être composée de plusieurs colonnes  
  
Pour mapper une clé primaire composée, on crée une classe intégrable **@Embeddable** qui ne contient que les champs de la clé primaire et on l'utilise dans l'entité principale avec l'annotation **@EmbeddedId**

# Clé primaire

```
@Embeddable
public class PersonneAdressePK{
    private long idPersonne;
    private long idAdresse;
    public PersonneAdressePK(){
    }
    // ...
}
```

```
@Entity
public class PersonneAdresse {
    @EmbeddedId
    protected PersonneAdressePK pAddrPK;
    //...
}
```



- Pour une clé primaire, on peut utiliser les types suivant :
  - **Type primitif** : byte, int, short, long et char
  - **Classes wrapper** : Byte, Integer, Short, Long et Character
  - String
  - java.math.BigInteger
  - java.util.Date et java.sql.Date



# Génération automatique de clé primaire



- L'annotation **@GeneratedValue** indique que la clé primaire est générée automatiquement lors de l'insertion en Bdd
- Elle doit être utilisée en complément de l'annotation **@id**
- Elle a 2 attributs :
  - **generator** : contient le nom du générateur à utiliser
  - **strategy** : permet de spécifier le mode de génération de la clé primaire
- **GenerationType.AUTO** (par défaut)  
La génération est gérée par l'implémentation de l'ORM Hibernate crée une séquence unique via la table **hibernate \_sequence**

# Génération automatique de clé primaire



- **GenerationType.IDENTITY**

La génération se fait à partir d'une propriété entity propre au système de gestion de bdd

- **GenerationType.TABLE**

La génération s'effectue en utilisant une table pour assurer l'unicité. Hibernate crée une table **hibernate\_sequence** qui stocke les noms et les valeurs des séquences à utiliser avec l'annotation **@TableGenerator**

```
@Entity
@TableGenerators(name="personneGen")
public class Personne{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE
                    ,generator = "personneGen")
    private Integer id;
}
```

# Génération automatique de clé primaire



- **GenerationType.SEQUENCE**

La génération se fait par une séquence définie par le système de gestion de bdd

À utiliser avec l'annotation **@SequenceGenerator**

```
@Entity
@SequenceGenerator(name="seqGen")
public class Personne{
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
                    generator = "seqGen")
    private Integer id;
}
```

# Héritage



- Il existe trois façons d'organiser l'héritage :
- **SINGLE\_TABLE**  
@Inheritance  
@DiscriminatorColumn  
@DiscriminatorValue
- **TABLE\_PER\_CLASS**  
@Inheritance
- **JOINED**  
@Inheritance
- La différence entre elles se situe au niveau de l'optimisation du stockage et des performances

# Héritage : SINGLE\_TABLE

- Tout est dans la même table
- Une colonne, appelée "**Discriminator**" définit le type de la classe enregistrée
- De nombreuses colonnes inutilisées

```
@Entity
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="compte_discriminator",
discriminatorType=DiscriminatorType.STRING,length=15)
public abstract class Compte implements Serializable{...}

@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte
implements Serializable {...}
```

# Héritage : TABLE\_PER\_CLASS



- Chaque Entity Bean fils a sa propre table
- Lourd à gérer pour le polymorphisme

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Compte implements Serializable{
    //...
}
```

La clé **@Id** ne peut pas être **@GeneratedValue** avec la stratégie **Identity**

```
@Entity
public class CompteEpargne extends Compte
implements Serializable {
    //...
}
```

# Héritage : JOINED

- Chaque Entity Bean a sa propre table
- Beaucoup de jointures

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public abstract class Compte implements Serializable {
    //...
}

@Entity
public class CompteEpargne extends Compte
implements Serializable {
    //...
}
```

# Héritage : récapitulatif

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
<b>Avantages</b>	Aucune jointure, donc très performant	Performant en insertion	Intégration des données proche du modèle objet
<b>Inconvénients</b>	Organisation des données non optimale	Polymorphisme lourd à gérer	Utilisation intensive des jointures, donc baisse des performances



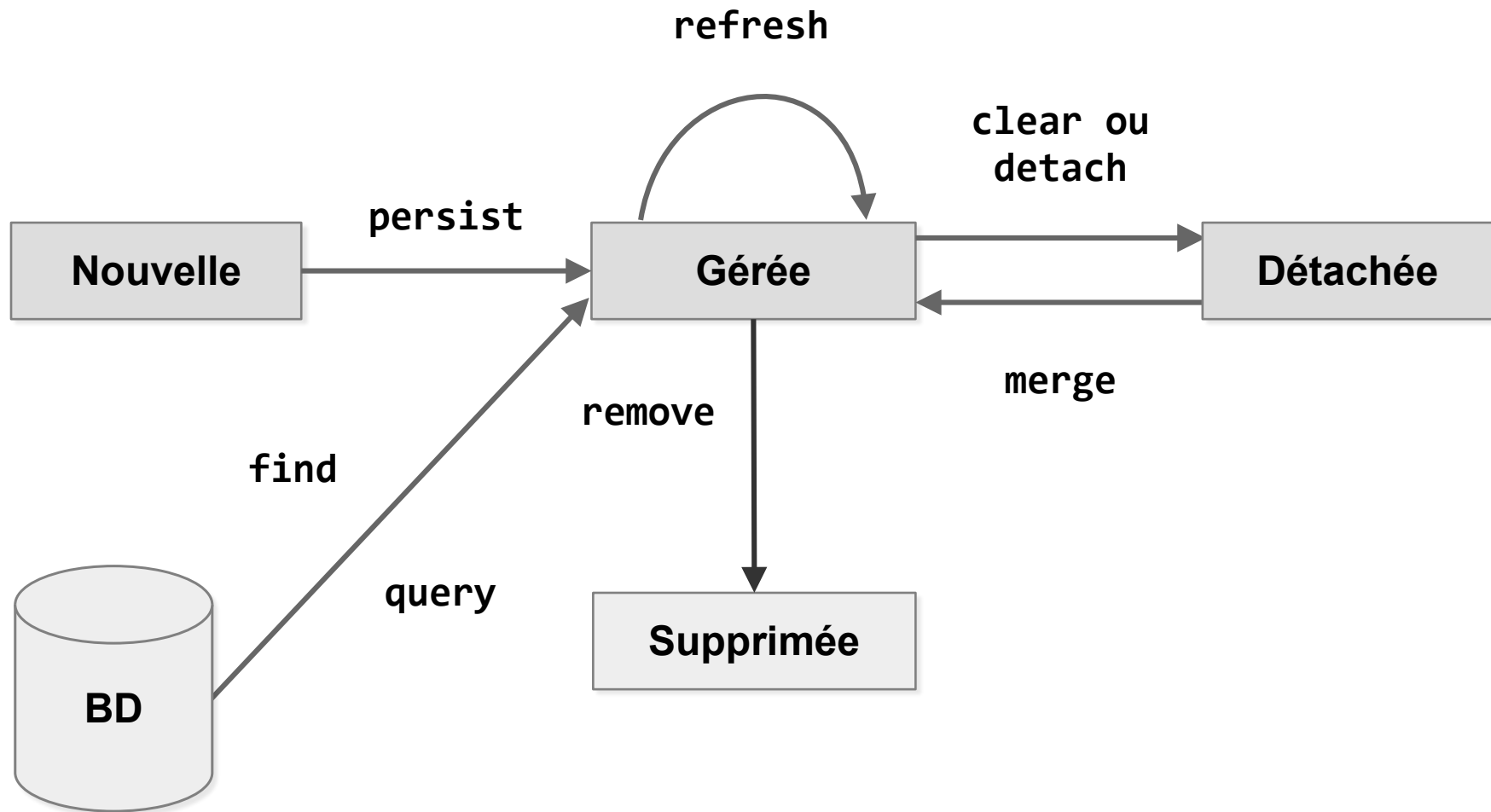
# Classe mère persistante



- Une entité peut aussi avoir une classe mère dont l'état est persistant, sans que cette classe mère ne soit une entité
- La classe mère a pour annotation **@MappedSuperclass**
- Aucune table ne correspondra à cette classe mère dans la base de données. L'état de la classe mère sera rendu persistant dans les tables associées à ses classes entités filles

```
@MappedSuperclass
public abstract class Base {
    @Id @GeneratedValue
    private Long Id;
    @ManyToOne
    private User user;
    ...
}
```

# Cycle de vie d'une entité



# Gestion de la concurrence



- La gestion de la concurrence est essentielle dans le cas de longues transactions
- Hibernate possède plusieurs modèles de concurrence :
  - **None** : la transaction concurrentielle est déléguée au SGBD → Elle peut échouer
  - **Optimistic (Versioned)** : si on détecte un changement dans l'entité, nous ne pouvons pas la mettre à jour  
**@Version(Numeric, Timestamp, DB Timestamp)**  
↳ On utilise une colonne explicite Version (meilleure stratégie)
  - **Pessimistic** : utilisation des LockMode spécifiques à chaque SGBD

# Gestion de la concurrence

## Versioned



- L'élément **@Version** indique que la table contient des enregistrements versionnés
- La propriété est incrémentée automatiquement par Hibernate
- Automatiquement, la requête générée inclura un test sur ce champ :

```
UPDATE Player SET version = @p0, PlayerName = @p1  
WHERE PlayerId = @p2  
AND version = @p3;
```

# Gestion de la concurrence

## Pessimistic



- On peut exécuter une commande séparée pour la base de données pour obtenir un verrou sur la ligne représentant l'entité :

```
Player player = entityManager.find(Player.class,1);
entityManager.lock(player, LockModeType.WRITE);
player.playerName = "other";
tx.commit();
```

```
SELECT PlayerId FROM Player with (updlock, rowlock)
WHERE PlayerId = @p0;
```

```
UPDATE Player SET PlayerName = @p1
WHERE PlayerId = @p2 AND PlayerName = @p3;
```

- **Inconvénient** : l'attente pour l'obtention du verrou (pour la modification si la ligne est verrouillée)  
Une exception est déclenchée après le Timeout parce que nous ne pouvons pas obtenir le verrou :

```
<property name="javax.persistence.lock.timeout" value="1000"/>
```

# Mapping des collections simples



- **@ElementCollection** sur une collection simple permet de générer une table **NomClasse\_nomVariable**
- **@CollectionTable** permet de personnaliser de la table

```
@ElementCollection(targetClass = String.class)
@CollectionTable(name = "prod_comments" , joinColumns =
@JoinColumn(name = "prod_id"))
private List<String> comments;
```

- **@MapKeyColumn** permet de personnaliser, la colonne de la clé dans la table pour la colonne valeur, on utilise **@Column**

```
@ElementCollection
@CollectionTable(name="EMP_PHONE")
@MapKeyColumn(name="PHONE_TYPE")
@Column(name="PHONE_NUM")
private Map<String, String> phoneNumbers;
```

# Gérer les exceptions

- Récupérer les **DataAccessException**

La cause de l'exception est l'erreur SQL

```
try {  
    dao.delete(delete1);  
} catch (DataAccessException e) {  
    System.out.println(e.getMessage());  
    System.out.println(e.getCause());  
    SQLException sqlEx = (SQLException) e.getCause();  
    System.out.println(sqlEx.getErrorCode());  
    System.out.println(sqlEx.getSQLState());  
}
```

# DAO



- Le pattern **DAO** (Data Access Object) permet d'isoler la couche métier de la couche de persistance
  - Permet de centraliser les requête SQL dans un seul objet
  - Permet de changer facilement de système de stockage de données (Bdd, XML ... )
- Avec l'objet DAO, on va réaliser les opérations CRUD
  - Créer l'objet en base (INSERT)
  - Rechercher l'objet en base pour le recréer (find, RETRIEVE)
  - Mettre à jour l'objet en base (UPDATE)
  - Supprimer l'objet en base (DELETE)
- On aura un DAO par objet Métier





**Plus d'informations sur <http://www.dawan.fr>**

**Contactez notre service commercial au  
09.72.37.73.73 (prix d'un appel local)**