



make it **clever**





**JPA**

**JÉRÉMY PERROUAULT**



# JPA

MAPPING ORM

# PRÉSENTATION ORM

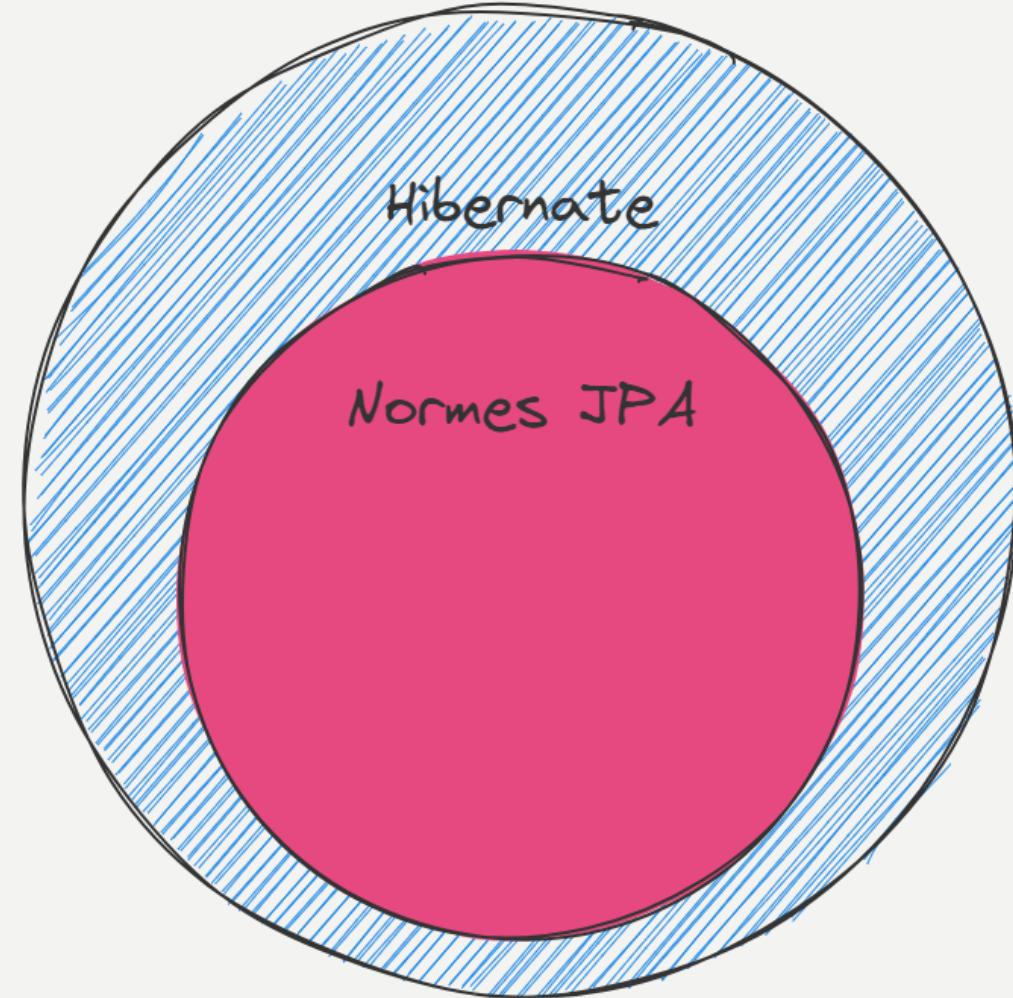
- **Object Relational Mapping**
- Fourni à l'application une **API** de plus haut niveau
- Permet d'éviter d'écrire du code **DML** et **DDL** fastidieux et répétitif
- Surcouche qui permet un accès au **SGBDR**
  - Plus conforme à la vision objet
- Prend en charge la communication avec le **SGBDR**

# INTRODUCTION

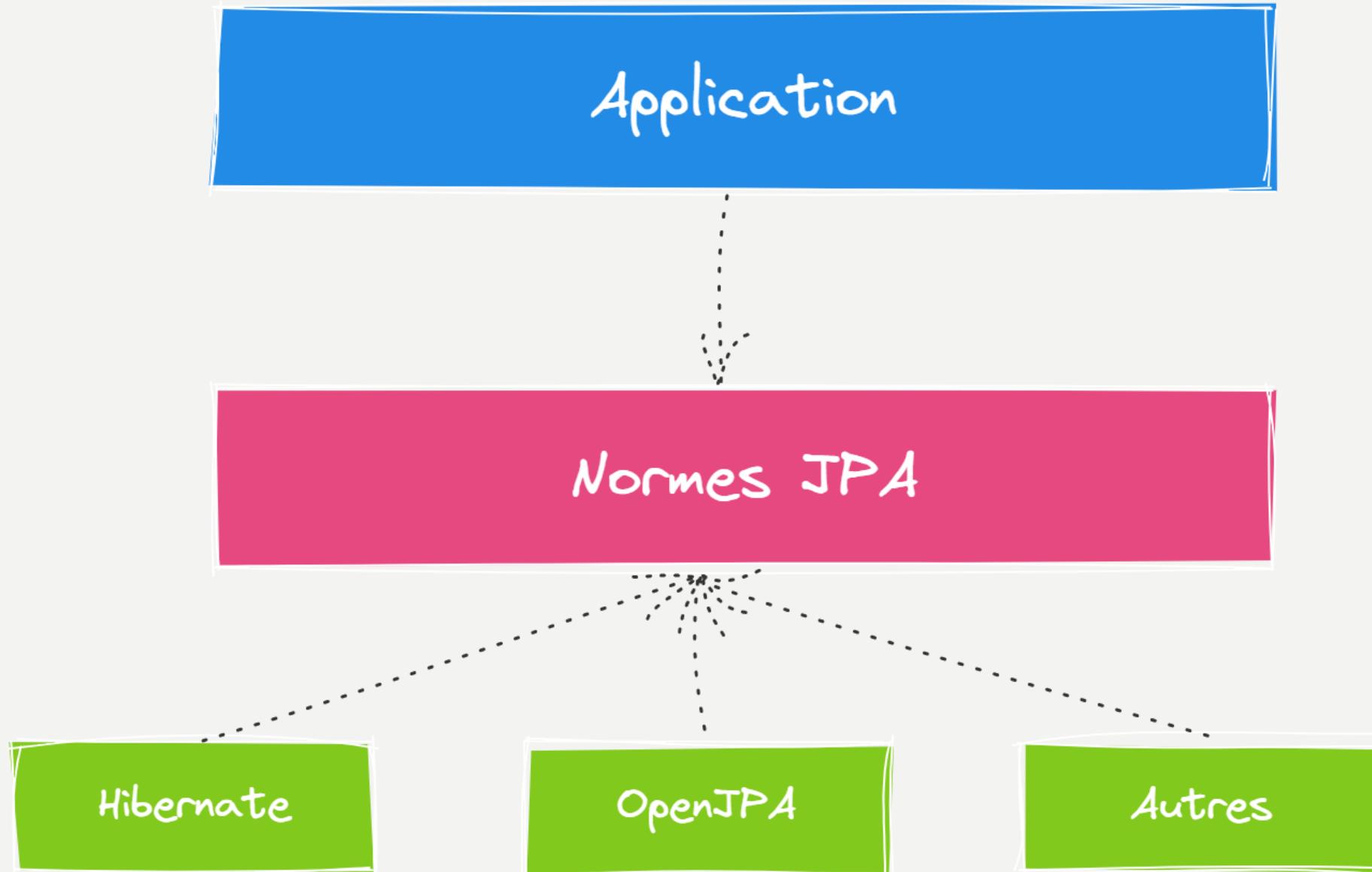
- Java Persistence API
  - Repose sur le principe **POJO**
  - S'affranchir de la gestion des données **SGBDR** (repose sur un **ORM**)
- EntityManager
- Langage **JPQL** (ou **JPA-QL**)
- Package `jakarta.persistence` (anciennement `javax.persistence`)

# INTRODUCTION

- Plusieurs implémentations possibles
  - **Hibernate**
  - **OpenJPA**
  - **Toplink**
  - **DataNucleus**
  - **EclipseLink**
  - ...



# INTRODUCTION



# INTRODUCTION

- Problématique
  - Modèle objet != Modèle relationnel

Modèle objet	Modèle relationnel
Graphe d'objets	Base de données relationnelle
Instances de classes	Enregistrements dans une table
Références	Relations (FK → PK)
« Clé primaire » optionnelle	Clé primaire obligatoire
Héritage	Pas d'héritage

# INTRODUCTION

- Exemple de mapping

Modèle objet (une classe)	Modèle relationnel (une table)
Produit.java	produit
int id	<u>pro_id</u>
String libelle	pro_libelle
Double prix	pro_prix

# INTRODUCTION

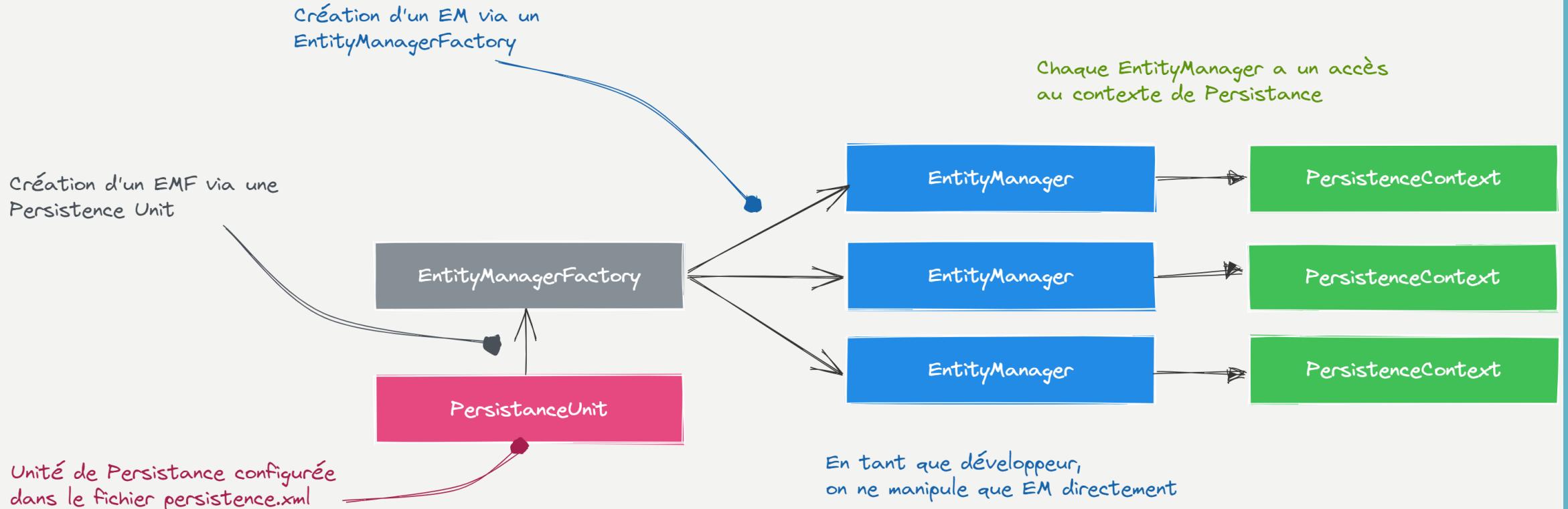
- Masquer la « plomberie » relationnelle
- Les connexions à la base de données ne sont pas visibles
- Plus d'utilisation du **SQL**
- Mapping par annotations



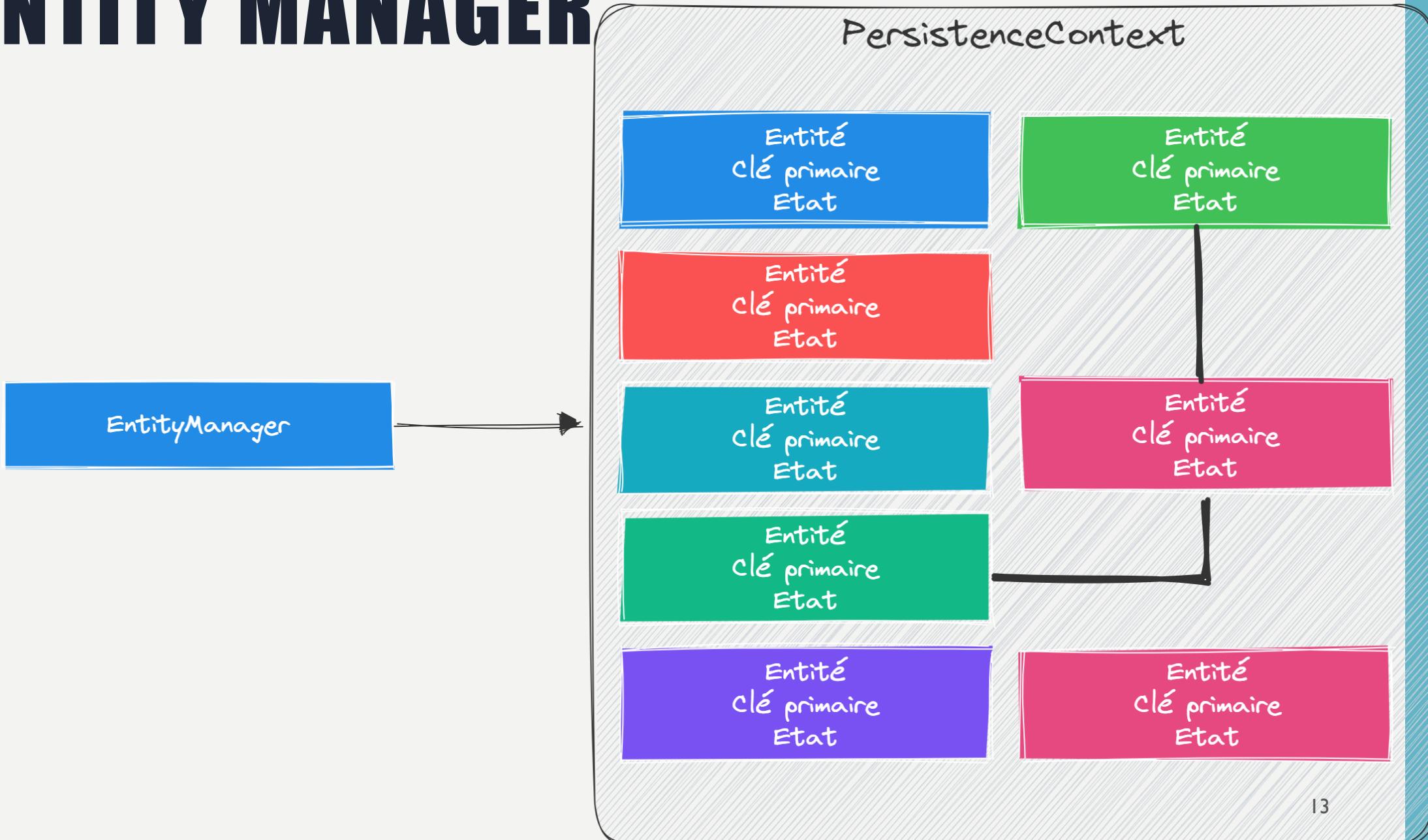
# **MANAGER**

ENTITY MANAGER

# ENTITY MANAGER



# ENTITY MANAGER



# ENTITY MANAGER

- Les méthodes de mise à jour de EntityManager (`persist`,`merge`,`remove`)
  - C'est en réalité fait par le `PersistenceContext`
    - Qui choisi le moment le plus opportun pour effectuer les changements en base de données
  - Mais c'est bien l'EntityManager qui commande ces mises à jour
  - Et nous n'avons accès qu'à l'EntityManager

# ENTITY MANAGER

- Pour créer un EntityManager
  - Il faut un EntityManagerFactory
- Pour créer un EntityManagerFactory
  - Il faut utiliser un PersistenceUnit
- Pour créer un PersistenceUnit
  - Il faut la configurer dans le fichier `src/main/resources/META-INF/persistence.xml`

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

# EXERCICE

- Intégrer la dépendance Hibernate Core
  - hibernate-core version 6.2.1.Final
- Créer la classe d'application principale
  - Avec la création d'un EntityManagerFactory et d'un EntityManager
  - Choisir un nom d'unité de persistance « EShopUnit »
- Exécuter l'application **JAVA** : elle ne fonctionne pas



# CONFIGURATION

## CONFIGURER L'UNITÉ DE PERSISTANCE

# PERSISTENCE UNIT

- Unité de persistance
  - Fichier `src/main/resources/META-INF/persistence.xml`
  - On lui précise le **DataSource**, le **provider** et des **options**
  - On lui précise le type de transaction (JTA / RESOURCE\_LOCAL)
    - Dans notre cas, ce sera toujours RESOURCE\_LOCAL
  - **Attention, chaque implémentation se configure différemment !**
    - L'exemple ci-après est une configuration **Hibernate**

# PERSISTENCE UNIT

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    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">
    <persistence-unit name="NomPersistenceUnit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <class>fr.formation.model.Personne</class>
        <class>fr.formation.model.Fournisseur</class>
        <class>fr.formation.model.Client</class>
        <class>fr.formation.model.Produit</class>

        <properties>
            <!-- Listes des propriétés liées à l'implémentation (Hibernate, OpenJPA, ...) -->
        </properties>
    </persistence-unit>
</persistence>
```

# PERSISTENCE UNIT

```
<persistence-unit name="NomPersistenceUnit" transaction-type="RESOURCE_LOCAL">
    <properties>
        <property name="hibernate.connection.url" value="jdbc:postgresql://localhost:5432/eshop" />
        <property name="hibernate.connection.driver" value="org.postgresql.Driver" />
        <property name="hibernate.connection.user" value="postgres" />
        <property name="hibernate.connection.password" value="root" />

        <!-- Permet d'exécuter les requêtes DDL pour la génération de la base de données -->
        <!-- Valeurs possibles : validate, update, create, create-drop -->
        <property name="hibernate.hbm2ddl.auto" value="update" />

        <!-- On imprime les requêtes SQL générées par Hibernate dans la console -->
        <property name="hibernate.show_sql" value="true" />
        <property name="hibernate.format_sql" value="false" />
    </properties>
</persistence-unit>
```

# PERSISTENCE UNIT

- Type de transaction
  - RESOURCE\_LOCAL
    - Utilisation d'un EntityManagerFactory
      - (possible d'injecter un EntityManagerFactory via @PersistenceUnit)
    - Création manuelle des EntityManager (veiller à n'en avoir qu'un seul actif à la fois)
    - Gestion manuelle des transactions
  - JTA
    - Non-utilisation d'un EntityManagerFactory
    - Injection d'un EntityManager via @PersistenceContext
    - Transaction gérée par le conteneur (**EJB** par exemple)

# EXERCICE

- Créer le fichier de persistance
- Relancer l'application principale : elle doit fonctionner



# **MAPPING**

**MAPPING PAR ANNOTATIONS**

# ENTITÉ

- `@Entity`
  - Classe **persistée** dans une table
    - Chaque instance correspond à un enregistrement
    - Les attributs correspondent aux colonnes
  - Les relations sont exprimées avec des annotations

```
@Entity
public class Produit {
    //...
}
```

# ENTITÉ

- `@Table`
  - Spécifie le mapping à la base de données
  - Quelques options
    - `name` Nom de la table dans la base de données
    - `indexes` Index (hors clé primaire et clés étrangères) dans la table
    - `uniqueConstraints` Contraintes de clé unique

```
@Entity  
@Table(name = "produit")  
public class Produit {  
    //...  
}
```

# CLÉ PRIMAIRE

- `@Id`
  - Indique que l'attribut est utilisé comme la clé primaire
  - **Obligatoire pour chaque entité**
  - Il faut préciser la stratégie de génération des identifiants (`@GeneratedValue`)
    - Par exemple pour une clé auto-incrémentée, la stratégie à utiliser est `IDENTITY`
      - `@GeneratedValue(strategy=GenerationType.IDENTITY)`

# CLÉ PRIMAIRES

- @Id

```
@Entity  
@Table(name = "personne")  
public class Personne {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private int id;  
  
    //...  
}
```

# CLÉ PRIMAIRE

- `@Id` est limité sur un seul champ
  - S'il y a besoin de spécifier une clé primaire sur deux ou plusieurs champs, il faut utiliser, au choix
    - `@IdClass`
    - `@EmbeddedId`

# CLÉ PRIMAIRES

- @IdClass

```
public class AchatId implements Serializable {  
    private int produitId;  
    private int clientId;  
}
```

```
@Entity  
@IdClass(AchatId.class)  
public class Achat {  
    @Id private int produitId;  
    @Id private int clientId;  
}
```

# CLÉ PRIMAIRES

- `@EmbeddedId`

```
@Embeddable
public class AchatId implements Serializable {
    private int produitId;
    private int clientId;
}
```

```
@Entity
public class Achat {
    @EmbeddedId private AchatId id;
}
```

# ATTRIBUTS

- `@Column`
  - Permet d'indiquer le nom de la colonne de l'attribut dans la table
  - Utile si le nom de la colonne est différent du nom de l'attribut de l'objet
  - Utile pour définir la définition de la colonne en base de données
  - Utile pour définir si la colonne est modifiable, nullable, sa taille, ...
- `@Temporal(TemporalType.DATE)`
  - Préciser que la colonne est de type DATE sur les attributs `java.util.Date` ou `java.util.Calendar`
- `@Enumerated(EnumType.ORDINAL)`
  - Préciser la valeur de l'énumérateur
- `@Transient`
  - Ignore cet attribut (par défaut, tout attribut est persisté)

# EXERCICE

- Créer la classe Produit
  - Id, Nom, Prix, Modele, Reference
- Lister tous produits depuis Application

```
List<Produit> myProduits = em.createQuery("select p from Produit p", Produit.class).getResultList();
```

# HÉRITAGE

- `@Inheritance`

- Permet d'indiquer l'héritage entre deux objets (héritage représenté en base de données)
- Il faut préciser la stratégie d'héritage
  - `SINGLE_TABLE`
    - Une seule table pour toutes les classes
    - Utilisation d'un champ discriminant
      - `@DiscriminatorColumn` sur la classe mère
      - `@DiscriminatorValue` sur la classe fille
  - `JOINED`
    - Une table par classe
    - `@PrimaryKeyJoinColumn` sur la classe fille
  - `TABLE_PER_CLASS`
    - Une table par classe contenant toutes les informations (redondance)

# HÉRITAGE

- @Inheritance
  - SINGLE\_TABLE

```
@Entity
@Table(name = "personne")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "per_type")
public class Personne {
    //...
}
```

```
@Entity
@DiscriminatorValue("Client")
public class Client extends Personne {
    //...
}
```

# HÉRITAGE

- @Inheritance
  - JOINED

```
@Entity
@Table(name = "personne")
@Inheritance(strategy=InheritanceType.JOINED)
public class Personne {
    /**
}
}
```

```
@Entity
@Table(name = "client")
@PrimaryKeyJoinColumn(name = "cli_id")
public class Client extends Personne {
    /**
}
}
```

# HÉRITAGE

- @Inheritance
  - TABLE\_PER\_CLASS

```
@MappedSuperclass
public class Personne {
    //...
}
```

```
@Entity
@Table(name = "client")
public class Client extends Personne {
    //...
}
```

# HÉRITAGE

- @Inheritance
  - TABLE\_PER\_CLASS

```
@MappedSuperclass
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    //...
}

@Entity
@Table(name = "client")
@AttributeOverride(name = "id", column=@Column(name = "cli_id"))
public class Client extends Personne {
    //...
}
```

Cas du **@MappedSuperclass**, pour préciser le nom de l'attribut id

# HÉRITAGE

- @Inheritance
  - TABLE\_PER\_CLASS

```
@MappedSuperclass
public class Personne {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String nom;
    //...
}

@Entity
@Table(name = "client")
@AttributeOverride(name = "id", column = @Column(name="cli_id"))
@AttributeOverride(name = "nom", column = @Column(name="cli_nom"))
public class Client extends Personne {
    //...
}
```

Cas du **@MappedSuperclass**, pour préciser le nom de plusieurs attributs

# EXERCICE

- Créer une classe **Personne**
  - Id, Nom
- Créer une classe **Client** qui hérite de **Personne**
  - Prénom, Email
- Créer une classe **Fournisseur** qui hérite de **Personne**
  - Responsable

# RELATIONS

- @OneToOne
  - Relation 1:1
- @OneToMany
  - Relation 1:n
  - List<Object>
- @ManyToOne
  - Relation n:1
  - Object
- @ManyToMany
  - Relation n:n

# RELATIONS

- Chaque relation a son inverse
- @OneToOne / @OneToOne
- @OneToMany / @ManyToOne
- @ManyToOne / @OneToMany
- @ManyToMany / @ManyToMany

# RELATIONS

- `@JoinColumn` (remplace `@Column` dans ce cas)
  - Permet de préciser la colonne de jointure

```
@Entity
@Table(name = "produit")
public class Produit {
    //...

    @ManyToOne
    @JoinColumn(name="PRO_FOURNISSEUR_ID")
    private Fournisseur fournisseur;

    //...
}
```

# RELATIONS

- Sur les relations inverse, il est possible de préciser la source
  - Option mappedBy dans les annotations **@OneToOne**, **@ManyToMany** et **@OneToMany**
  - On précise ici le nom de l'attribut source

```
@Entity  
@Table(name = "produit")  
public class Produit {  
    //...  
  
    @ManyToOne  
    @JoinColumn(name = "pro_fournisseur_id")  
    private Fournisseur fournisseur;  
  
    //...  
}
```

```
@Entity  
@Table(name = "fournisseur")  
public class Fournisseur extends Personne {  
    //...  
  
    @OneToMany(mappedBy = "fournisseur")  
    private List<Produit> produits;  
    //...  
}
```

# RELATIONS

- **@JoinTable**

- Permet de préciser la table et les colonnes de jointure

```
@Entity
@Table(name = "produit")
public class Produit {
    //...

    @ManyToMany
    @JoinTable(
        name = "achat",
        joinColumns = @JoinColumn(name = "ach_produit_id"),
        inverseJoinColumns = @JoinColumn(name = "ach_client_id"))
    private List<Client> clients;

    //...
}
```

# RELATIONS

- **@JoinTable**

- |                        |                                                             |
|------------------------|-------------------------------------------------------------|
| – name                 | Précise le nom de la table de jointure                      |
| – uniqueConstraints    | Précise les colonnes de clé unique                          |
| – joinColumns          | Précise les informations pour l'entité en cours             |
| • name                 | Nom de la colonne clé étrangère de l'entité en cours        |
| • referencedColumnName | Nom de la colonne de référence (table de l'entité en cours) |
| – inverseJoinColumns   | Précise les informations pour l'entité ciblée               |
| • name                 | Nom de la colonne clé étrangère de l'entité ciblée          |
| • referencedColumnName | Nom de la colonne de référence (table de l'entité ciblée)   |

# EXERCICE

- Modifier la classe Produit
  - Ajouter la relation avec le fournisseur
- Modifier la classe Fournisseur
  - Ajouter la liste des produits
- Ajouter les commandes, achats

# RELATIONS

# RELATIONS

- Stratégie de chargement Lazy Loading
  - Chargement à la demande
  - Les données ne sont chargées que si demandées
  - Fonctionne dans un contexte de session
    - Ne fonctionne pas en dehors !
- Stratégie de chargement Eager Loading
  - Chargement brutal
  - Les données sont chargées en même temps que l'objet
  - **Attention, la montée en charge (en mémoire) peut aller très vite, à éviter !**

# VALIDATEURS

- Validateurs **JPA**

- `@NotEmpty`
  - Permet d'indiquer que la colonne ne doit pas être vide
- `@NotNull`
  - Permet d'indiquer que la colonne ne doit pas être nulle
- `@NotBlank`
  - Combinaison de `@NotEmpty` et `@NotNull`
- `@Size`
  - Permet d'indiquer une taille minimum et / ou maximum
- `@Min`
  - Permet d'indiquer une valeur minimum sur un entier
- `@Positive`
  - Permet d'indiquer une valeur positive sur un entier ou un réel
- ...

# VALIDATEURS

- Définition de colonne ou validateurs ?
  - Les validateurs jouent un rôle pendant l'enregistrement d'un objet
    - Ils vont empêcher l'action si le nom est obligatoire par exemple, alors qu'il n'est pas saisi
    - Permet un gain de ressources vers la base de données
  - En aucun cas ils définissent la structure de la colonne en base de données
- Le mieux est encore d'utiliser les deux !

# VALIDATEURS

```
@Entity
@Table(name = "produit")
public class Produit {
    //...

    @Column(name = "pro_nom", columnDefinition = "VARCHAR(50) NOT NULL")
    @NotBlank
    @Size(max = 50)
    private String nom;
}
```

```
@Entity
@Table(name = "produit")
public class Produit {
    //...

    @Column(name = "pro_nom", length = 50, nullable = false)
    @NotBlank
    @Size(max = 50)
    private String nom;
}
```

# EXERCICE

- Ajouter la dépendance *Hibernate Validator*
- Préciser des validateurs



# **MANAGER**

**ENTITY MANAGER**

# ENTITY MANAGER

- Fait le lien entre les données de la base de données et les objets entités
- Opérations essentielles
  - persist
  - merge
  - remove
  - find
  - createQuery
  - ...
- Décide quand et comment récupérer les mises à jour (base de données)
- Gère l'état des instances dont il a la charge
  - Ces instances sont dites **Managed**

# ENTITY MANAGER

- persist

```
public Produit save(Produit produit) {  
    em.persist(produit);  
    return produit;  
}
```

- merge

```
public Produit save(Produit produit) {  
    return em.merge(produit);  
}
```

# ENTITY MANAGER

- find

```
public Produit findById(int id) {  
    return em.find(Produit.class, id);  
}
```

- remove

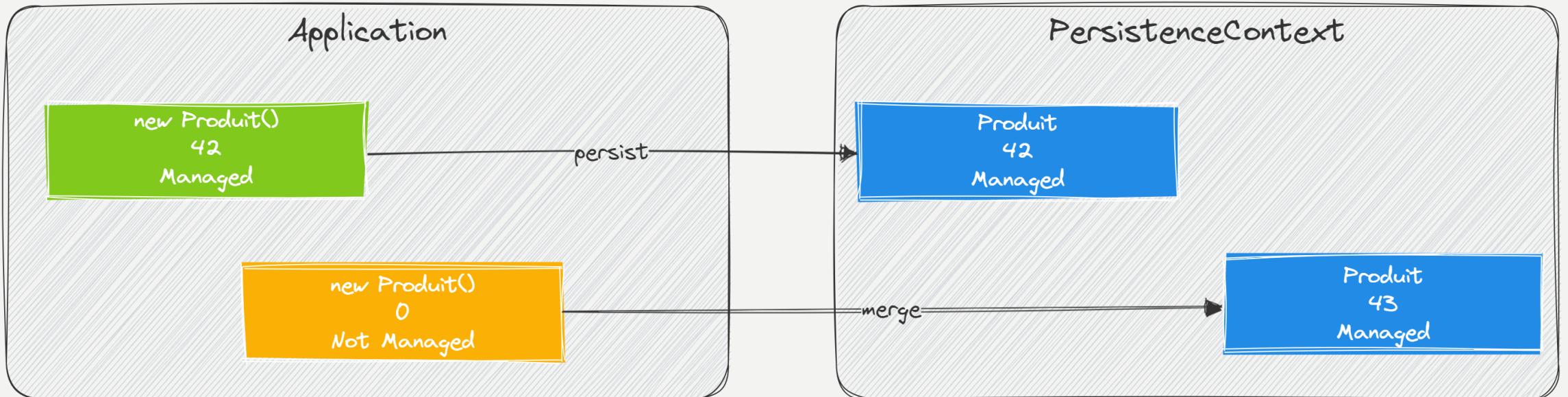
```
public void delete(Produit produit) {  
    em.remove(produit);  
}
```

# ENTITY MANAGER

- L'opération *merge*
  - Crée une copie de l'entité passée en paramètre
  - Il y a donc deux instances différentes de cette même entité
    - Une non-managée par EntityManager (celle passée en paramètre de la méthode *merge*)
    - Une managée par EntityManager (celle retournée par la méthode *merge*)
- Pour cette raison, on utilise *merge* pour supprimer
  - Utiliser *merge* lors de la suppression garanti sa gestion par l'EntityManager
    - Parce qu'il ne peut pas supprimer un objet qu'il ne gère pas

```
public void delete(Produit produit) {  
    em.remove(em.merge(produit));  
}
```

# ENTITY MANAGER

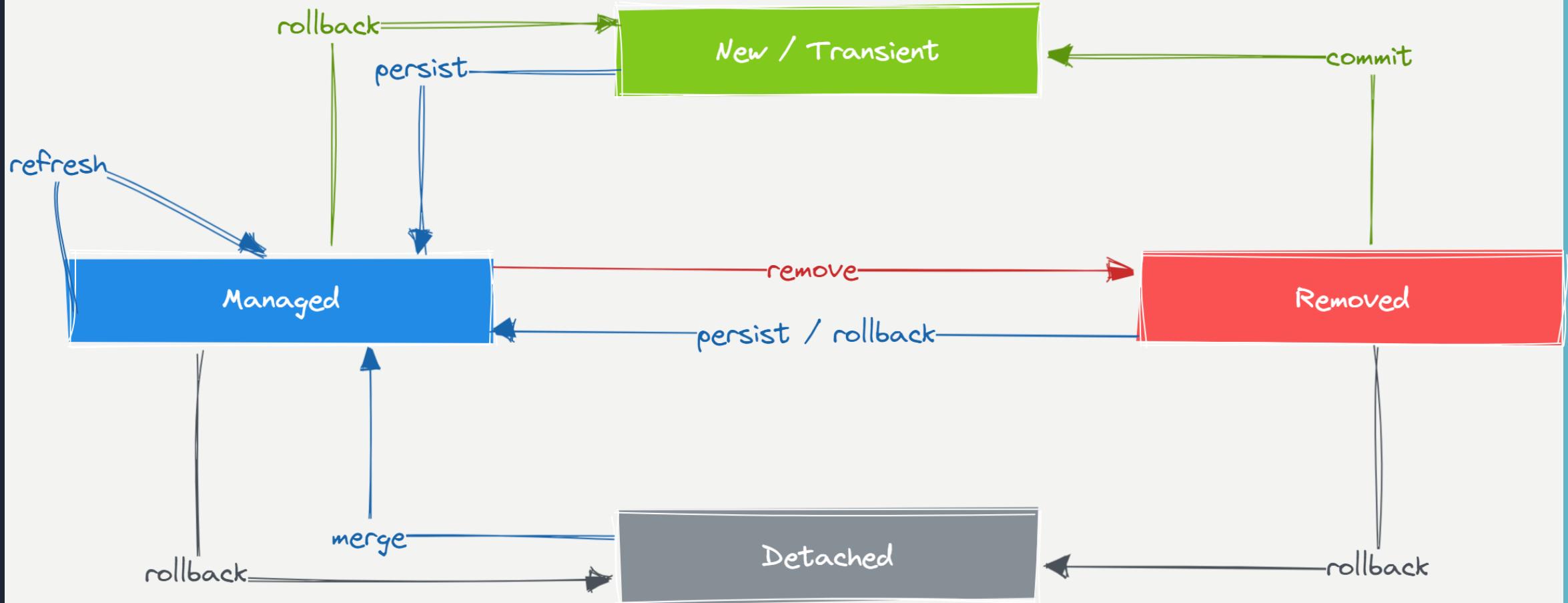


- Avec `persist`, l'instance du nouveau `Produit` est l'instance managée
- Avec `merge`, l'instance du nouveau `Produit` n'est pas l'instance managée
  - Une nouvelle instance a été créée par `EntityManager` !

# ENTITY MANAGER

Etat de l'instance	Comportement
New / Transient	Non géré
Managed	Géré
Removed	Supprimé (suppression logique)
Detached	Détaché (plus géré)

# ENTITY MANAGER



# ENTITY MANAGER

- **Attention**
  - Il faut **gérer la transaction** ! (voir extrait ci-dessous)
  - Pour les requêtes de sélection, pas de soucis puisqu'il n'y a pas de modification en base de données
  - Pour toutes les requêtes qui modifient les données (UPDATE, INSERT, DELETE)
    - il faut penser au commit !

```
EntityTransaction tx = em.getTransaction(); //Récupérer la transaction  
tx.begin(); //Démarrer la transaction  
tx.commit(); //Appliquer les traitements en base de données  
tx.rollback(); //Annuler les traitements
```

# EXERCICE

- Dans Application
  - Créer un nouveau Fournisseur
  - Le sauvegarder en base en utilisant un EntityManager
    - Penser à créer la transaction, puis à la commiter !
  - Lister les fournisseurs : on doit voir le nouveau apparaître
  - Chercher un fournisseur par son ID : afficher son nom
- Utiliser les méthodes persist, createQuery et find



# REQUÊTES

JPQL (JPA-QL)

# JPQL (JPA-QL)

- Langage inspiré du **SQL**
- Pensé avec le paradigme objet

```
public List<Produit> findAll() {  
    Query query = em  
        .createQuery("select p from Produit p", Produit.class);  
  
    return query.getResultList();  
}  
  
public Produit findByLibelle(String libelleProduit) {  
    Query query = em  
        .createQuery("select p from Produit p where p.libelle = ?1", Produit.class);  
  
    //On insère les paramètres  
    query.setParameter(1, libelleProduit);  
  
    return query.getSingleResult();  
}
```

# JPQL (JPA-QL)

- Avec ce paradigme objet, on a la possibilité de naviguer dans les attributs
- Lorsqu'un attribut est une relation ToOne, on peut y accéder directement sans jointure
  - Contrairement au **SQL**
  - C'est **Hibernate** qui fera la jointure

```
em.createQuery("select p from Produit p where p.fournisseur.id = 1", Produit.class);
```

- Lorsqu'un attribut est une relationToMany, il faut une jointure

```
em.createQuery("select f from Fournisseur f left join f.produits p where p.id = 1", Produit.class);
```

# JPQL (JPA-QL)

- Chargement Lazy Loading

```
public Produit findWithAchats(int id) {  
    Query query = em  
        .createQuery("select p from Produit p left join fetch p.achats a where p.id = ?1", Produit.class);  
    //...  
}
```

- En ajoutant le mot-clé **fetch** sur une jointure
  - On demande à **Hibernate** de charger la liste associée

# JPQL (JPA-QL)

- Agrégation

```
public Integer count() {  
    return em.createQuery("select count(*) from Produit", Integer.class).getSingleResult();  
}
```

# REQUÊTES NATIVES

- **JPQL** est le langage recommandé
  - Mais il se peut qu'on ait besoin d'utiliser une spécificité **SQL** ou structurelle
  - Dans ce cas, on peut utiliser les requêtes natives

```
em.createNativeQuery("SELECT * FROM produit WHERE pro_nom = ?1", Produit.class);
```

# EXERCICE

- Liste de requêtes **JPQL**, uniquement la requête (sans la partie **JAVA**)



# REPOSITORY

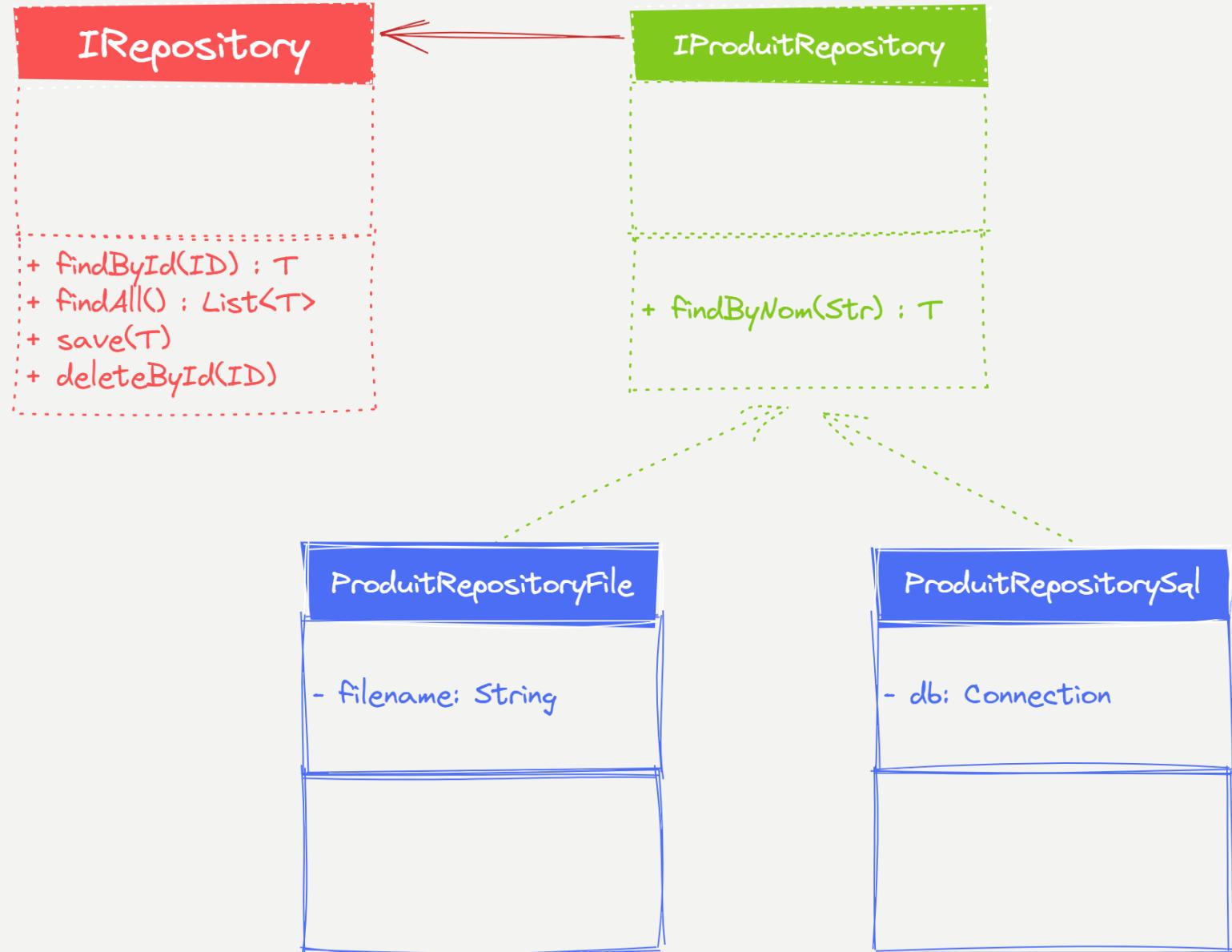
PATTERN REPOSITORY

# PATTERN REPOSITORY

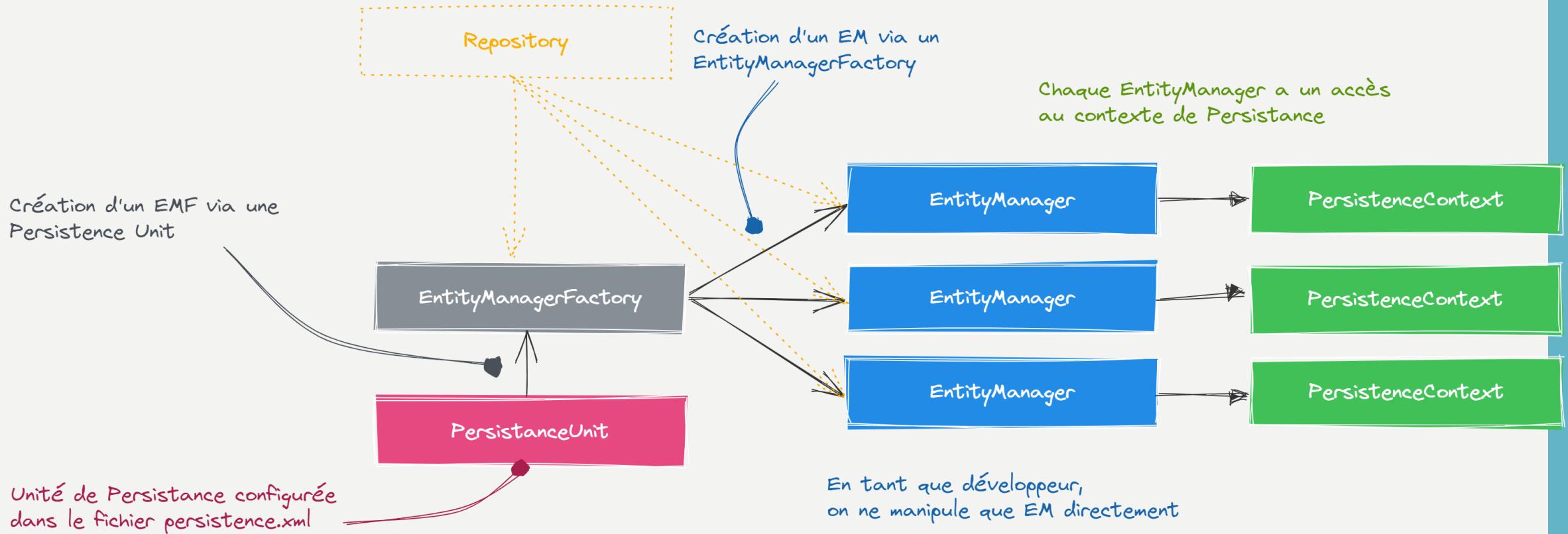
- Chaque **Repository** spécialisée a la responsabilité de traiter les données (**CRUD**)
  - C'est donc lui, et lui seul, qui accède à l'**EntityManager** et qui le manipule

```
public interface IRepository<T, ID> {  
    public List<T> findAll();  
    public T findById(ID id);  
    public T save(T entity);  
    public void deleteById(ID id);  
}
```

# PATTERN REPOSITORY



# PATTERN REPOSITORY



# EXERCICE

- Rendre opérationnel l'exercice de ce cours avec les **Repository**
  - Compléter les **Repository (JPQL)**
- Générer des données
  - Programme ApplicationGenerator
  - Ajouter des produits
  - Ajouter des commandes (relation clients / produits)
- Afficher des données (en console !)
  - Afficher la liste des produits, des clients
  - Afficher la liste des produits pour un client donné (#id client)
  - Afficher la liste des produits pour un fournisseur donné (#id fournisseur)
  - **Ne pas utiliser la stratégie EAGER**