JPA Introduction

adriencaubel.fr

Table des matières

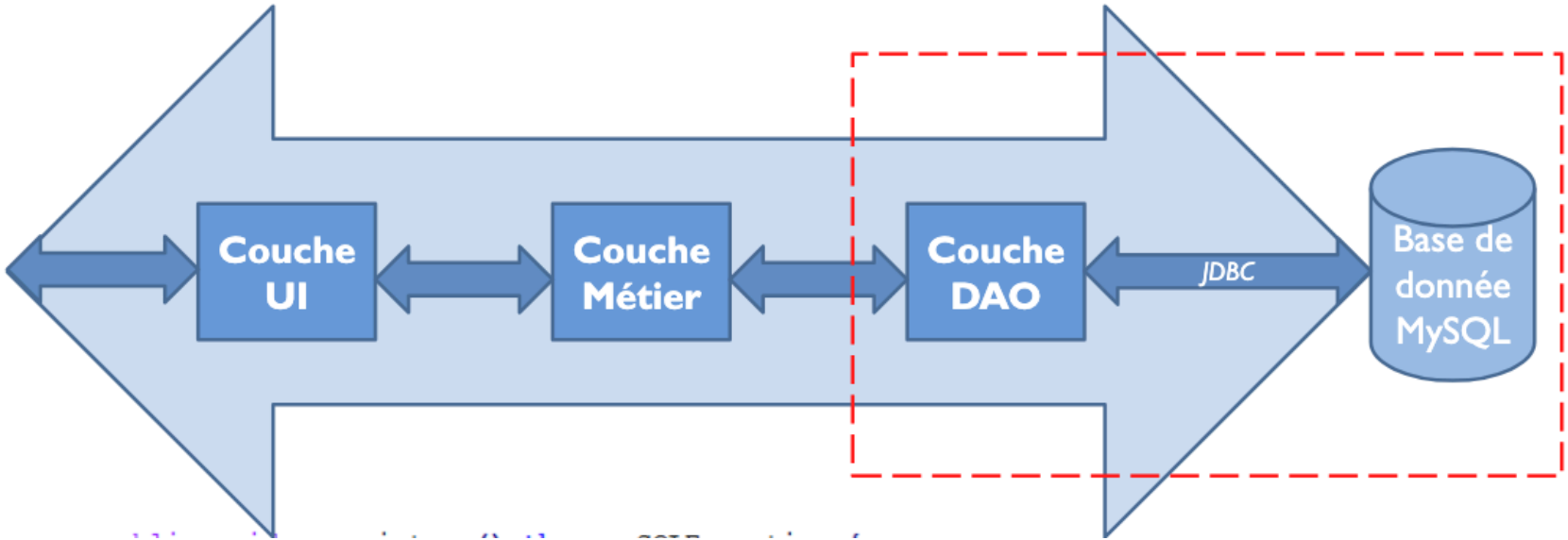
Les limites de JDBC

Limites de JDBC

- Bas niveau => écriture de requête SQL
- Duplication de code
  - ouverture connexion
  - requête
- Connaître le moteur SGBD
  - e.g. `LIMIT` n'est pas du standard SQL mais MySQL

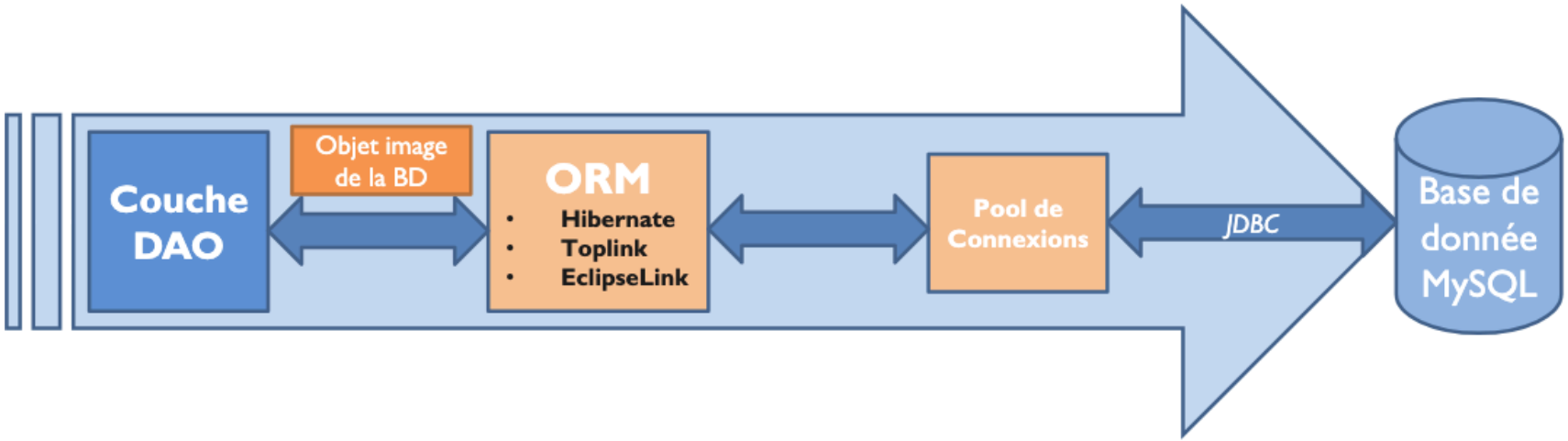
⇒ On souhaiterait une solution pour ne pas avoir à se soucier du SGBD et écrire du code objet

L'accès à la base de données JDBC



1. Obtenir une instance de Connection qui se connecte à la base de données
2. Obtenir une instance de Statement à partir de la connexion
3. Configurer et exécuter une requête SQL via le Statement
4. Exploiter les résultats retournés par la base de données
5. Fermer les différentes instances utilisées

L'accès à la base de données JPA

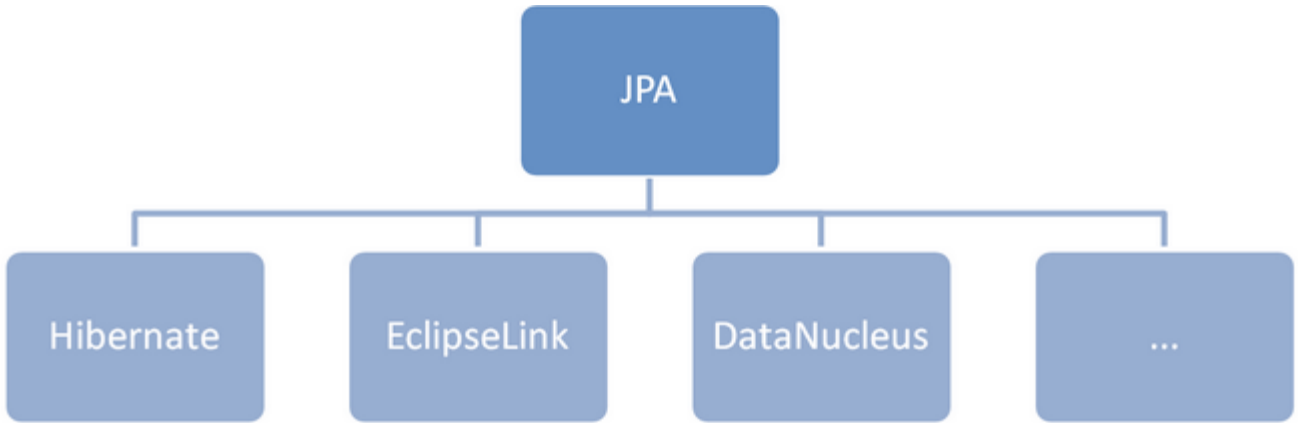



```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MaBaseDeTestPU");
EntityManager em = emf.createEntityManager();

Client client = new Client("Adrien", "CAUBEL")

em.persist(client);
```

L'API JPA



- JPA n'est qu'une API dont l'utilisation nécessite une implémentation
- S'appuie sur JDBC

https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

Le contexte de persistance

Définition

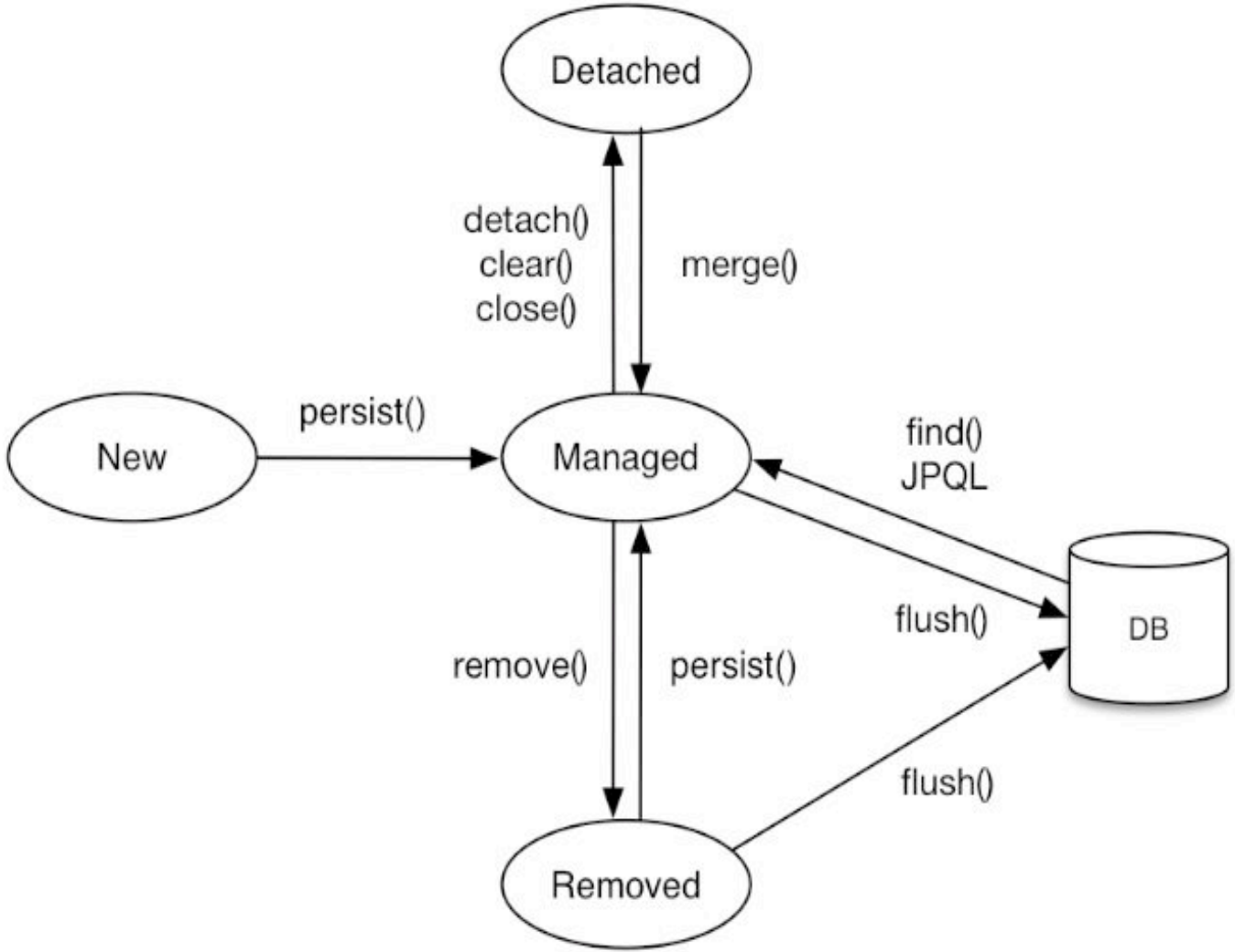
Le contexte de persistance c'est l'ensemble des entités gérées (état `MANAGED`) à un instant donné.

Ces entités, sont gérées par le **EntityManager**

=> Ce contexte est considéré comme un cache de premier niveau

Les opérations prises en charge par le gestionnaire d'entités

Opération	Description
`persist()`	Insère l'état d'une entité dans la base de données. Cette nouvelle entité devient alors une entité gérée.
`remove()`	Supprime l'état de l'entité gérée et ses données correspondantes de 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 » : **fait une COPIE**
`find()`	Exécute une requête simple de recherche de clé.
`createQuery()`	Crée une instance de requête en utilisant le langage JPQL.
`flush()`	Synchronise avec la base de données


Detached
detach()
clear()
close()
merge()
New
persist()
Managed
find()
JPQL
flush()
DB
remove()
persist()
flush()
Removed

Cycle de vie d'une entité : MANAGED

```java
// Etat new
Person person = new Person("John Doe");

// Etat managed (pas persisté en base)
em.persist(person);

// Les modifications apportées à une entité managée sont enregistrées dans le contexte de persistance,
// mais elles ne sont  pas écrites en base tant que la transaction n'est pas validée.
person.setName("John Smith");

// L'entité est insérée dans la base à ce moment.
em.getTransaction().commit();
```

Cycle de vie Entité JPA - DETACH

```
// Etat Managed (récupéré de la BDD)
Person foundPerson = em.find(Person.class, person.getId());

// Etat Detached - plus managée par l'EntityManager
em.detach(foundPerson);

// Seulement une modification in-memory sur l'objet foundPerson
// Le contexte de persistance ne suivra pas ce changement et il ne sera pas enregistré dans la base de données.
foundPerson.setName("John Smith");

// Comme aucun changement sur les entités managées, rien ne sera update en BDD
em.getTransaction().commit();
```

- La modification réalisée sur `foundPerson` ne sera pas persistée car elle est réalisée sur une entité détachée.

Cycle de vie Entité JPA - MERGE

Si on veut persister, nous devons rattacher l'entité en utilisant un `merge()`

```
Person updatedPerson = em.merge(foundPerson); // Reattaches the entity
updatedPerson.setName("John Smith");
em.getTransaction().commit();
```

Attention, ce n'est pas l'objet `foundPerson` qui est attaché, mais une **copie** de cette objet.

Obtenir une fabrique EntityManagerFactory

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

Rappel : `EntityManager` est responsable de gérer les entités JPA

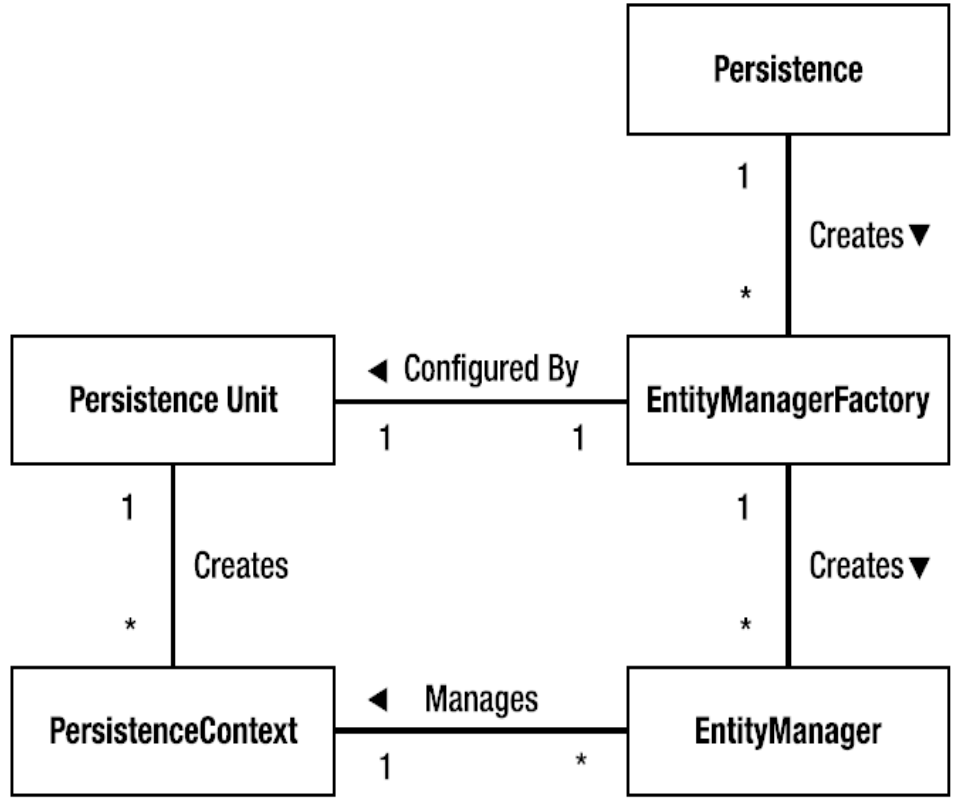


Figure 2-1. Relationships between JPA concepts

1. Le fichier persistence.xml

Le fichier `persistence.xml` contient la configuration de base pour le mapping notamment en fournissant les informations sur la connexion à la base de données à utiliser. Il est stocké dans `resources/META-INF/persistence.xml`

Il contient :

- Des propriétés génériques connues par JPA (url de connexion, user et password)
- Des propriétés spécifiques à l'ORM choisi (ici Hibernate)

1. Le fichier persistence.xml

```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="monapp-unit" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

        <properties>
            <!-- Configuring JDBC properties -->
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/monapp?useSSL=false" />
            <property name="javax.persistence.jdbc.user" value="user" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <!-- Hibernate properties -->
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect" />
            <property name="hibernate.hbm2ddl.auto" value="validate" />

            <!-- Configuring Connection Pool -->
```

2. Créer la fabrique EntityManagerFactory

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("monapp-unit"); // même nom

EntityManager em = emf.createEntityManager();
```

EntityManager

Les interactions entre la base de données et les beans entités sont assurées par un objet de type `javax.persistence.EntityManager`

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("monapp-unit"); // même nom

EntityManager em = emf.createEntityManager();
```

Entity Manager Opérations

```java
// 1. Persist : Enregistrer une nouvelle entité
Person person = new Person("Alice", 25);
em.persist(person);

// 2. Find : Rechercher une entité par sa clé primaire
Person foundPerson = em.find(Person.class, person.getId());

// 3. Merge : Mettre à jour une entité détachée
Person detachedPerson = new Person("Bob", 30);
detachedPerson.setName("Updated Bob");
Person managedPerson = em.merge(detachedPerson);
```

Entity Manager Opérations

```java
// 4. Remove : Supprimer une entité
em.remove(managedPerson);

// 5. Query : Exécuter une requête JPQL
List<Person> persons = em.createQuery("SELECT p FROM Person p", Person.class)
                         .getResultList();
```

Flushing & Dirty Checking

Flushing

Nous avons vu que l'EntityManager joue le rôle de cache de premier niveau, l'ensemble des entités MANAGED sont stockées dans une Map. Ainsi lorsqu'une requête arrive on vérifie en premier si l'entité est présente dans le cache (et on la retourne si oui) avant de faire une requête à la base de données

Problématique

Comment assurer que les données entre le cache et la base de données sont correctement synchronisées ?

Flush Mode (stratégie de vidage)

La stratégie de flush est définie par le paramètre `flushMode` de la session Hibernate en cours d'exécution. Bien que JPA ne définisse que deux stratégies de vidage (`AUTO` et `COMMIT`), Hibernate propose un éventail beaucoup plus large de types de vidage :

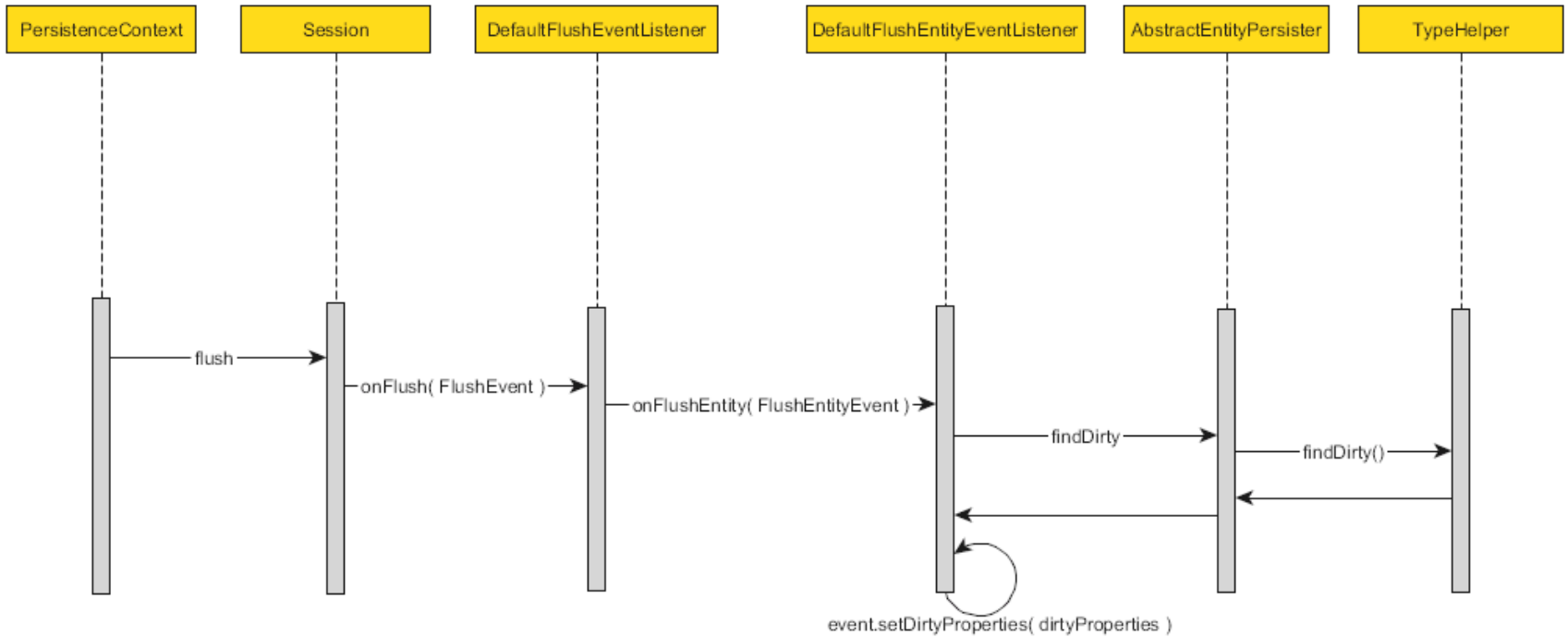
Mode de flush	Description
ALWAYS	Vide (flush) la Session **avant chaque requête**.
AUTO *(par défaut)*	Vide (flush) la Session **uniquement si nécessaire**.
COMMIT	Tente de **repousser le flush jusqu'au commit** de la transaction (mais peut **flush avant** dans certains cas).
MANUAL	Le flush est **géré par l'application** : il faut appeler `Session.flush()` explicitement pour appliquer les changements du contexte de persistance.

Dirty Checking

> Dirty Checking (vérification des modifications) is the mechanism by which Hibernate automatically detects changes made to persistent entities and synchronizes those changes with the database

- Hibernate garde une copie (snapshot) de l'état initial d'une entité quand elle est chargée depuis la base.
- Lorsqu'une propriété est modifiée pendant la session, l'entité devient *dirty* (modifiée).
- Au moment du flush, Hibernate compare l'état actuel avec le snapshot.
- Si différence → Hibernate envoie un UPDATE en base.

Dirty Checking



Transactions

Gestion des transactions

Avec JDBC, nous devons gérer manuellement les transactions

- `void begin()` Débuter la transaction
- `void commit()` Valider la transaction
- `void rollback()` Annuler la transaction
- `boolean isActive()` Déterminer si la transaction est active

=> **Qu'en est-il avec JPA ?**

- deux cas, le premier si on n'a pas de container EE, l'autre si on a un container EE

Cas sans container EE

- En incluant seulement la dépendance hibernate
- Nous n'exécutons pas notre code dans un conteneur

```
@Test
public void testAddLigneDetail() {
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    // entityManager.getTransaction().begin();   NOT WORKING

    Commande commande = new Commande();
    commande.addLigneDetails(new LigneDetail());


    entityManager.persist(commande);

    // entityManager.getTransaction().commit();  NOT WORKING
    entityManager.close();
}
```

Cas avec container EE

- Si on utilise un conteneur JavaEE comme TomEE ou WildFly ou un framework comme Spring
- Alors ma transaction peut être gérée automatiquement (JTA / Spring Tx)
- => préciser que la méthode est transactionnelle (`@Transactional`)

```
public class UserService {
    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void createUser(String name, String email) {
        User user = new User();
        user.setName(name);
        user.setEmail(email);
        entityManager.persist(user); // Transaction automatiquement gérée
    }
}
```