



Ecole Nationale des Sciences Appliquées Al Hoceima

Développement des Applications Java / Java EE

Troisième partie : Gestion de la
persistance

Niveaux : ID2 et GI2

Année universitaire : 2024/2025

Pr. Tarik BOUDAA

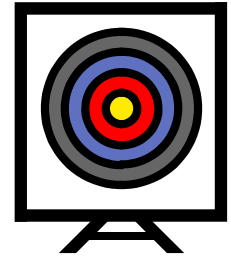


Enseignant :

Pr. Tarik BOUDAA

Email : *t.boudaa@uae.ac.ma*

Objectifs



- Comprendre la problématique de persistance de données
- Comprendre la gestion des transactions et quelques problèmes liés à l'accès concurrent
- Comprendre le principe du mapping objet relationnel
- Comprendre et prendre en main quelques outils Java de gestion de la persistance



JPA : les bases

Introduction

- Souvent les données manipulées par les applications doivent être stockées dans des bases de données relationnelles.
- Dans un système de gestion de base de données relationnelle, les données sont organisées en tables formées de lignes et de colonnes ; elles sont identifiées par des clés primaires et, parfois, par des index. Les relations entre tables utilisent les clés étrangères et joignent les tables en respectant des contraintes d'intégrité, ...
- Ce vocabulaire est totalement étranger à un langage orienté objet comme Java. En Java par exemple, nous manipulons des objets qui sont des instances de classes ; les objets héritent les uns des autres, ...
- Cependant, bien que les objets encapsulent soigneusement leur état et leur comportement, cet état n'est accessible que lorsque la machine virtuelle (JVM) s'exécute : lorsqu'elle s'arrête ou que le ramasse-miettes nettoie la mémoire, tout disparaît.
- Un objet est persistant s'il peut stocker son état afin de pouvoir le réutiliser plus tard

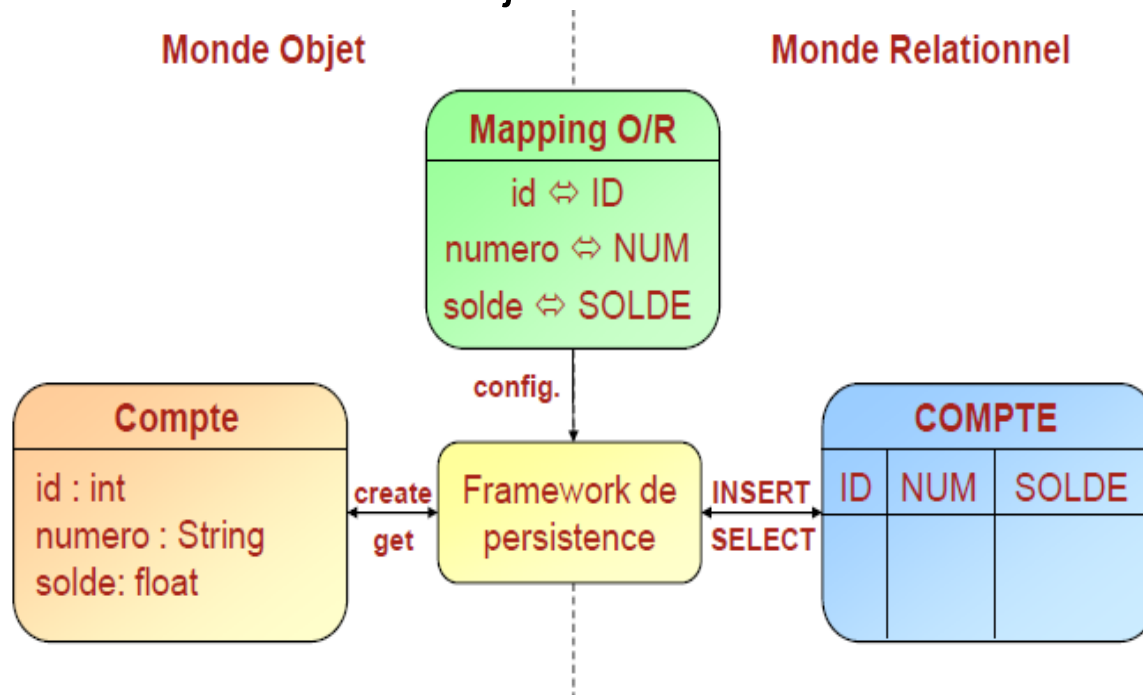


Introduction

- Il existe plusieurs moyen pour persister l'état des objets :
 - ❖ Sérialisation (insuffisante pour les applications complexes)
 - ❖ JDBC (plusieurs inconvénients pour les grands projets)
 - ❖ **ORM (Mapping Objet Relationnel)**

Mapping objet relationnel (ORM)

Le mapping objet-relationnel (en anglais object-relational mapping ou ORM) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé. On pourrait le désigner par « correspondance entre monde objet et monde relationnel »



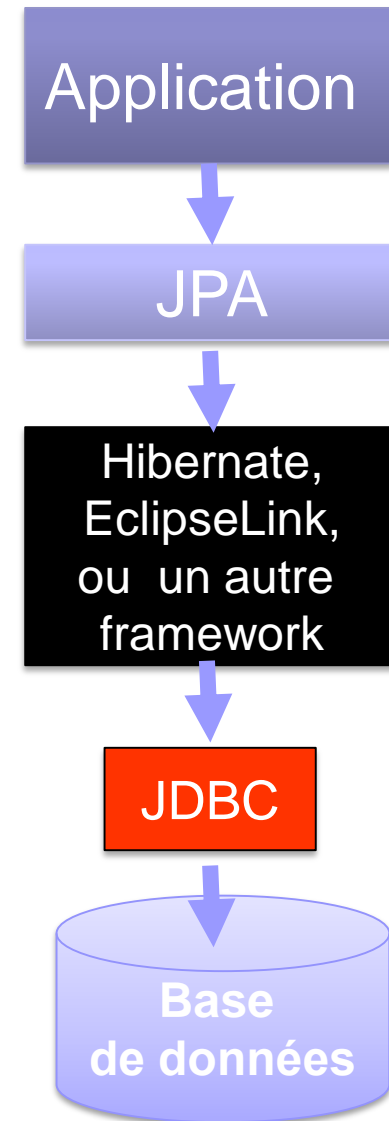
Mapping objet relationnel (ORM)

Un outil de mapping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités
- Proposer une interface qui permette de facilement mettre en œuvre des actions de type CRUD
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée
- Proposer un support des transactions
- Assurer une gestion des accès concurrents (*verrou, deadlock, ...*)
- Fournir des fonctionnalités pour améliorer les performances (*cache, lazy loading, ...*)
- ...

JPA (Java Persistence API)

- La spécification JPA (*Java Persistence API*) a pour objectif d'offrir un modèle d'ORM indépendant d'un produit particulier (comme Hibernate, EclipseLink, etc.). Cette technologie est basée sur:
 - Un ensemble d'interfaces et de classes permettant de séparer l'application et le fournisseur d'un service de persistance (l'ORM).
 - Un ensemble d'annotations pour préciser la mise en correspondance (*mapping*) entre classes Java et tables relationnelles.
 - Un ORM fournisseur de persistance (par exemple Hibernate),
 - Une configuration avec un fichier XML « *persistence.xml* » ou un code Java pour décrire les moyens de la persistance (fournisseur, datasource, etc.)
- La technologie JPA est utilisable dans les applications Web (conteneur Web), ou dans les EJB (serveur d'applications) ou bien dans les applications standards (Java Standard Edition).



JPA : Configuration

Exemple d'une configuration avec un fichier persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<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="com.ensah.gs_etudiants" transaction-type="RESOURCE_LOCAL">
    <description>Hibernate EntityManager Demo</description>
    <class>com.bo.Etudiant</class>
    <exclude-unlisted-classes>true</exclude-unlisted-classes>
    <properties>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MariaDB103Dialect" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.cj.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/dbTestCours" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
    </properties>
  </persistence-unit>
</persistence>
```

JPA : EntityManager et EntityManagerFactory

■ EntityManager

- ✓ L'EntityManager est l'interface principale utilisée pour interagir avec le contexte de persistance dans JPA. Il est responsable de la gestion des entités et de l'exécution des requêtes (insertions, mises à jour, suppressions, recherches, etc.).
- ✓ Les entités gérées par l'EntityManager sont dans le contexte de persistance et toute modification de ces entités est suivie et synchronisée avec la base de données lors du commit de la transaction.
- ✓ Un EntityManager **n'est pas thread-safe**. Il doit être utilisé dans le contexte d'un seul threads

JPA : EntityManager et EntityManagerFactory

■ EntityManagerFactory :

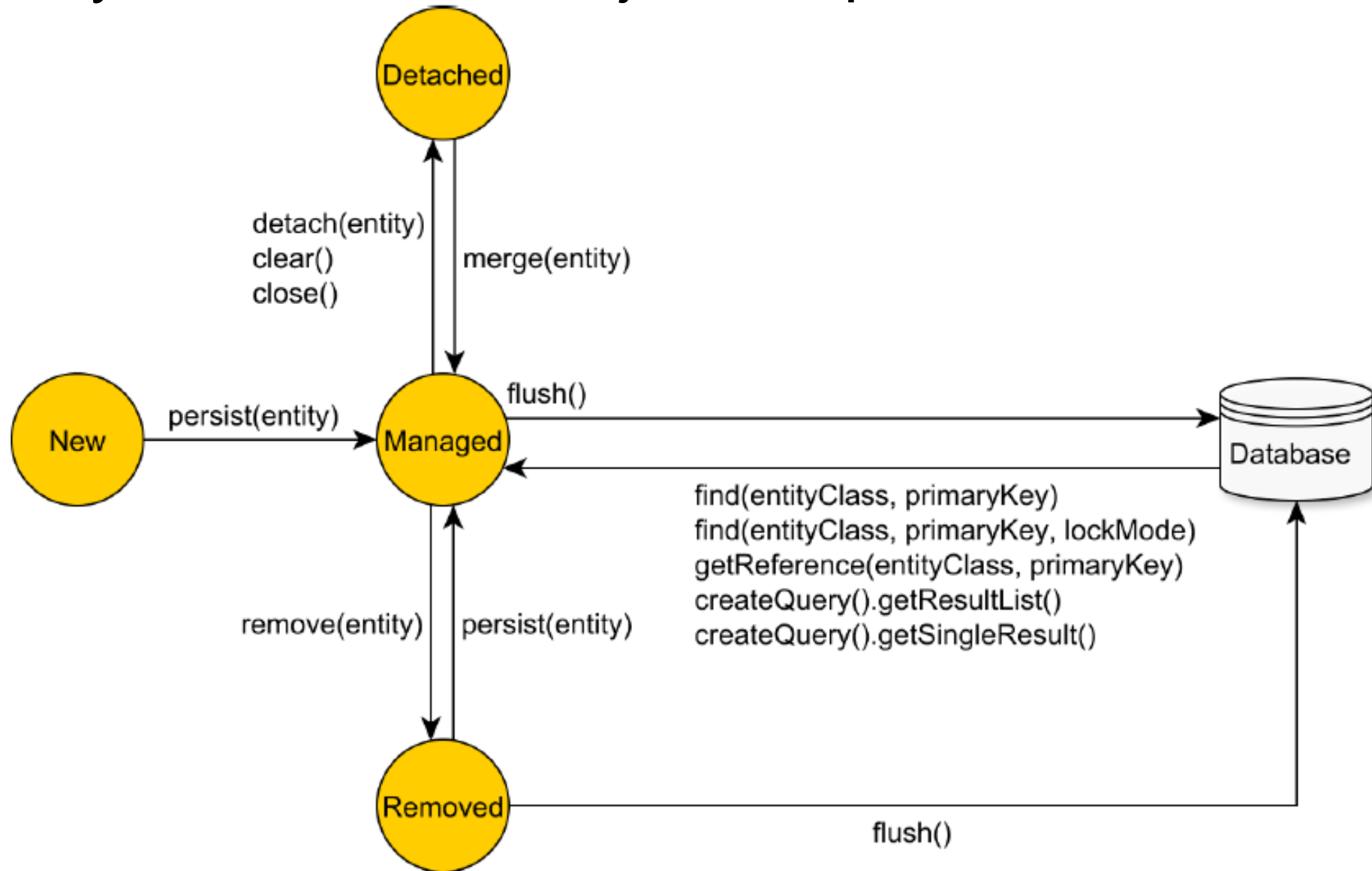
- ✓ L'EntityManagerFactory est un objet qui est responsable de la création d'instances d'EntityManager. Il est un coûteux objet de configuration et doit être créé une seule fois par application pour gérer l'accès à la base de données. Il doit être fermé lors de la fin de l'application.
- ✓ L'EntityManagerFactory **est thread-safe**. Il peut être partagé entre plusieurs threads et utilisé pour créer plusieurs EntityManager en même temps.

Cycle de vie d'un objet manipulé avec JPA

Il y a 4 états possibles pour les objets manipulés par JPA :

État	Description
Nouveau ou Transient <i>(New)</i>	Une entité nouvellement créée qui n'est associée à aucune ligne de base de données est considérée comme étant dans l'état Nouveau ou Transitoire. Une fois qu'elle devient gérée, le contexte de persistance émet une instruction d'insertion lors du flush.
Géré ou Persistant <i>(Managed)</i>	Une entité persistante est associée à une ligne (ou plusieurs) de base de données et est gérée par le contexte de persistance en cours d'exécution. Les modifications d'état sont détectées par le mécanisme de dirty checking et propagées à la base de données sous forme d'instructions de mise à jour lors du flush.
Détaché <i>(Detached)</i>	Une fois que le contexte de persistance en cours est fermé, toutes les entités gérées précédemment deviennent détachées. Les modifications successives ne sont plus suivies et aucune synchronisation automatique avec la base de données ne se produira.
Supprimé <i>(Removed)</i>	Une entité supprimée est simplement programmée pour la suppression, et l'instruction de suppression réelle de la base de données est exécutée lors du flush.

Cycle de vie d'un objet manipulé avec JPA



Méthode de manipulation des entités

- Quelques méthodes de manipulation des entités via EntityManager
 - ✓ persist : Permet de persister l'état d'une entité en base de données
 - ✓ merge : Fusionner l'état de l'entité donnée dans le contexte de persistance actuel.
 - ✓ remove : Supprime une instance d'une entité
 - ✓ find : Rechercher par clé primaire.
 - ✓ close : ferme l'entityManager.
 - ✓ getTransaction : retourne l'objet EntityTransaction permettant la gestion des transactions.

Méthode de manipulation des entités

■ Autres méthodes

- ✓ `flush()` : La méthode `flush()` force l'EntityManager à synchroniser l'état des entités en mémoire avec la base de données. Cela ne valide pas la transaction mais garantit que les modifications sont envoyées à la base.
- ✓ `clear()` : La méthode `clear()` permet de détacher toutes les entités gérées de l'EntityManager. Après cet appel, l'EntityManager ne suivra plus aucune entité. Cela peut être utile pour libérer la mémoire ou lorsque vous ne voulez plus gérer des entités.
- ✓ `detach()` : La méthode `detach()` permet de détacher une entité spécifique de l'EntityManager. Après l'appel de cette méthode, l'entité n'est plus managée.
- ✓ `refresh()` : La méthode `refresh()` permet de recharger une entité à partir de la base de données, en actualisant son état avec les valeurs persistées dans la base.

Méthode de manipulation des entités

■ Exemple

```
EntityManagerFactory emf =  
Persistence.createEntityManagerFactory("com.ensah.gs_etudiants");  
// on obtient une em  
EntityManager em = emf.createEntityManager();  
// On commence une transaction  
EntityTransaction tx = em.getTransaction();  
tx.begin();  
// enregistrer un étudiant dans la base de données  
em.persist(etd);  
// On valide la transaction  
tx.commit();  
//Fermer em  
em.close();
```

■ Pour un exemple complet voir:

<https://drive.google.com/file/d/1AP7k7WhP2KTTZx9JU3ik9QX4Q5MGTVsP/view>

JPQL

Java Persistence Query Language (JPQL) est un langage de requête orienté objet utilisé pour interagir avec les entités d'une base de données dans le cadre de l'API Java Persistence (JPA). Contrairement au SQL traditionnel, JPQL travaille avec des entités Java plutôt qu'avec des tables. Il permet d'effectuer des opérations de sélection, mise à jour, suppression et insertion sur des objets persistants. Les requêtes JPQL utilisent une syntaxe similaire à SQL, mais font référence aux noms d'entités et de leurs attributs, plutôt qu'aux noms de tables et de colonnes. Par exemple, une requête JPQL pour obtenir tous les patients pourrait être écrite comme : **SELECT p FROM Patient p**

JPQL est indépendant de la base de données, ce qui permet de changer la base sous-jacente sans modifier les requêtes.

Annotations de mapping de base

■ Annotation @Entity

Cette annotation marque une classe comme une entité persistante, c'est-à-dire une classe qui est mappée à une table dans la base de données.

@Entity

```
public class Etudiant {  
    ...  
}
```

@Entity(name = "ETUDIANT_TAB")

```
public class Etudiant {  
    ...  
}
```

Annotations de mapping de base

■ Annotation @ID

Cette annotation désigne l'attribut qui sera utilisé comme identifiant unique pour l'entité, correspondant à la clé primaire de la table en base de données.

@Id

```
private Long id;
```

Annotations de mapping de base

■ Annotation @Column

Cette annotation est utilisée pour spécifier les propriétés d'une colonne dans la table de base de données, telles que le nom de la colonne, sa longueur, ou s'il peut être null ou non, unique ou non.

```
@Column(name = "nomCol", nullable = false)
```

```
private String nom;
```

Annotations de mapping de base

■ Annotation @GeneratedValue

Cette annotation est utilisée pour définir la stratégie de génération de la clé primaire (identifiant) de l'entité. Elle est souvent utilisée avec @Id pour que la valeur de la clé primaire soit générée automatiquement.

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

Annotations de mapping de base

■ Exemple

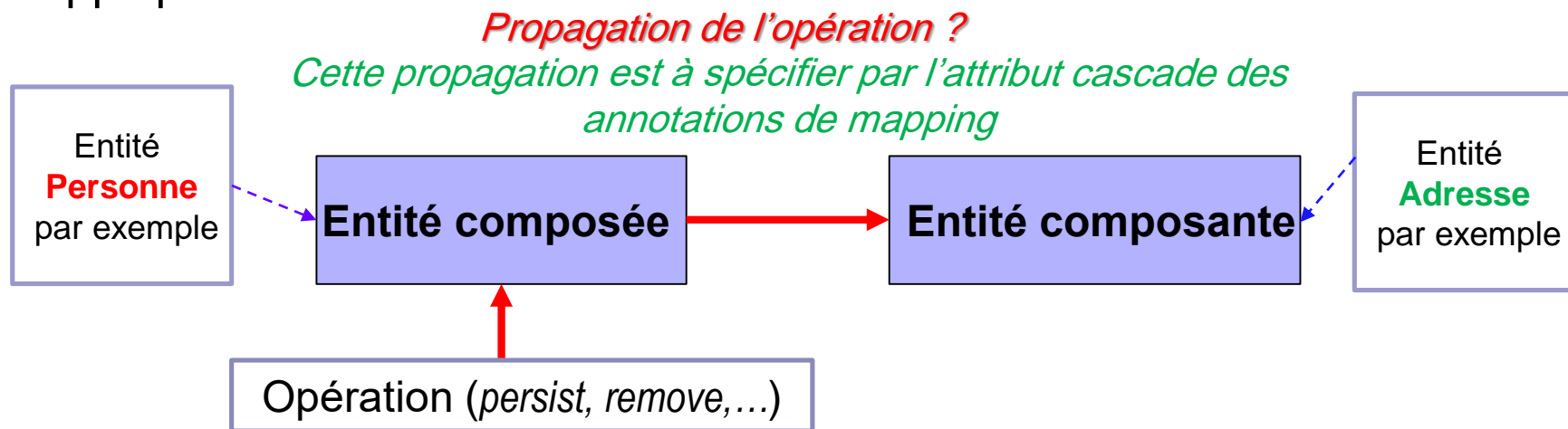
```
@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "nom_etudiant", length = 50, nullable = false)
    private String nom;
    @Column(nullable = false, unique = true)
    private String cin;
    private String prenom;
    // Un objet peut avoir des propriétés que l'on ne souhaite pas rendre
    // persistantes dans la base. Il faut alors impérativement les marquer avec
    // l'annotation @Transient.
    @Transient
    private int valeurCalculée;

    //getters & setter

}
```

Propagation des opérations (Cascade)

- Une entité dépend souvent de l'existence d'une autre entité, par exemple la relation Personne-Adresse. Sans la Personne, l'entité Adresse n'a aucune signification propre. Lorsque nous supprimons l'entité Personne, notre entité Adresse doit également être supprimée.
- La cascade (*Cascading*) est le moyen d'y parvenir. Lorsque nous effectuons une action sur l'entité cible, la même action sera appliquée à l'entité associée.



Propagation des opérations (Cascade)

■ JPA Cascade Type

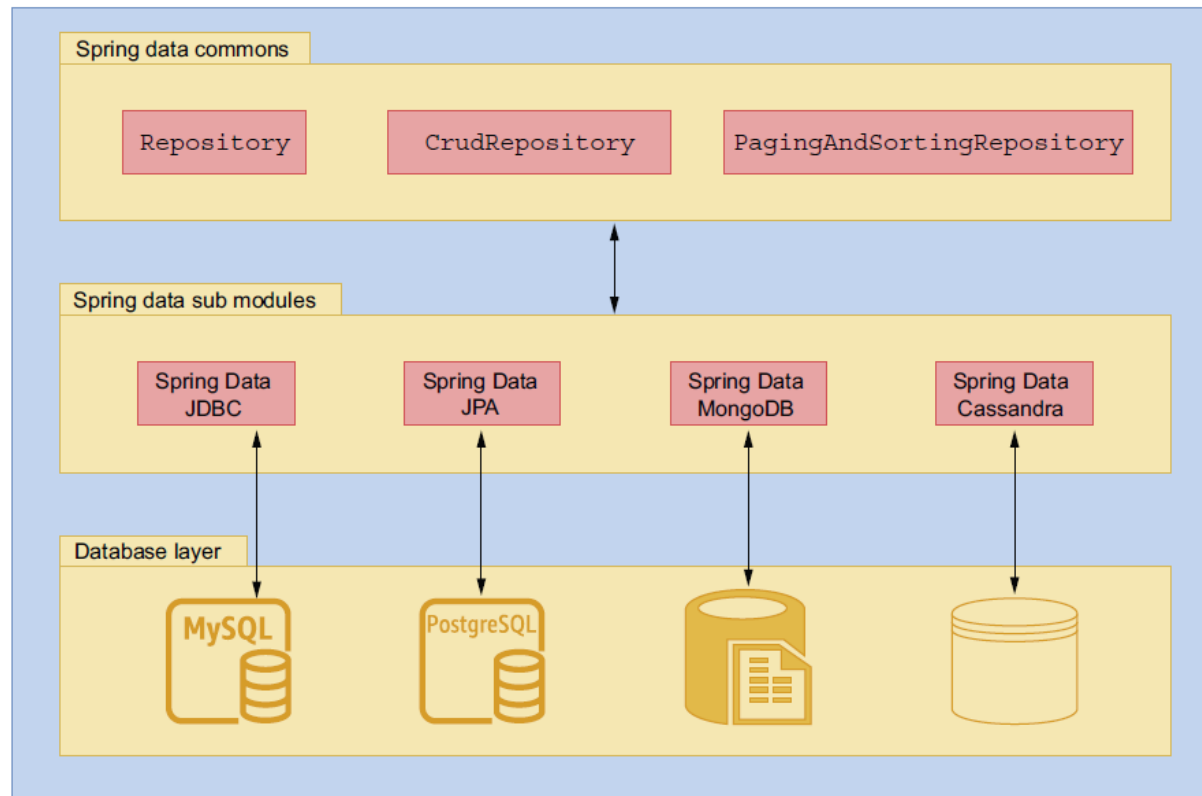
- ✓ **ALL** : Propage toutes les opérations - y compris celles spécifiques à Hibernate - d'un parent à une entité enfant.
- ✓ **PERSIST** : Propage l'opération de persistance d'un parent à une entité enfant. Lorsque nous enregistrons l'entité personne, l'entité adresse sera également enregistrée.
- ✓ **MERGE** : Propage l'opération de fusion d'une entité parent à une entité enfant.
- ✓ **REMOVE** : Propage l'opération de suppression de l'entité parent à l'entité enfant. Il y a également **DELETE**, qui est spécifique à Hibernate et analogue à REMOVE de JPA.
- ✓ **REFRESH** : Les opérations d'actualisation relisent la valeur d'une instance donnée à partir de la base de données. Lorsque nous utilisons cette opération avec Cascade REFRESH, l'entité enfant est également rechargée à partir de la base de données chaque fois que l'entité parent est actualisée.
- ✓ **DETACH** : Lorsque nous utilisons DETACH, l'entité enfant sera également supprimée du contexte de persistance.



Spring Data JPA

Spring Data

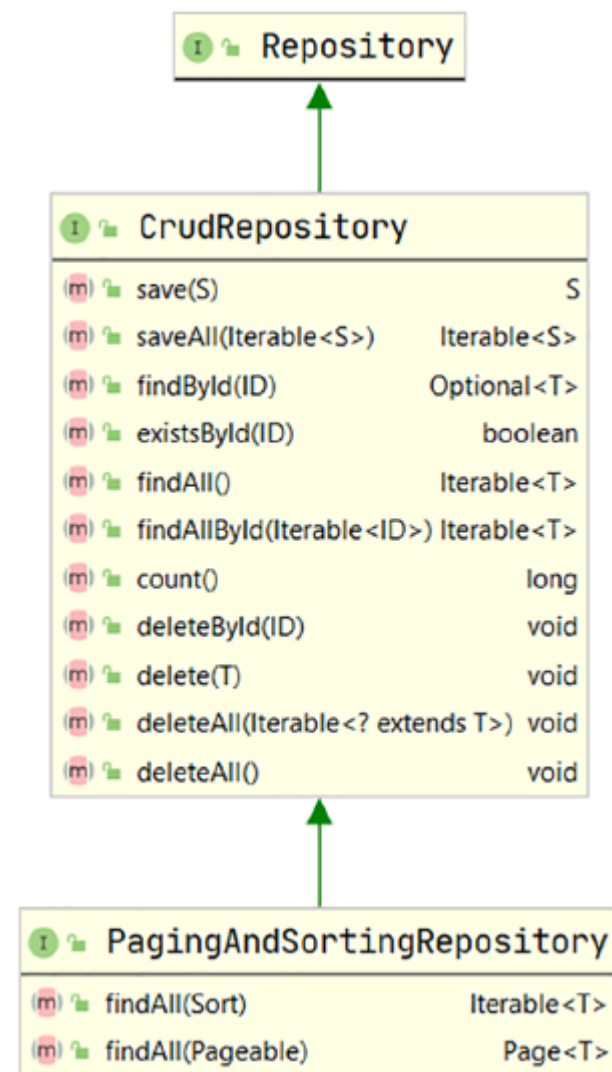
Spring Data est un projet de Spring qui facilite l'accès aux bases de données en simplifiant l'écriture du code de manipulation des données. Il fournit une abstraction au-dessus de divers systèmes de gestion de bases de données (SQL et NoSQL), permettant d'interagir avec eux sans écrire beaucoup de code *boilerplate*.



Spring Data

■ Spring Data Commons

Le module Spring Data Commons fournit une base sur laquelle reposent les autres sous-modules. Chaque sous-module est destiné à un type spécifique de base de données. Les interfaces Repository, CrudRepository et PagingAndSortingRepository font partie du module Spring Data Commons.





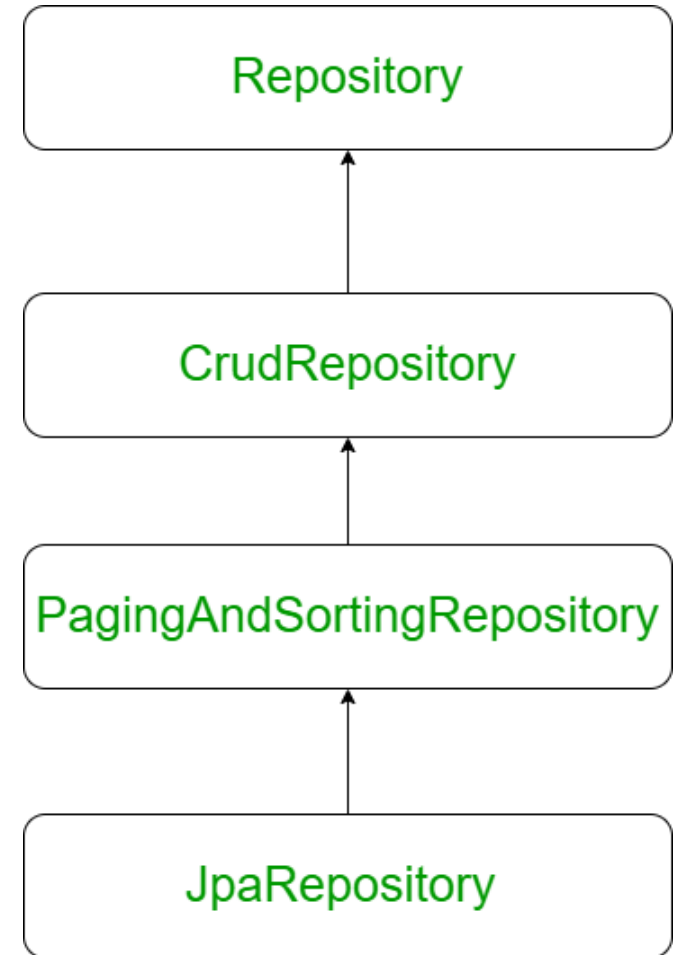
Spring Data JPA

Spring Data JPA est un module de Spring Data qui simplifie l'accès aux bases de données en utilisant JPA (Java Persistence API). L'objectif principal de Spring Data JPA est de réduire le code nécessaire pour interagir avec une base de données relationnelle tout en s'intégrant facilement avec le framework Spring.

Spring Data JPA

■ L'interface JpaRepository

L'interface `JpaRepository` est une extension de l'interface `PagingAndSortingRepository` et fait partie de Spring Data JPA. Elle fournit des méthodes prédéfinies pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur des entités JPA, ainsi que des fonctionnalités de pagination et de tri. Elle permet aux développeurs de manipuler des données dans une base de données de manière abstraite, sans avoir à écrire des requêtes SQL.



Spring Data JPA

■ Dépendance de Spring Data JPA

En plus des dépendances essentielles du Framework Spring il faut ajouter la dépendance spring data jpa:

```
<dependency>  
  <groupId>org.springframework.data</groupId>  
  <artifactId>spring-data-jpa</artifactId>  
  <version>x.y.z</version>  
</dependency>
```

Spring Data JPA

- **Ecriture d'un DAO avec JpaRepository**
- L'écriture d'un DAO offrant les opérations CRUD avec Spring data jpa est réduite à l'écriture d'une interface qui hérite de l'interface **JpaRepository** de Spring data jpa:

```
public interface ExemplePersonneRepository01  
extends JpaRepository<Personne, Long> {  
}
```


Spring Data JPA

■ Approche par convention de nommage des méthodes

Spring DATA JPA utilise la convention de nommage des méthodes qui permet de générer automatiquement les requêtes correspondantes sans avoir besoin d'écrire du SQL ou du JPQL.

```
public interface PersonneRepository extends JpaRepository<Personne, Long> {  
    Personne getPersonneByCin(String cin);  
    List<Personne> getPersonneByNomAndPrenom(String nom, String prenom);  
    List<Personne> findAllByAge(int age);  
    List<Personne> findAllByNomOrderByAge(String nom);  
    boolean existsByNom(String nom);  
    long countByAge(int age);  
    List<Personne> findByNomOrPrenom(String nom, String prenom);  
    List<Personne> findByNomStartsWith(String nom);  
}
```

Pour plus d'informations sur ces conventions voir:

<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>

Spring Data JPA

■ Utilisation de Named Queries

Spring Data JPA permet d'utiliser les requêtes nommées (Named Queries), qui sont une fonctionnalité native de JPA.

```
@Entity
@NamedQuery(name = "Personne.rechercherParNomEtPrenom", query =
"select p from Personne p where p.nom=?1 and p.prenom=?2")
public class Personne {
}
```

```
public interface PersonneRepository extends JpaRepository<Personne, Long> {

    List<Personne> rechercherParNomEtPrenom(String nom, String prenom);

}
```

Spring Data JPA

■ Utilisation de l'annotation @Query

Spring Data JPA permet d'écrire des méthodes avec des requêtes personnalisées en utilisant l'annotation @Query. Cette annotation peut être utilisée pour spécifier des requêtes en JPQL ou en SQL natif

```
public interface PersonneRepository extends JpaRepository<Personne, Long> {  
    @Query("select p from Personne p where p.age=?1")  
    List<Personne> findAllByAge(int age);  
    @Query("select p from Personne p where p.prenom=:prenom and p.age >:age")  
    List<Personne> findGreaterThan(@Param("prenom") String prenom, @Param("age") int age);  
    @Query(value = "select * from Personne where age=?1", nativeQuery = true)  
    List<Personne> findAllByAgeNative(int age);  
    @Modifying  
    @Transactional  
    @Query("update Personne p set p.age=:age where p.nom=:nom")  
    int updatePersonneAgeByNom(@Param("age") int age, @Param("nom") String nom);  
}
```

Spring Data JPA

■ Ecriture d'un Repository personnalisé

Pour enrichir un repository avec des fonctionnalités personnalisées, il faut d'abord définir une interface par exemple *IPersonDaoCustom* et une implémentation pour ces fonctionnalités personnalisées, comme suit :

```
package com.ensah.core.dao;
import java.util.List;
import com.ensah.core.bo.Person;
public interface IPersonDaoCustom {
    List<Person> getPersonsByFirstName(String firstName);
    //d'autres méthodes
}
```

```
package com.ensah.core.dao;
import java.util.List;
public class PersonDaoImpl implements IPersonDaoCustom {
    @Autowired
    private EntityManager entityManager;
    @Override
    public List<Person> getPersonsByFirstName(String firstName) {
        Query query = entityManager.createNativeQuery(
            "SELECT em.* FROM Person WHERE firstname LIKE ?", Person.class);
        query.setParameter(1, firstName + "%");
        return query.getResultList();
    }
    //d'autres méthodes
}
```

Spring Data JPA

■ Ecriture d'un Repository personnalisé

Ensuite, l'interface repository doit étendre l'interface *IPersonDaoCustom* et *JpaRepository*, comme suit :

```
package com.ensah.core.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.ensah.core.bo.Person;

public interface IPersonDao extends JpaRepository<Person, Long>, IPersonDaoCustom {

}
```

Spring Data JPA

■ Pagination avec PagingAndSortingRepository

public interface

```
PagingAndSortingRepository<T, ID> extends  
CrudRepository<T, ID> {  
    Page<T> findAll(Pageable pageable);  
    Iterable<T> findAll(Sort sort);  
}
```

Spring Data JPA

■ Exemple de pagination

```
Pageable pageable = PageRequest.of(0,3);  
jpaRepository.findAll(pageable);  
int pageNumber = pageable.getPageNumber();
```

```
pageable = pageable.next();  
jpaRepository.findAll(pageable);  
pageNumber = pageable.getPageNumber();
```

Spring Data JPA

■ Exemple avec tri

```
public List<Personne> getSortedPrsonDescending() {  
    return (List<Personne>) jpaRepository.  
        findAll(Sort.by(Sort.Direction.DESC, "age"));  
}
```


Exemple sur machine

- Pour plus d'exemples voir :

https://drive.google.com/file/d/1MTdx8mNwj9k84GBFLNFI5L6a2FFgP-k/view?usp=drive_link

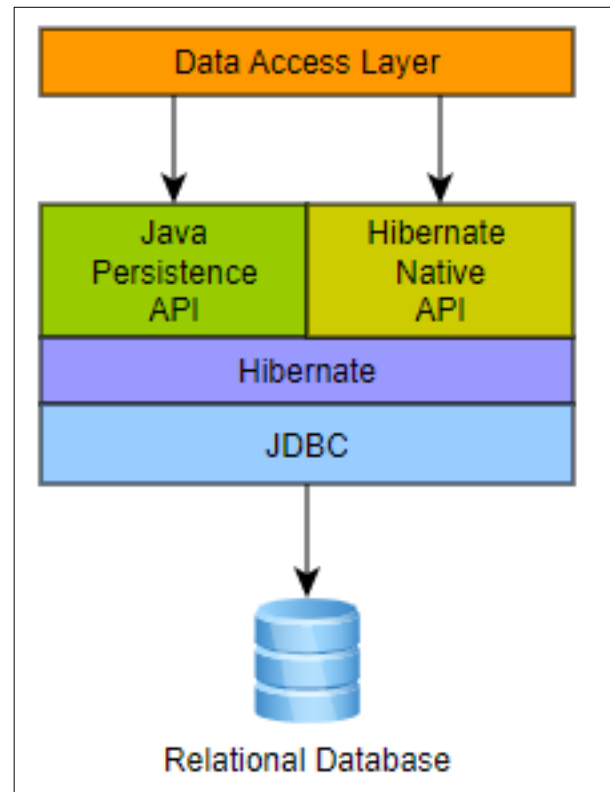




Framework Hibernate

Le framework Hibernate

- Hibernate est un outil de mapping O/R qui permet la persistance transparente pour des objets Java dans des bases de données relationnelles. Il regroupe un ensemble de librairies assurant la tâche de persistance.

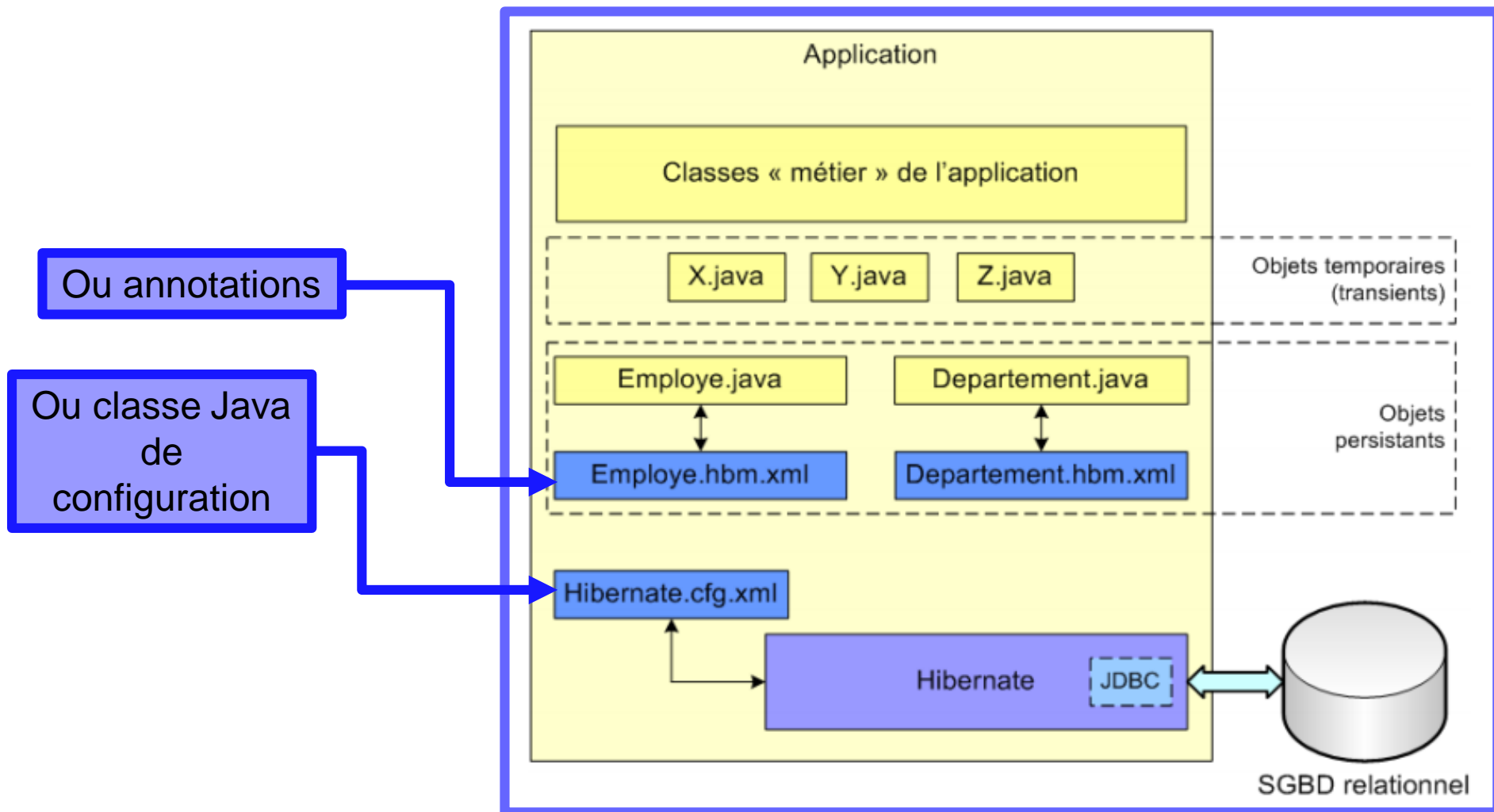




Le framework Hibernate

- Hibernate génère le code SQL
- Avec Hibernate, il n'y a pas d'objet ResultSet à gérer : Cycle de récupération manuelle des ResultSet + Casting de chaque ligne du resultset (type Object) vers un type d'objet métier.
- Persistance transparente : Le développeur peut faire de ses classes métiers des classes persistantes sans ajout de code tiers.
- Les objets métiers sont plus faciles à manipuler.
- Peu de dépendance envers une base de données précise.

Contenu d'une (simple) application Hibernate



Le fichier hibernate.cfg

Parce qu'Hibernate est conçu pour fonctionner dans différents environnements, il existe beaucoup de paramètres de configuration à fixer (*paramètres d'accès à la base de données : login, mot de passe, le dialecte SQL à utiliser, taille du pool de connexion, configuration des transactions, ...*)

Ces configurations s'effectuent dans le fichier ***hibernate.cfg.xml*** (ou ***hibernate.properties***)

Les dernières versions du Framework permettent également de faire **une configuration par une classe Java**.

Le fichier hibernate.cfg

■ Exemple d'un fichier hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database host -->
    <property name="hibernate.connection.url">jdbc:mysql://localhost/tpj2ee2017</property>
    <!-- JDBC Driver for MySQL -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <!-- Database user login -->
    <property name="hibernate.connection.username">root</property>
    <!-- Database user password -->
    <property name="hibernate.connection.password">boudaa</property>
    <!-- SQL Dialect, we will use MYSQL Dialect -->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <!-- we cant to show sql command in console (just for debug purpose) -->
    <property name="hibernate.show_sql">true</property>
    <!-- we generate the database schema once and we update it if necessary -->
    <property name="hibernate.hbm2ddl.auto">update</property>
    <!-- transaction manager -->
    <property name="hibernate.transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <!-- here we specify the mapping classes -->
    <mapping class="com.ensah.Etudiant" />
  </session-factory>
</hibernate-configuration>
```

Hibernate permet également de faire **une configuration par une classe Java.**

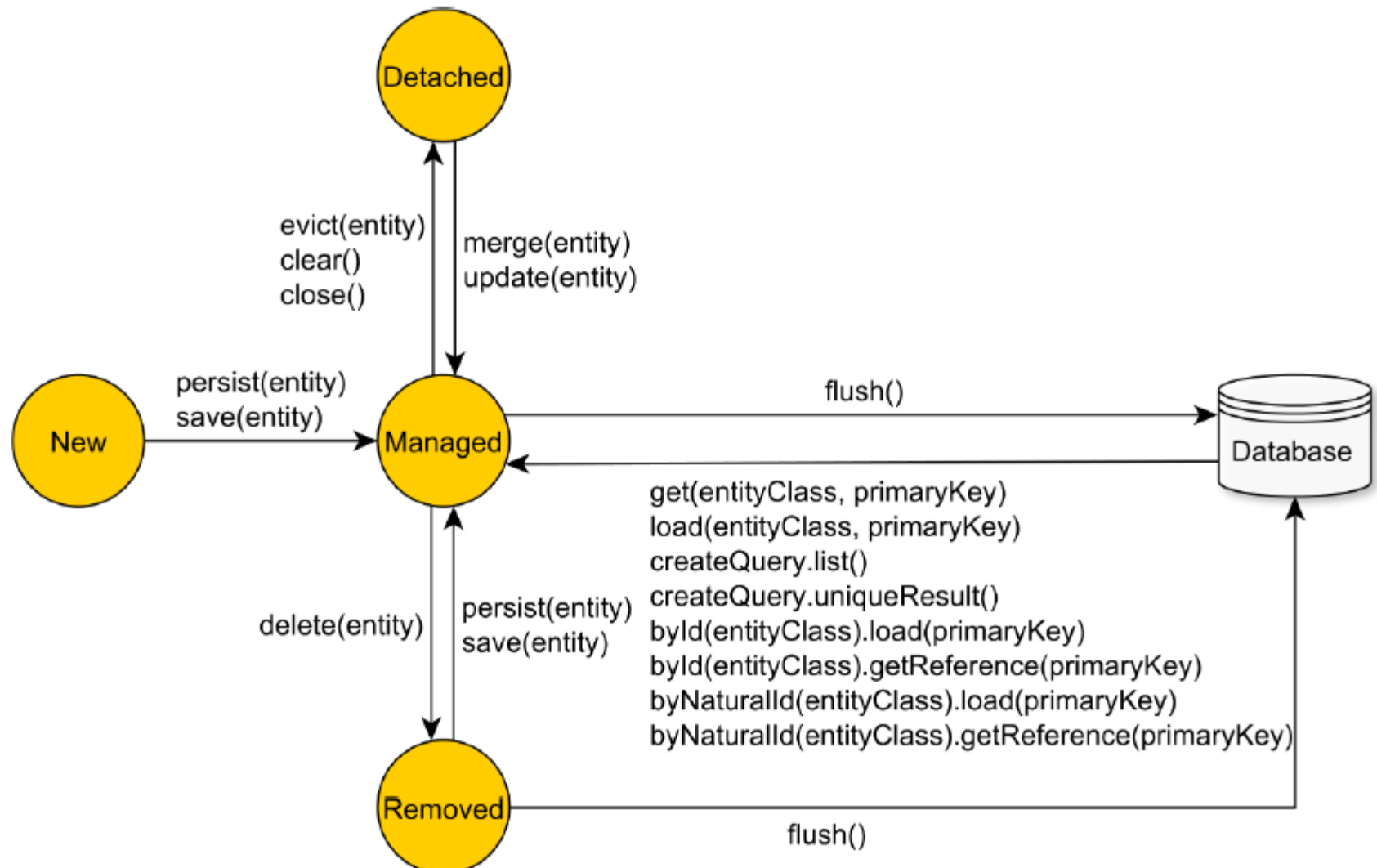
Session Hibernate

- 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. Fabrique des objets Transaction. La Session contient un cache (de premier niveau) des objets persistants, qui sont utilisés lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

SessionFactory

- Elle permet de fabriquer les Sessions. Elle est *threadsafe*.
- Dans application généralement on crée une seule instance de type SessionFactory
- Elle est Client du **ConnectionProvider** (*une fabrique de connexions JDBC*)
- **SessionFactory** peut contenir un cache optionnel de données (**de second niveau**)

Le cycle de vie d'un objet manipulé avec Hibernate et méthodes de la session



Manipulation des données avec Hibernate

■ Méthode d'exécution d'une opération sur la base de données

```
Session session = null;
Transaction tx = null;
try {
    // On récupère l'instance de type SessionFactory
    SessionFactory sf = SessionFactoryBuilder.getSessionFactory();
    // on obtient une session
    session = sf.getCurrentSession();
    // On commence une transaction
    tx = session.beginTransaction();
    // on execute les opérations bases de données (save, delete,...)
    //...
    // On valide la transaction, ceci ferme également la session
    tx.commit();
} catch (HibernateException ex) {
    // Si il y a des problèmes et une transaction a été déjà crée on l'annule
    if (tx != null) {
        // Annulation d'une transaction
        tx.rollback();
    }
    // On n'oublie pas de remonter l'erreur originale
    throw ex;
} finally {
    // Si la session n'est pas encore fermée par commit
    if (session != null && session.isOpen()) {
        session.close();
    }
}
```

Exemple sur machine

Dans les codes sources

<https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xzZtZEC5V6x0s/view>

Ecriture d'un DAO pour la gestion de la persistance des objets de la classe Etudiant (Opérations CRUD) (la classe **HibernateEtudiantDaoImpl**) avec deux types de configuration:

Configuration avec XML:

[Exemple_Cours_Hibernate_01_02](#)

Configuration Java:

[Exemple_Cours_Hibernate_01_01](#)



Hibernate et méthodes de requêtage des données

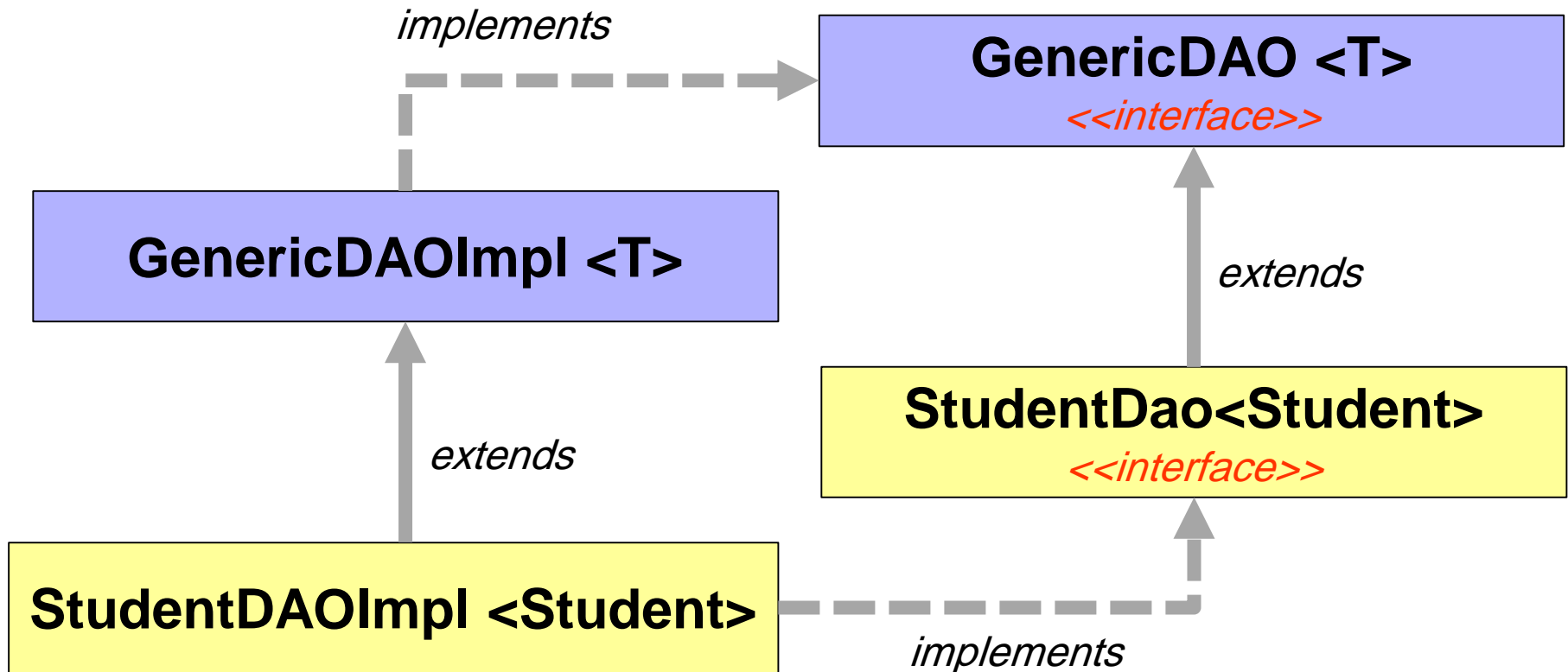
- Hibernate offre plusieurs moyens pour la manipulation et le requêtage des données :
 - ❖ Langage HQL
 - ❖ Langage JPQL (Via API JPA)
 - ❖ API Criteria
 - ❖ Requêtes SQL natives



Le pattern DAO

Le pattern DAO (Data Access Object) est un pattern structurel qui permet d'isoler la couche application/métier de la couche de persistance (généralement une base de données relationnelle, mais il peut s'agir de tout autre mécanisme de persistance) à l'aide d'une API abstraite. La fonctionnalité de cette API est de cacher à l'application toutes les complexités impliquées dans l'exécution des opérations CRUD dans le mécanisme de stockage sous-jacent. Cela permet aux deux couches d'évoluer séparément sans rien savoir l'une de l'autre.

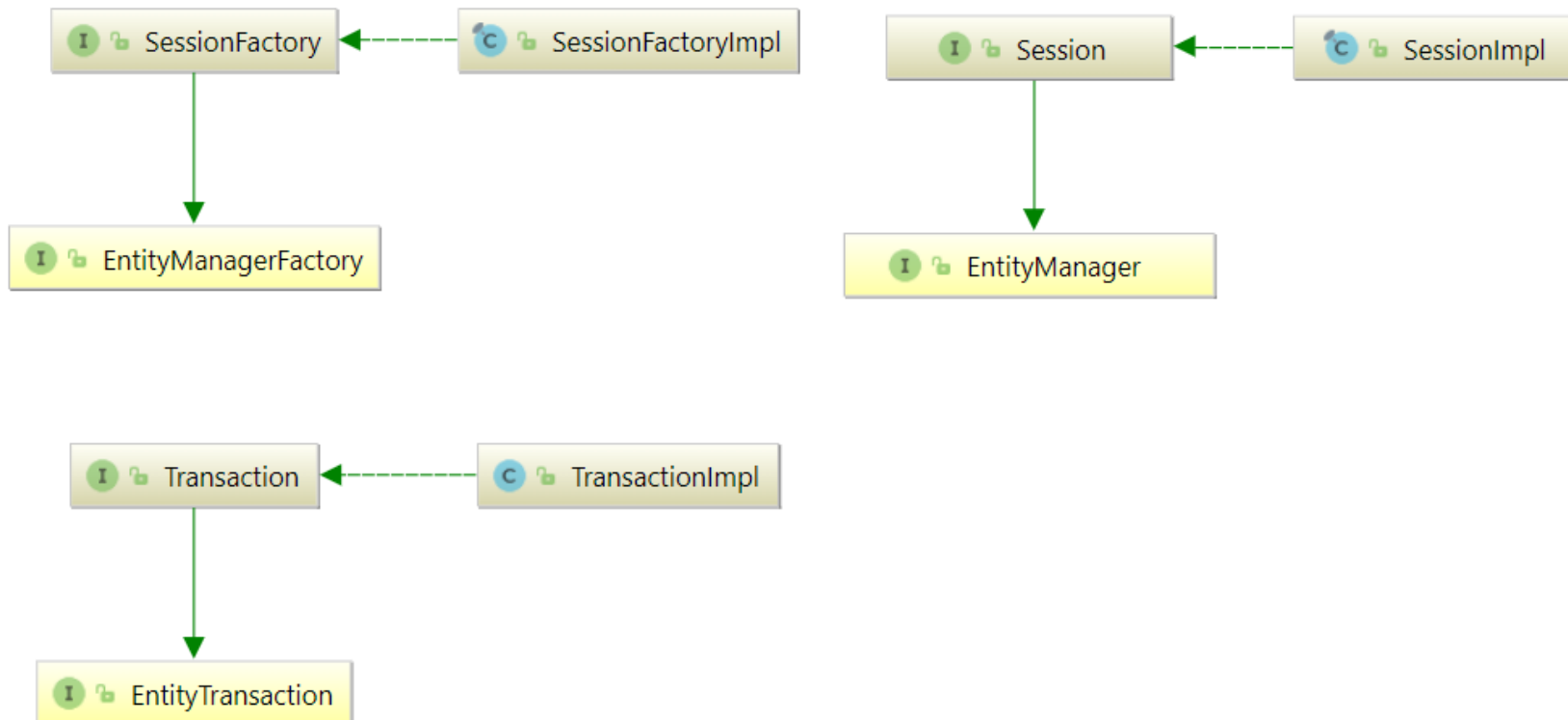
Le pattern DAO et généricité



Hibernate comme implémentation de JPA

En tant qu'une implémentation de JPA, Hibernate met en œuvre les spécifications de JPA.

Les relations entre les interfaces JPA et les implémentations spécifiques d'Hibernate peuvent être visualisées dans le diagramme suivant :



Hibernate comme implémentation de JPA

Hibernate	JPA
Hibernate est un Framework implémentant JPA et possède d'autres fonctionnalités qui lui sont propres	JPA est une spécification qui définit la gestion des données relationnelles dans les applications Java.
Session (n'est pas thread-safe)	EntityManager (n'est pas thread-safe)
Transaction	EntityTransaction
SessionFactory (thread-safe)	EntityManagerFactory (thread-safe)
Hibernate Query Language (HQL)	Java Persistence Query Language (JPQL)



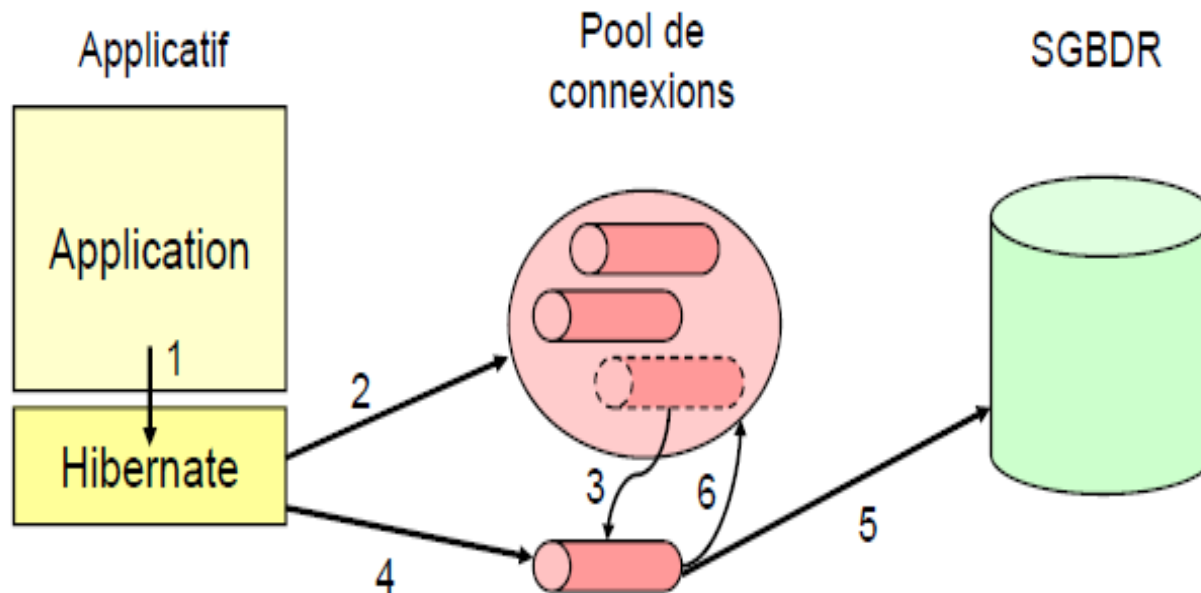
Hibernate et Performance

Hibernate implémente et intègre des techniques et des outils pour garantir une bonne performance, parmi lesquelles :

- Pool de connexion
- Lazy loading (chargement tardif)
- Systèmes de cache

Pool de connexions JDBC

- Un pool de connexion est un cache de connexions de base de données gérée de telle sorte que les connexions peuvent être réutilisés ceci permet d'améliorer les performances d'exécution des commandes sur une base de données.



Pool de connexions JDBC

- Hibernate ne gère pas directement le pool de connexions, mais il s'intègre avec plusieurs solutions existantes.
 - HikariCP (recommandé)
 - Apache Commons DBCP
 - C3p0

Lorsque on utilise Hibernate avec Spring Boot 3.x, HikariCP est déjà activé par défaut. Pour une application Hibernate standalone, il faut configurer manuellement le pool souhaité.

Pool de connexions JDBC

Avec Spring boot, nous pouvons spécifier les valeurs des paramètres du pool HikariCP en utilisant le préfixe `spring.datasource.hikari` et en ajoutant le nom du paramètre Hikari.

`spring.datasource.hikari.connectionTimeout=30000`

`spring.datasource.hikari.idleTimeout=600000`

`spring.datasource.hikari.maxLifetime=1800000`

...

Pour plus d'informations voir :

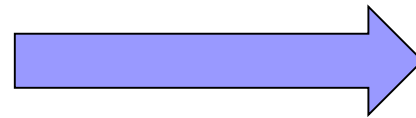
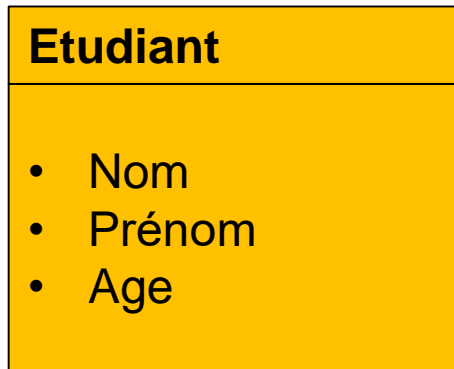
<https://github.com/brettwooldridge/HikariCP>

Lazy loading

- La chargement à la demande (ou "*lazy loading*") est la stratégie native mise en œuvre par Hibernate pour le chargement optimal de données. Elle consiste à ne charger que le minimum de données, puis de générer une nouvelle requête SQL pour récupérer les données supplémentaires lorsque celles-ci seront demandées par le programme.

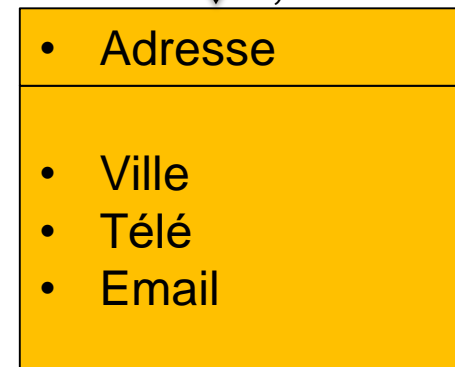
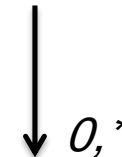
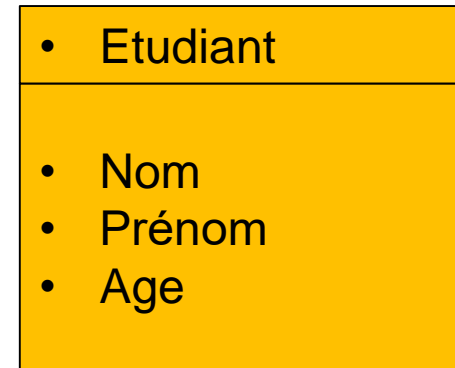
Lazy loading

Lorsque on demande à Hibernate de charger un étudiant de la base de données, il va charger juste l'objet étudiant sans sa dépendance (ses adresses)



etudiant.getAdresses()

*Hibernate effectue un SQL
Select pour récupérer
(tardivement) la
dépendance « les
adresses de l'étudiant »*



Lazy loading

- Dans la pratique, Hibernate charge tous champs de la table principale, et les clés étrangères sont stockées sous forme simplifiée (seul l'ID est renseigné), ce que l'on nomme un **proxy**. Lorsque le programme essaiera d'accéder aux membres de ce proxy, Hibernate exécutera une requête SQL et récupérera les données nécessaires afin de le remplir.

Lazy loading

- Précaution à prendre en compte en cas d'utilisation du chargement Lazy

1. Charger un objet X

2. Fermer la session



LazyInitializationException

3. Charger les dépendances de l'objet X

Lazy loading

- Précaution à prendre en compte en cas d'utilisation du chargement Lazy

1. Charger un objet X

2. Charger les dépendances de X

3. Fermer la session

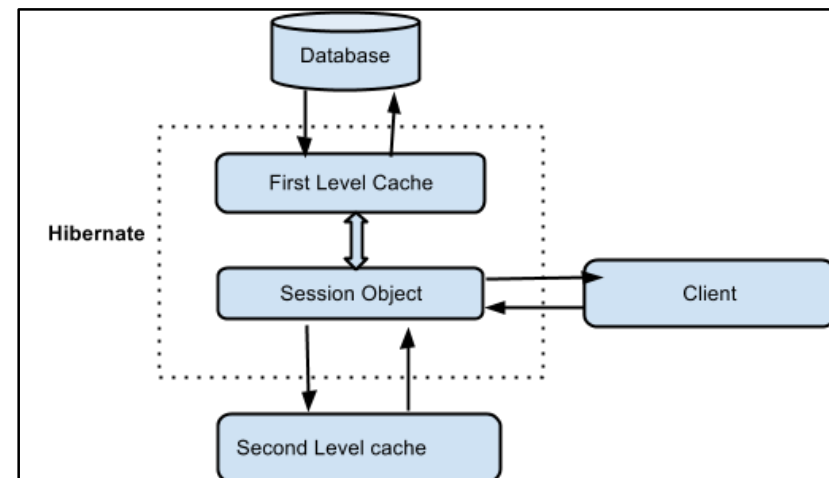
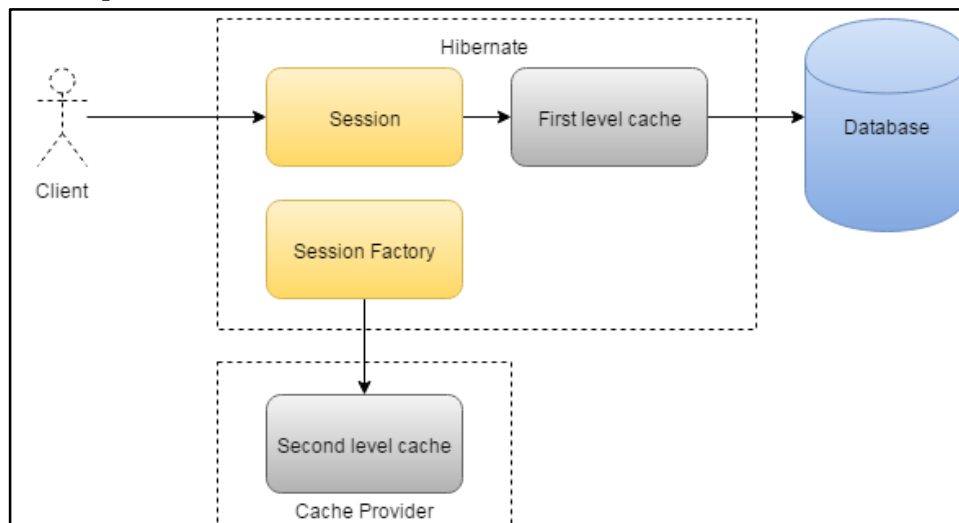
4. Accéder aux dépendances de l'objet X



Les différents caches d'Hibernate

Hibernate fonctionne avec deux niveaux de cache. **Le cache de premier niveau (obligatoire)** est lié à la session Hibernate. Quand la session est fermée, le cache n'a plus d'existence. **Le cache de second niveau (optionnel)** est quant à lui lié à la session factory d'Hibernate. La portée de ce cache est la JVM. Les objets cachés sont donc visibles depuis l'ensemble des transactions.

Il existe enfin un cache de requête qui permet de **cacher les résultats des requêtes exécutées**.





JPA : Mapping des associations et de l'héritage

Relation unidirectionnelle de type One-to-One

```
package com.bo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
@SuppressWarnings("unused")

@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String cin;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="id_adresse_etudiant")
    private Adresse adresse;

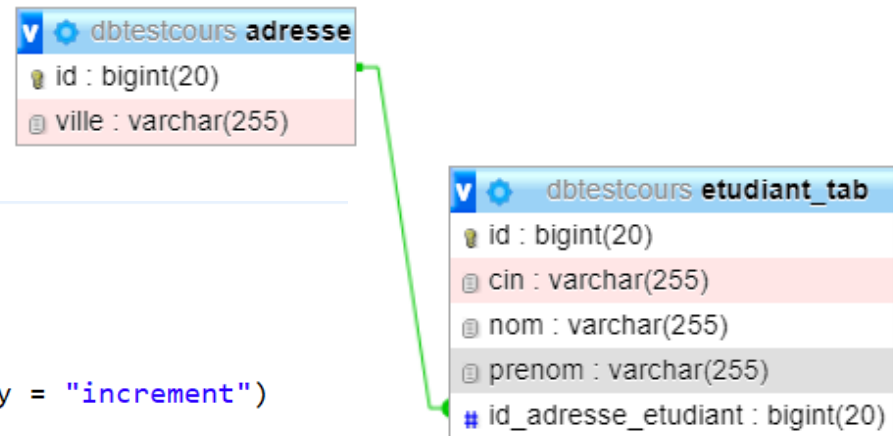
    private String prenom;
    //getters & setters
}

@Entity
public class Adresse {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;

    private String ville;

    // getters/setters
}
```



dbtestcours adresse
id : bigint(20)
ville : varchar(255)

dbtestcours etudiant_tab
id : bigint(20)
cin : varchar(255)
nom : varchar(255)
prenom : varchar(255)
id_adresse_etudiant : bigint(20)

Relation bidirectionnelle de type One-to-One

```
@Entity
public class Adresse {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;

    @OneToOne(mappedBy = "adresse" )
    private Etudiant etudiant;

    private String ville;

    public Long getId() {
        return id;
    }
}
```

adresse est le nom de l'attribut
dans la classe Etudiant

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name="id_adresse_etudiant")
private Adresse adresse;
```

Exemple sur machine

Les codes sources sont disponibles sur :

<https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xztZEC5V6x0s/view>

- Mapping d'une relation one to one Unidirectionnelle
Exemple_Cours_Hibernate_2_one_to_one_uni
- Mapping d'une relation one to one bidirectionnelle :
Exemple_Cours_Hibernate_2_one_to_one_bi



Relation unidirectionnelle de type One-to-many

```
@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String cin;
```

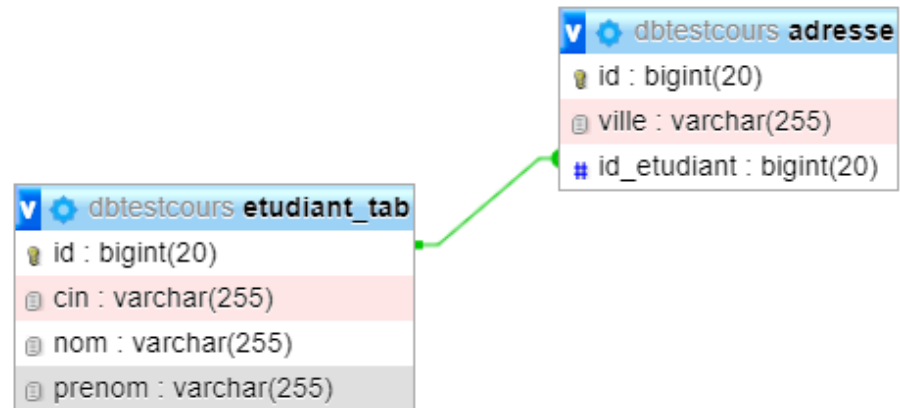
```
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="id_etudiant")
    private Set<Adresse> adresses;
```

```
    public Set<Adresse> getAdresses() {
        return adresses;
    }
}
```

```
@Entity
public class Adresse {
```

```
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;
```

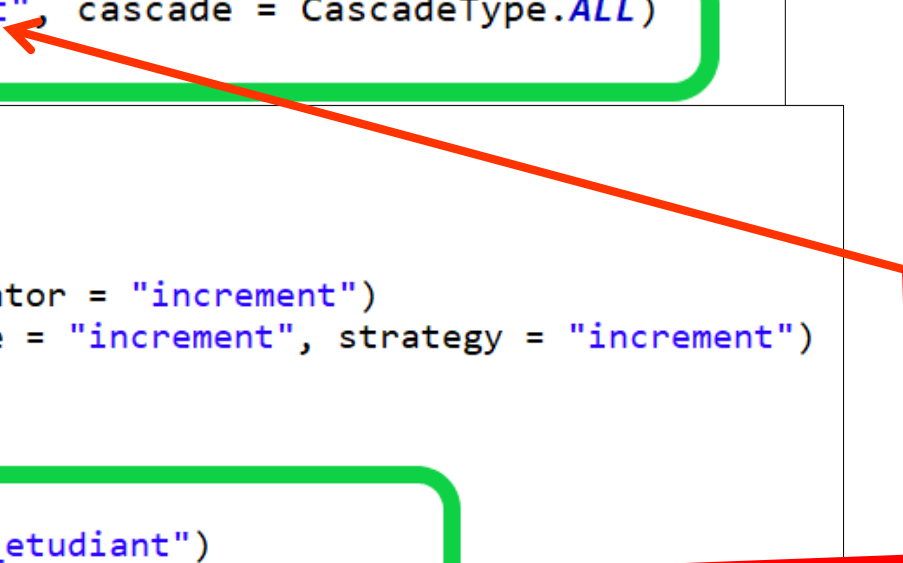
```
    private String ville;
```



Relation bidirectionnelle de type One-to-many

```
@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String cin;

    @OneToOne(mappedBy = "etudiant", cascade = CascadeType.ALL)
    private Set<Adresse> adresses;
```



```
@Entity
public class Adresse {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;

    @ManyToOne
    @JoinColumn(name = "id_etudiant")
    private Etudiant etudiant;

    private String ville;
```

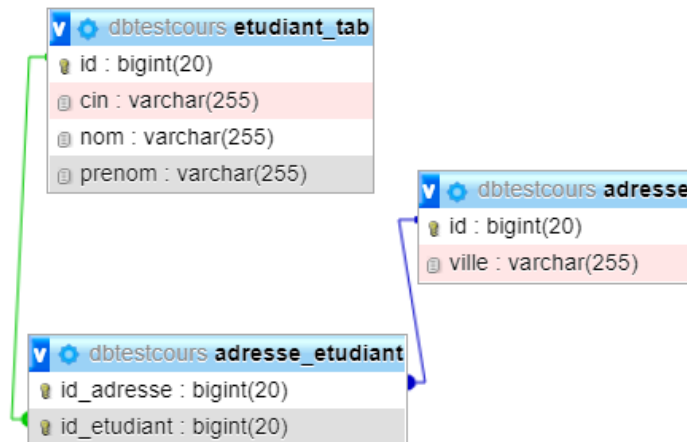
Relation many-to-many

■ Classe Etudiant

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(name = "adresse_etudiant",
joinColumns = @JoinColumn(name = "id_etudiant"),
inverseJoinColumns = @JoinColumn(name = "id_adresse"))
private Set<Adresse> adresses = new HashSet<Adresse>();
```

■ Classe Adresse

```
@ManyToMany(mappedBy = "adresses")
private Set<Etudiant> etudiants = new HashSet<Etudiant>();
```



Exemple sur machine

Les codes sources sont disponibles sur :

[https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xztZEC5V6x0s/vi
ew](https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xztZEC5V6x0s/vi
ew)

- Mapping d'une relation one to many unidirectionnelle :
Exemple_Cours_Hibernate_2_one_to_many_uni
- Mapping d'une relation one to many bidirectionnelle :
Exemple_Cours_Hibernate_2_one_to_many_bi
- Mapping many-to-many
Exemple_Cours_Hibernate_2_many_to_many



Mapping de l'héritage

■ Différentes stratégies pour le mapping:

Les bases de données relationnelles ne disposent pas de la notion de l'héritage: pour résoudre ce problème, la spécification JPA propose plusieurs stratégies pour faire la correspondance entre une hiérarchie d'héritage et les tables de la base de données:

- **Avec l'annotation @MappedSuperclass**
- **Stratégie à Table unique:** les entités de différentes classes avec une classe mère commune seront placées dans une seule table.
- **Table jointe :** chaque classe a sa table et l'interrogation d'une entité de sous-classe nécessite de joindre les tables.
- **Table par classe :** Chaque classe est associée à sa propre table. Toutes les propriétés d'une classe sont dans sa table, donc aucune jointure n'est requise.

Héritage avec @MappedSuperclass

```
@MappedSuperclass
public class Personne {
```

Notez que cette classe n'a plus d'annotation @Entity, car elle ne aura pas une table en base de données

Si nous utilisons cette stratégie, les classes mères ne peuvent pas contenir d'associations avec d'autres entités.

```
@Id
@GeneratedValue(generator = "increment")
@GenericGenerator(name = "increment", strategy = "increment")
private Long id;
private String nom;
private String cin;
private String prenom;
```

```
@Entity
public class Etudiant extends Personne {

    private String cne;

    public String getCne() {
        return cne;
    }

    public void setCne(String cne) {
        this.cne = cne;
    }
}
```

Résultat dans MySQL:

dbtestcours etudiant	
id	bigint(20)
cin	varchar(255)
nom	varchar(255)
prenom	varchar(255)
cne	varchar(255)

Stratégie Table unique (*Single Table*)

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Personne {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "i
    private Long id;
    private String nom;
    private String prenom;
```

L'identifiant des entités est également défini dans la super-classe.

```
@Entity
public class Prof extends Personne {
    private String cin;

    public String getCin() {
        return cin;
    }

    public void setCin(String cin) {
        this.cin = cin;
    }
}
```

```
@Entity
public class Etudiant extends Personne {

    private String cne;

    public String getCne() {
        return cne;
    }







    public void setCne(String cne) {
        this.cne = cne;
    }
}
```

Stratégie Table unique (*Single Table*)

Étant donné que les enregistrements de toutes les entités seront dans la même table, Hibernate a besoin d'un moyen de les différencier. Par défaut, cela se fait via une colonne discriminante appelée DTYPE qui a le nom de l'entité comme valeur.

dbtestcours	personne
DTYPE	: varchar(31)
id	: bigint(20)
nom	: varchar(255)
prenom	: varchar(255)
cne	: varchar(255)
cin	: varchar(255)

+ Options

				DTYPE	id	nom	prenom	cne	cin			
<input type="checkbox"/>		Éditer		Copier		Supprimer	Etudiant	1	boudaa	Mohamed	A11112	NULL
<input type="checkbox"/>		Éditer		Copier		Supprimer	Prof	2	boudaa	Mohamed	NULL	A11111

Stratégie Table unique (*Single Table*)

Pour personnaliser la colonne discriminateur, nous pouvons utiliser l'annotation *@DiscriminatorColumn*:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type_personne", discriminatorType = DiscriminatorType.INTEGER)
public class Personne {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String prenom;
```

```
@Entity
@DiscriminatorValue("2")
public class Prof extends Personne {
    private String cin;

    public String getCin() {
        return cin;
    }

    public void setCin(String cin) {
        this.cin = cin;
    }
}
```

```
@Entity
@DiscriminatorValue("1")
public class Etudiant extends Personne {
    private String cne;

    public String getCne() {
        return cne;
    }

    public void setCne(String cne) {
        this.cne = cne;
    }
}
```

type_personne	id	nom	prenom	cne	cin
1	1	boudaa	Mohamed	A11112	NULL
2	2	boudaa	Mohamed	NULL	A11111

Exemple sur machine

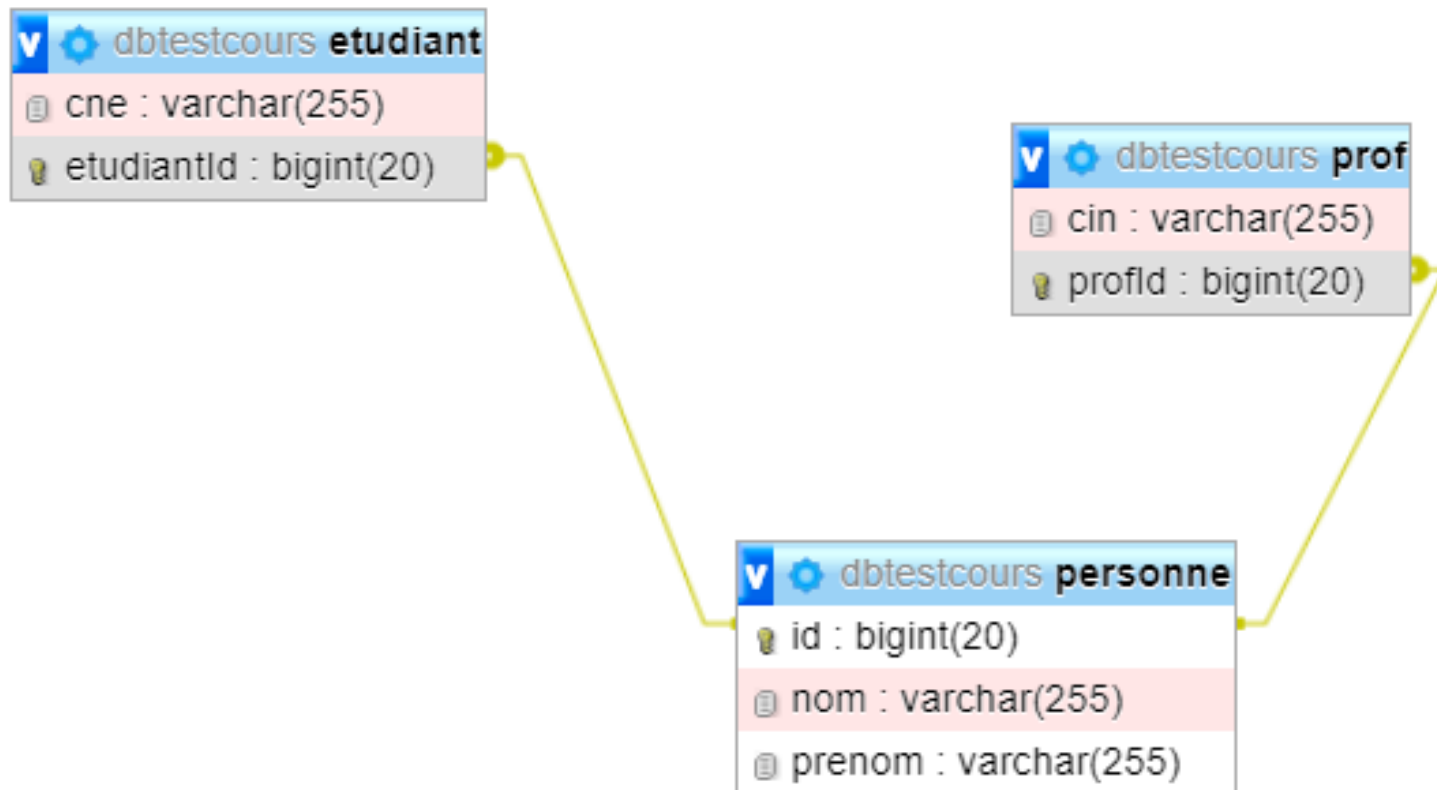
Les codes sources sont disponibles sur :

[https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xztZEC5V6x0s/vi
ew](https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xztZEC5V6x0s/vi
ew)

- Héritage avec @MappedSuperclass
Exemple_Cours_Hibernate_2_heritage_strategie01
- Stratégie *Single Table (1)*
Exemple_Cours_Hibernate_2_heritage_strategie2
- Stratégie *Single Table (2)*
Exemple_Cours_Hibernate_2_heritage_strategie2_1



Stratégie avec table de jointure (*Joined Table*)



Stratégie avec table de jointure (*Joined Table*)

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Personne {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String prenom;
```

```
@Entity
@PrimaryKeyJoinColumn(name = "etudiantId")
public class Etudiant extends Personne {

    private String cne;
```

```
@Entity
@PrimaryKeyJoinColumn(name = "profId")
public class Prof extends Personne {

    private String cin;
```

En utilisant cette stratégie, chaque classe de la hiérarchie est mappée à sa table. La seule colonne qui apparaît à plusieurs reprises dans toutes les tables est l'identifiant, qui sera utilisé pour les joindre en cas de besoin

Stratégie table par classe (Table Per Class)

v	dbtestcours	personne
🔑		id : bigint(20)
📄		nom : varchar(255)
📄		prenom : varchar(255)

v	dbtestcours	etudiant
🔑		id : bigint(20)
📄		nom : varchar(255)
📄		prenom : varchar(255)
📄		cne : varchar(255)

v	dbtestcours	prof
🔑		id : bigint(20)
📄		nom : varchar(255)
📄		prenom : varchar(255)
📄		cin : varchar(255)

Stratégie table par classe (Table Per Class)

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Personne {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String prenom;
```

```
@Entity
public class Etudiant extends Personne {

    private String cne;
```

```
@Entity
public class Prof extends Personne {

    private String cin;
```

Ce n'est pas très différent du simple mappage de chaque entité sans héritage. La différence est lors d'exécution d'une sélection sur la classe de base, elle retournera également tous les enregistrements des sous-classes en utilisant une instruction UNION en arrière-plan.
(Requêtes polymorphiques - *Polymorphic Queries*)

Requêtes polymorphiques

- ❑ l'interrogation d'une classe de base récupérera également toutes les entités de sous-classes.
- ❑ Cela fonctionne également pour n'importe quelle super-classe ou interface, qu'il s'agisse d'une @MappedSuperclass ou non.

```
Session session = SessionFactoryBuilder.getSessionFactory().getCurrentSession();

Transaction tx = session.beginTransaction();
// Requete sur la super-classe Personne
String hqlQuery = "from Personne ";

Query<Etudiant> query = session.createQuery(hqlQuery);

List<Etudiant> list = query.getResultList();

tx.commit();

for(Personne it :list) {
    System.out.println(it);
}
```

Exemple sur machine

Les codes sources sont disponibles sur :

<https://drive.google.com/file/d/1xGI0ZMusMW4NBwnJjgF2xztZEC5V6x0s/view>

- Stratégie avec table de jointure
Exemple_Cours_Hibernate_2_heritage_strategie3
- Stratégie *Table Per Class* & Requêtes polymorphiques:
Exemple_Cours_Hibernate_2_heritage_strategie4





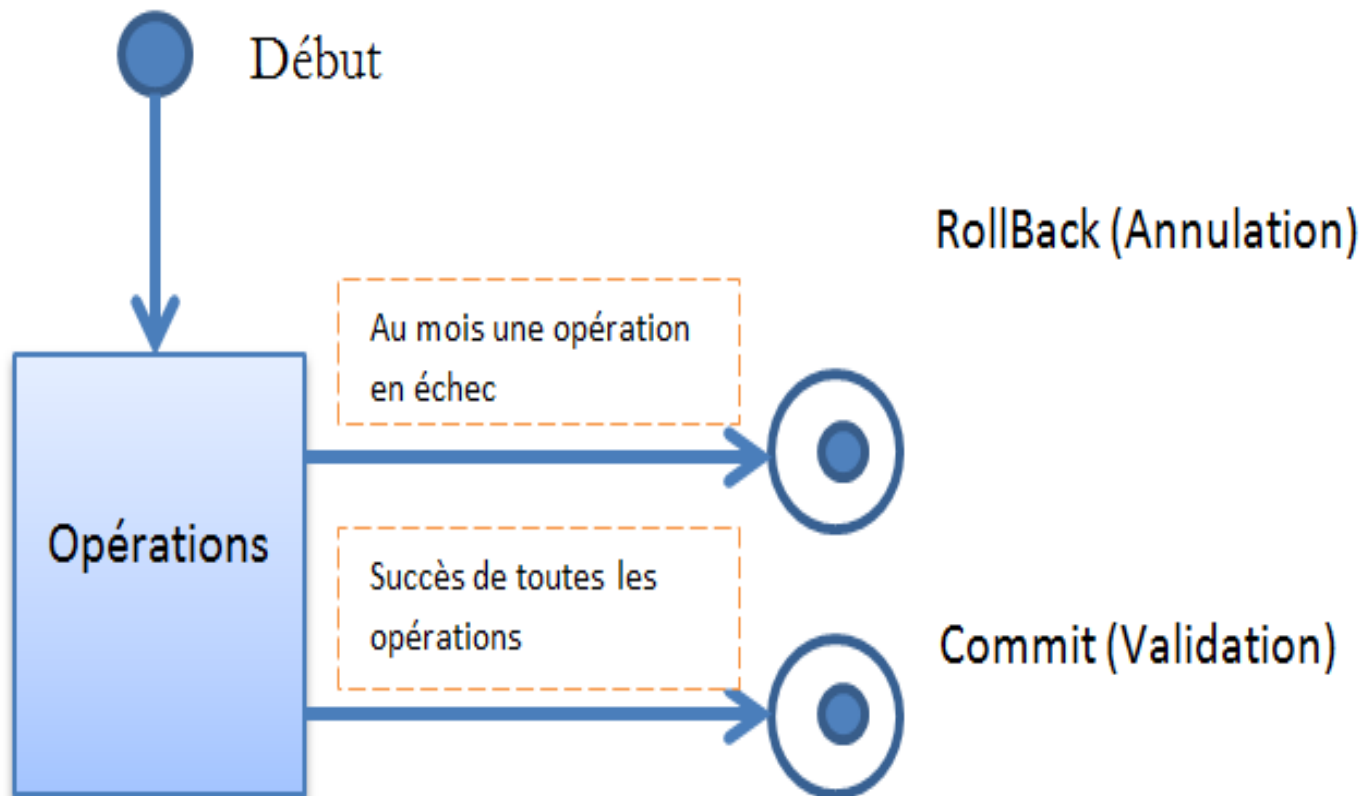
Spring et gestion des transactions

Gestion des transactions

- Une transaction est un ensemble indivisible d'opérations dans lequel tout aboutit ou rien n'aboutit.
- La gestion des transactions est un élément essentiel dans les applications d'entreprises car elle permet de garantir l'intégrité et la cohérence des données.
- Sans les transactions les données et les ressources pourraient être endommagées et laissés dans un état incohérent.
- Oublier de gérer les transactions dans une application n'engendre pas nécessairement de dysfonctionnement visible. Cependant, cet oubli ne permettant pas de garantir la consistance des données manipulées, des erreurs délicates à détecter peuvent survenir.
- La notion de transaction est donc primordiale et doit être mise en œuvre dans toute application réalisant des mises à jour dans des sources de données.

Gestion des transactions

- Une transaction doit vérifier les propriétés fondamentales suivantes, communément désignées sous l'acronyme **ACID** (**a**tomicté, **c**onsistance, **i**solation, **d**urabilité) :



Gestion des transactions

■ Atomicité :

Une transaction est une opération atomique constituée d'une suite d'opérations. Cette propriété garantit que toutes les actions sont entièrement exécutées ou qu'elles n'ont aucun effet.

Gestion des transactions

■ Cohérence :

la transaction laisse toujours les données dans un état cohérent.

Cette propriété assure que chaque transaction amènera le système d'un état valide à un autre état valide. Tout changement à la base de données doit être valide selon toutes les règles définies.

Gestion des transactions

■ Isolation :

Puisque plusieurs transactions peuvent manipuler le même jeu de données au même moment, chaque transaction doit être isolée des autres afin d'éviter la corruption des données

La propriété d'isolation **assure que l'exécution simultanée de transactions produit le même état que celui qui serait obtenu par l'exécution en série des transactions**. Chaque transaction doit s'exécuter en isolation totale : si T1 et T2 s'exécutent simultanément, alors chacune doit demeurer indépendante de l'autre.

Gestion des transactions

■ Durabilité :

Dès lors qu'une transaction est terminée, les résultats doivent être durables dans le temps. C'est-à-dire les modifications sont définitives et entièrement visibles par le reste des applications qui utilisent la même source de données.

Types de transactions :

Il existe deux grands types de transactions, les transactions locales et les transactions globales.

- **Transactions locales.** Adaptées lorsqu'une seule ressource transactionnelle est utilisée.
- **Transactions globales.** Utilisables si une ou plusieurs ressources sont présentes. À partir de deux ressources, nous sommes toujours en présence de transactions globales. En cas d'utilisation des transactions globales, le gestionnaire de transactions est externalisé par rapport aux ressources et est capable de dialoguer avec elles grâce à des interfaces normalisées.
- Les transactions globales sont cependant beaucoup plus difficiles à mettre en œuvre. En effet, ce type de transaction nécessite la mise en œuvre d'un gestionnaire de transactions externe. Ce dernier est souvent fourni par le serveur d'applications.
- Spring masque la complexité derrière les différents framework de gestion de transactions avec une API générique.



Problèmes liés aux accès concourants :

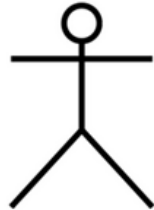
- Lecture sale
- Lecture non reproductible
- Lecture fantôme

Problèmes liés aux accès concourants :

- **Lecture sale:** Pour deux transactions T1 et T2, T1 lit un champ qui a été mis à jour par T2 mais pas encore validé. Plus tard, si T2 est annulée, la valeur du champ lu par T1 était en réalité une valeur temporaire désormais invalide.

Lecture sale

Etape 1

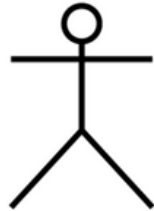


Acteur 1

Début de la transaction 1
Récupération de l'entité E
Modification de E

Entité
E

Etape 2

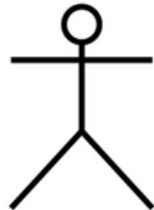


Acteur 2

Début de la transaction 2
Récupération de l'entité E

Entité
E

Etape 3



Acteur 1

Annulation de la
transaction 1

Entité
E

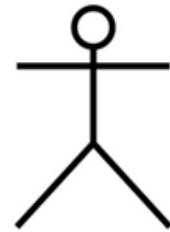
- L'acteur 2 travaille sur des données fausses non validés par l'acteur 1 (données sales)

Lecture non reproductible

- **Lecture non reproductible:** Pour deux transactions T1 et T2, T1 lit un champ et T2 met ensuite ce champ à jour. Plus tard, si T1 lit à nouveau le même champ, la valeur est différente.

Lecture non reproductible

Etape 1

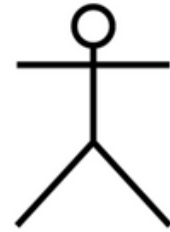


Acteur 1

Début de la transaction 1
Récupération de l'entité E

Entité
E

Etape 2

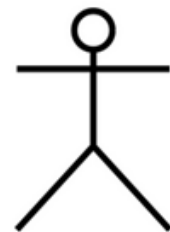


Acteur 2

Début de la transaction 2
Récupération et
modification de l'entité E
commit de la transaction 2

Entité
E

Etape 3



Acteur 1

Toujours dans la
transaction 1 on récupère
la même entité E

Entité
E

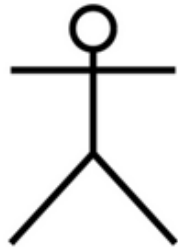
- Dans ce cas au sein d'une même transaction la lecture successive d'une même entité donne deux résultats différents.

Lecture fantôme

- **Lecture fantôme:** Pour deux transactions T1 et T2, T1 lit quelques lignes d'une table, puis T2 insère de nouvelles lignes dans la table. Plus tard, si T1 lit à nouveau la même table, il existe des lignes supplémentaires.

Lecture fantôme

Etape 1

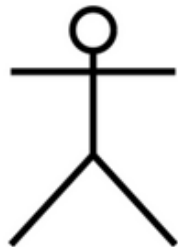


Acteur 1

Début de la transaction 1
Récupération des entités E
répondant à un jeu de
critères X

Entité
E

Etape 2

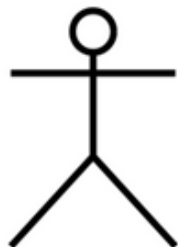


Acteur 2

Début de la transaction 2
Insertion de nouvelles
entités E répondant au jeu
de critères X
commit de la transaction2

Entité
E

Etape 3



Acteur 1

Réexécution de la
recherche selon le même
jeu de critères X

Entité
E

- Dans ce cas une même recherche fait apparaitre ou disparaître des entités.

Niveaux d'isolation transactionnelle

- En fonction du type de l'application, il existe plusieurs façons de gérer les collisions. Au niveau de la connexion JDBC, le niveau d'isolation transactionnelle permet de spécifier le comportement de la transaction selon la base de données utilisée.
- Il existe en standard quatre niveaux d'isolation transactionnelle

Niveau d'isolation	Lecture sale	Lecture non répétable	Lecture fantôme
TRANSACTION_READ_UNCOMMITTED	OUI	OUI	OUI
TRANSACTION_READ_COMMITTED	NON	OUI	OUI
TRANSACTION_REPEATABLE_READ	NON	NON	OUI
TRANSACTION_SERIALIZABLE	NON	NON	NON

Niveaux d'isolation transactionnelle

- En théorie, les transactions doivent être totalement isolées les unes des autres (c'est-à dire sérialisables) pour éviter les problèmes précédents. Cependant, un tel niveau d'isolation a un impact important sur les performances car les transactions s'exécutent alors l'une après l'autre. Dans la pratique, les transactions peuvent s'exécuter à des niveaux d'isolation plus faibles de manière à améliorer les performances.

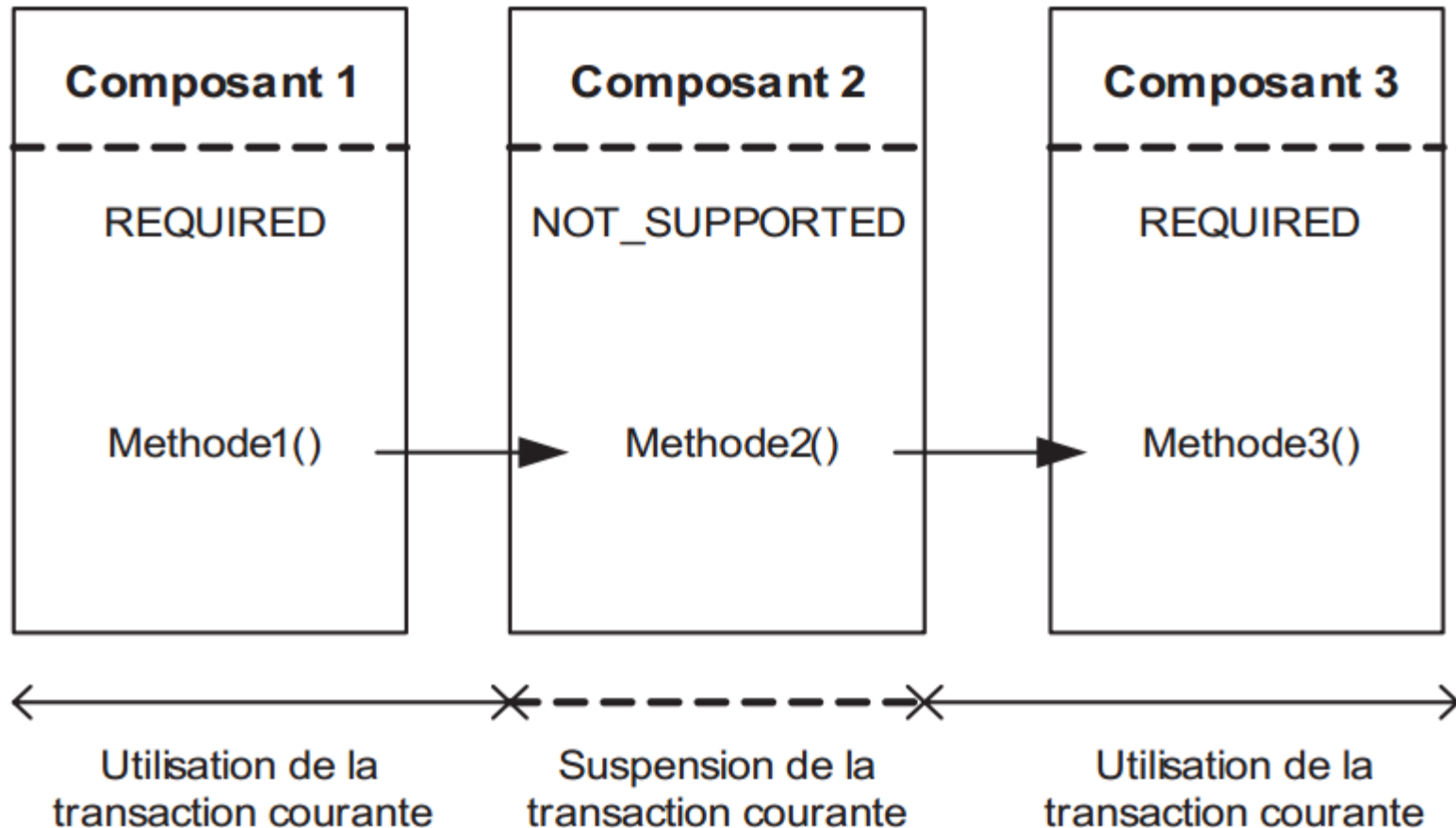
Comportements transactionnels: propagation

Mot-clé de comportement transactionnel	Description
REQUIRED	La méthode doit forcément être exécutée dans un contexte transactionnel s'il existe. S'il n'existe pas lors de l'appel, il est créé.
SUPPORTS	La méthode peut être exécutée dans un contexte transactionnel s'il existe. Dans le cas contraire, la méthode est tout de même exécutée, mais hors d'un contexte transactionnel.
MANDATORY	La méthode doit forcément être exécutée dans un contexte transactionnel. Si tel n'est pas le cas, une exception est levée.
REQUIRES_NEW	La méthode impose la création d'une nouvelle transaction pour la méthode.
NOT_SUPPORTED	La méthode ne supporte pas les transactions. Si un contexte transactionnel existe lors de son appel, celui-ci est suspendu.
NEVER	La méthode ne doit pas être exécutée dans un contexte transactionnel. Si tel est le cas, une exception est levée.
NESTED	La méthode est exécutée dans une transaction imbriquée si un contexte transactionnel existe lors de son appel.

Comportements transactionnels: propagation

Type de comportement transactionnel	Transaction initiale	Transaction utilisée
REQUIRED	Aucune T1	T1 T1
SUPPORTS	Aucune T1	Aucune T1
MANDATORY	Aucune T1	Erreur T1
REQUIRES_NEW	Aucune T1	T1 T2
NOT_SUPPORT	Aucune T1	Aucune Aucune
NEVER	Aucune T1	Aucune Erreur
NESTED	Aucune T1	T1 T1 (imbriquée)

Comportements transactionnels: propagation



La notion de verrou

- La notion de verrou est implémentée pour fournir des solutions aux différents types de collisions. Il existe deux possibilités, le **verrouillage pessimiste** et le **verrouillage optimiste**.

Verrouillage pessimiste

- Ce mécanisme de verrouillage fort est géré directement par le système de stockage des données. Pendant toute la durée du verrou, aucune autre application ou fil d'exécution ne peut accéder à la donnée. Pour les bases de données relationnelles, cela se gère directement au niveau du langage SQL. À l'image d'Hibernate, plusieurs frameworks facilitent l'utilisation de ce type de verrou. Une requête SQL de type `select for update` est alors utilisée. Ce verrouillage particulièrement restrictif peut impacter les performances des applications. En effet, ces dernières ne peuvent accéder à l'enregistrement tant que le verrou n'est pas levé.

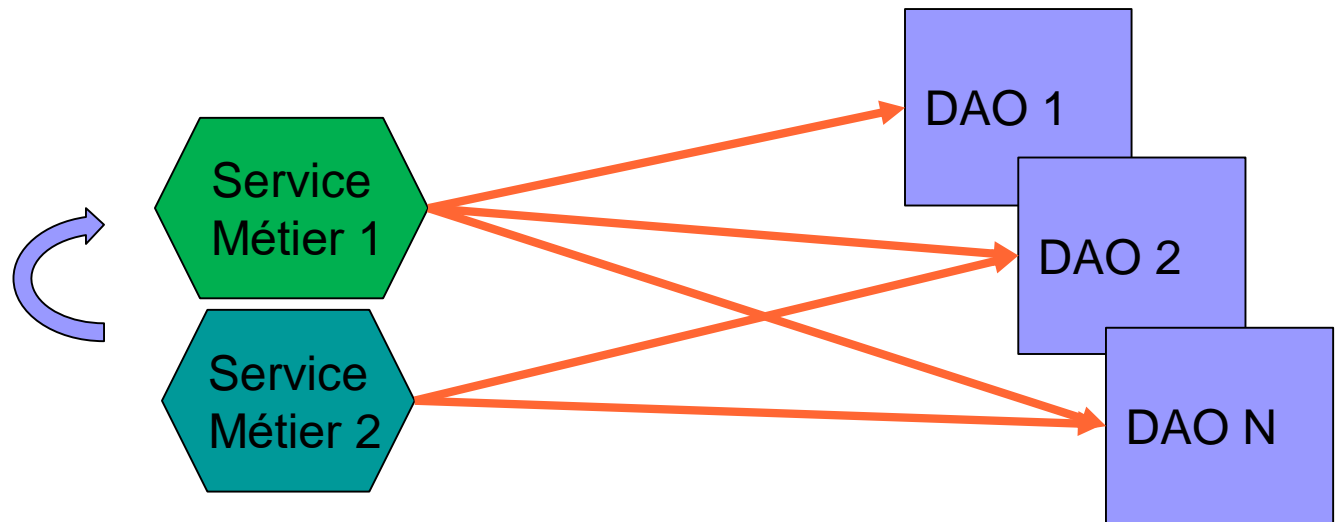
Verrouillage optimiste :

- Ce type de verrouillage adopte la logique de détection de collision. Son principe est qu'il peut être acceptable qu'une collision survienne, à condition de pouvoir la détecter et la résoudre.
- Ce type de verrouillage doit être implémenté dans l'application elle-même. Il ne nécessite pas de verrou dans le système de stockage des données. Pour les bases de données relationnelles, il est généralement implémenté en ajoutant une colonne aux différentes tables impactées. Cette colonne représente une version ou un indicateur de temps indiquant l'état de l'enregistrement lorsqu'il est lu. De ce fait, si cet état change entre la lecture et la modification, nous nous trouvons dans le cas d'un accès concourant.

Gestion de la démarcation

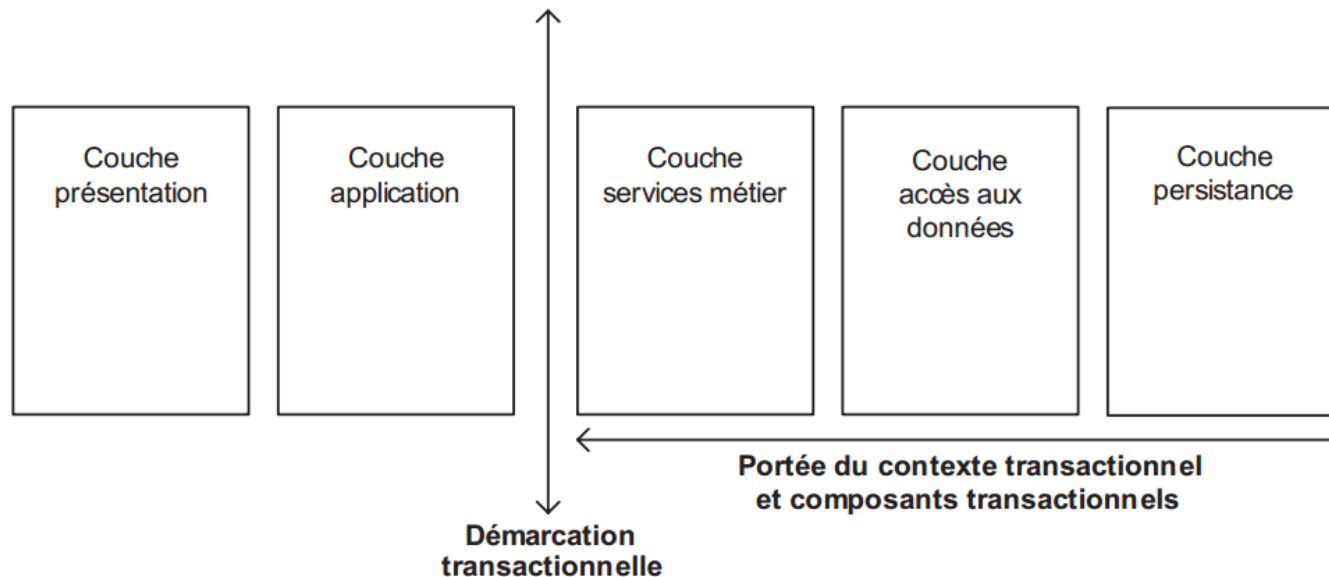
- Généralement les transactions doivent être gérées dans la couche services métier et non pas dans la couche DAO. En effet:
 - Dans un service on pourra avoir besoin d'exécuter plusieurs méthode DAO au sein d'une même transaction.
 - Les DAOs ne seront pas réutilisables si nous gérons la transaction au niveau DAO

Le support des transactions doit de plus être suffisamment flexible pour permettre de gérer les appels entre services.



La démarcation transactionnelle doit être réalisée au niveau des services métier. Ces derniers peuvent s'appuyer sur plusieurs composants d'accès aux données. Plusieurs appels à des méthodes de ces composants sous-jacents peuvent donc être réalisés dans une même transaction.

Gestion de la démarcation



- La démarcation consiste à spécifier le commencement et la fin de la transaction.
- La définition de types de comportement transactionnel rend ce support flexible et déclaratif. Spring implémente ces stratégies dans sa gestion des transactions.
- Avec l'API générique de démarcation de Spring on peut éviter d'utiliser une connexion JDBC ou une session d'un outil d'ORM dans la couche service métier (c'est un anti-pattern).

Configuration de la gestion des Transactions avec Spring

En utilisant une configuration par code Java il faut ajouter l'annotation

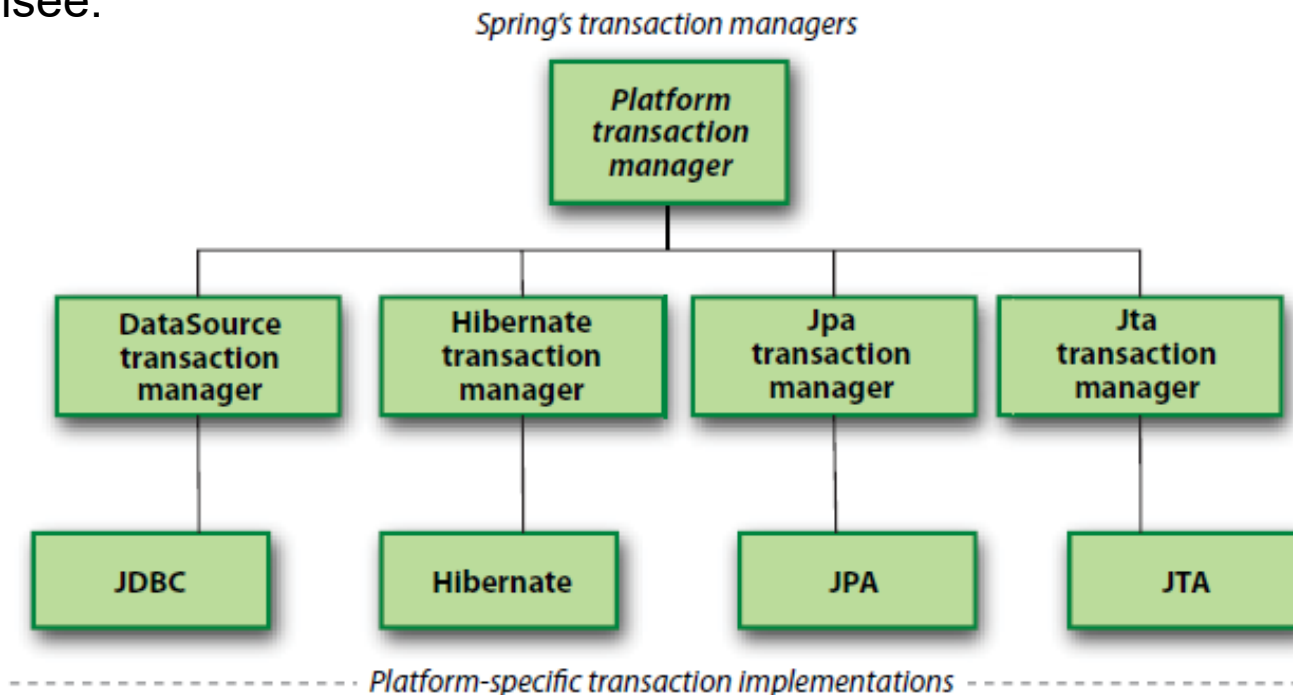
@EnableTransactionManagement pour activer la prise en charge

transactionnelle:

```
@Configuration
@EnableTransactionManagement
public class AppConfig {
    ...
}
```

Configuration de la gestion des Transactions avec Spring

Le module Spring Transaction essaie de simplifier cette situation en utilisant une interface unique pour la gestion des transactions : **PlatformTransactionManager**. Le module fournit ensuite plusieurs implémentations selon la technologie sous-jacente utilisée.



Configuration de la gestion des Transactions avec Spring

■ Exemple de configuration du Gestionnaire de transaction pour Hibernate

```
@Configuration
@EnableTransactionManagement
public class AppConfig {
    @Bean
    @Autowired
    public PlatformTransactionManager transactionManager(final
SessionFactory sessionFactory) {
        final HibernateTransactionManager txManager = new
        HibernateTransactionManager();
        txManager.setSessionFactory(sessionFactory);
        return txManager;
    }
}
```

Configuration de la gestion des Transactions avec Spring

■ Exemple de configuration du Gestionnaire de transaction pour JPA

```
@Configuration
@EnableTransactionManagement
public class AppConfig {
    @Bean
    @Autowired
    public PlatformTransactionManager transactionManager(final
EntityManagerFactory emf){
        JpaTransactionManager tm = new JpaTransactionManager();
        tm.setEntityManagerFactory(emf);
        return tm;
    }
}
```

Configuration de la gestion des Transactions avec Spring

- Annoter les classes services avec **@transactional**

```
public class UserService {
```

```
    @Transactional
```

```
    public Long registerUser(User user) {
```

```
        //Code de la méthode service
```

```
        // Généralement, les méthodes services peuvent
```

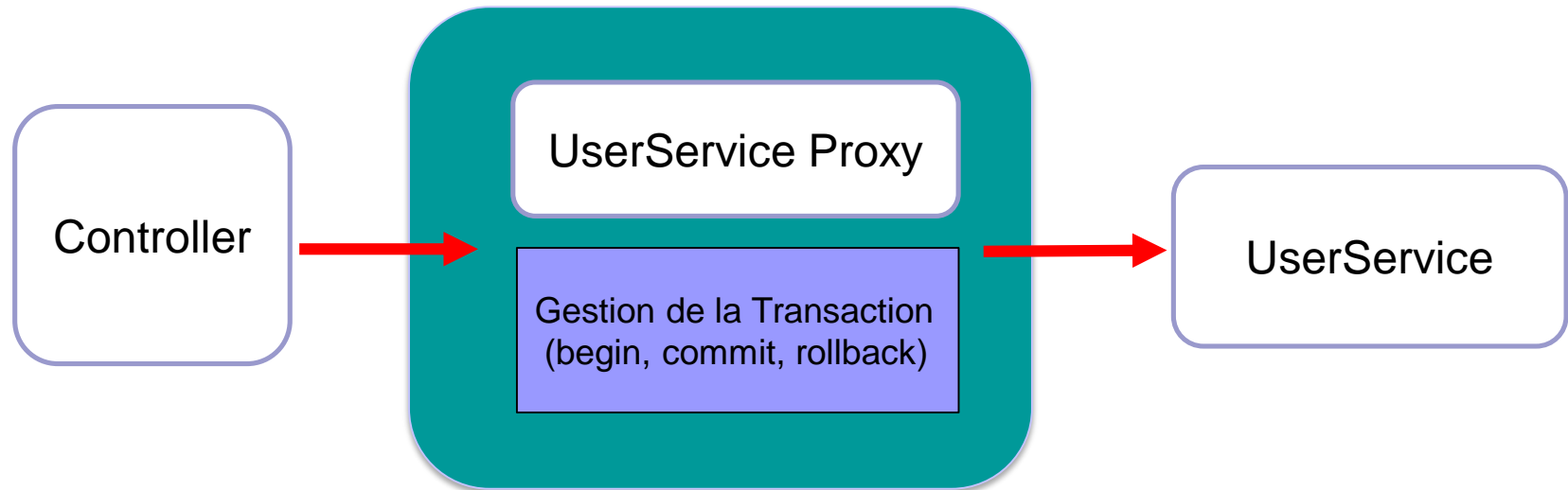
```
        //exécuter plusieurs méthodes de différents DAOs
```

```
    }
```

```
}
```

Configuration de la gestion des Transactions avec Spring

- Spring utilise des proxies en background



Utilise le gestionnaire
de transactions implémentant
`PlatformTransactionManager`

Configuration de la gestion des Transactions avec Spring

■ Configurer le niveau de propagation avec @transactional

@Transactional(propagation = Propagation.REQUIRED) **(Valeur par défaut)**

@Transactional(propagation = Propagation.REQUIRES_NEW)

@Transactional(propagation = Propagation.SUPPORTS)

@Transactional(propagation = Propagation.MANDATORY)

@Transactional(propagation = Propagation.NOT_SUPPORTED)

@Transactional(propagation = Propagation.NEVER)

@Transactional(propagation = Propagation.NESTED)

Configuration de la gestion des Transactions avec Spring

■ Configurer le niveau d'isolation avec @transactional

//Utiliser le niveau d'isolement par défaut du SGBD sous-jacent..

@Transactional(isolation = Isolation.**DEFAULT**) (**Valeur par défaut**)

//Les autres niveaux correspondent aux niveaux d'isolement vus précédemment

@Transactional(isolation = Isolation.**READ_COMMITTED**)

@Transactional(isolation = Isolation.**READ_UNCOMMITTED**)

@Transactional(isolation = Isolation.**REPEATABLE_READ**)

@Transactional(isolation = Isolation.**SERIALIZABLE**)

Exemple sur machine

Les codes sources sont disponibles sur :

https://drive.google.com/drive/folders/1VUbOnsq0KrcIY1N7UIpjKnv8rCZ0-MZc?usp=drive_link

