RAPPORT SUR LA GESTION DE LA PERSISTANCE DE LA BASE DE DONNÉES ORACLE À TRAVERS : HIBERNATE, SPÉCIFICATION JPA ET SPRING DATA

2024/2023

Préparé par

josué KPATCHA
Othman EL HADRATI

Encadré par :

P.CHERRADI

SOMMAIRE

INTRODUCTION

Ce rapport offre un aperçu détaillé de la gestion de la persistance des données dans une base de données Oracle en utilisant trois technologies principales : JDBC, JPA et Hibernate, ainsi que Spring Data.

Nous commençons par définir les exigences fondamentales qui influencent le choix de la technologie de persistance. En soulignant les limites de JDBC, nous mettons en lumière les défis rencontrés lors de son utilisation directe.

Ensuite, nous explorons la spécification JPA et ses différents types de mapping, offrant ainsi une abstraction pour la gestion de la persistance des données. Nous plongeons ensuite dans Hibernate, un framework ORM populaire qui simplifie la persistance des données . De là, nous examinons Spring Data, qui fournit une abstraction supplémentaire et simplifie l'accès aux données. Enfin, des exemples pratiques sont fournis pour illustrer chaque approche.

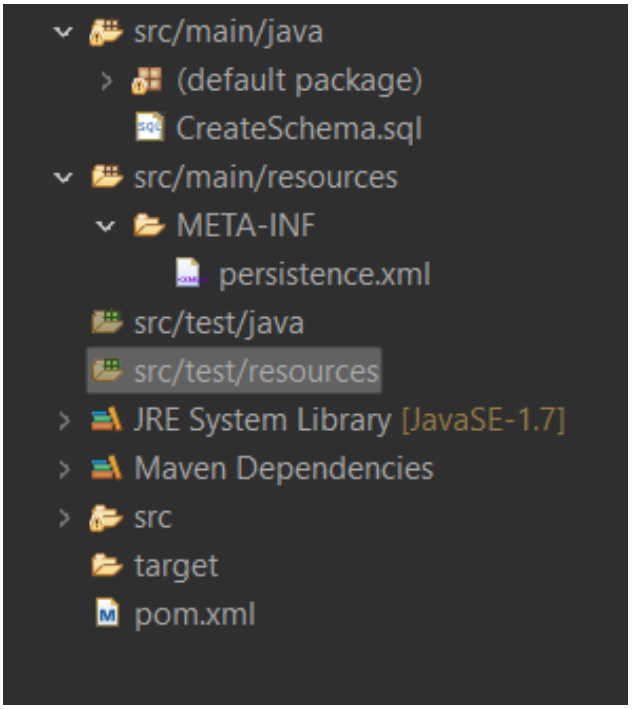
En conclusion, nous résumons les principaux points abordés et exprimons notre gratitude envers ceux qui ont contribué à ce rapport, tout en soulignant l'importance de choisir la bonne technologie de persistance des données en fonction des besoins spécifiques du projet.

01

Dans cette présentation, nous avons utilisé Eclipse IDE en conjonction avec des projets Maven pour développer des applications connectées à une base de données Oracle. La version de Java utilisée était Java 21, et nous avons intégré la base de données Oracle 23c dans nos projets pour assurer la compatibilité et la performance optimale.

L'ARCHITECTURE D'UN PROJECT MAVEN

Le projet Maven est structuré avec un répertoire **src/main/java** contenant le code source principal, organisé sous le package par default **(on peut crée un Package par nous même)**. Les ressources principales sont situées dans **src/main/resources/META-INF** avec le fichier de configuration **persistence.xml**. Le code des tests unitaires se trouve dans **src/test/java**. Le projet utilise Java SE 7 , et les bibliothèques référencées sont gérées via Maven avec le fichier de configuration **pom.xml**. Les fichiers compilés sont placés dans le répertoire **target**.



JDBC est une marque déposée de Sun/Oracle, souvent considéré comme étant l'acronyme de **Java DataBase Connectivity** et désigne une **API** de bas niveau de Java EE pour permettre un accès aux bases de données avec **Java**.

Elle permet de se connecter à une base de données et d'interagir avec notamment en exécutant des requêtes **SQL** pure .

L'architecture de **JDBC** permet d'utiliser la même **API** pour accéder aux différentes bases de données grâce à l'utilisation de **pilotes** (**drivers**) qui fournissent une implémentation spécifique à la base de données à utiliser.

Chaque base de données a alors la responsabilité de fournir un pilote qui assure la connexion entre **l'API** et les actions exécutées de manière propriétaire sur la base de données.

Quelles sont les raisons pour lesquelles l'utilisation de l'API JDBC a diminué ces derniers temps ?

*OracleCon.java

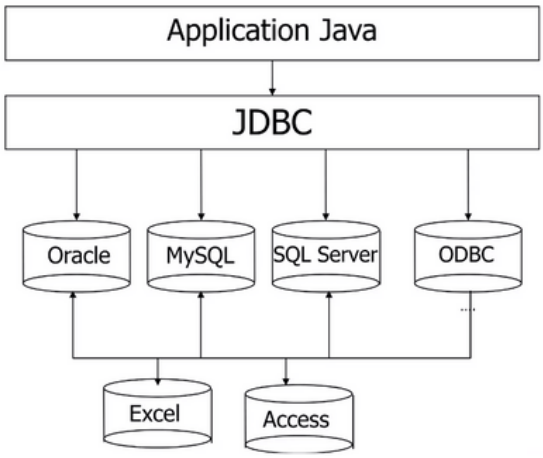
```java
package OracleCon;

import java.sql.*;
class OracleCon{
public static void main(String args[]){
try{
//step1 on recupere le pilote compatible avec Oracle
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 connexion avec la base de donnee
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");

//step3 executeur des queries
Statement stmt=con.createStatement();

//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");

while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object
con.close();

}catch(Exception e){ System.out.println(e);}

}
}
```

Exemple d'un code qui utilise JDBC pour se connecter a une base de données

Les limites de JDBC peuvent être résumées comme suit :

Complexité et verbosité : L'utilisation de JDBC nécessite souvent une quantité importante de code pour des opérations simples telles que l'établissement de la connexion à la base de données ou la manipulation des résultats de requêtes.

Gestion manuelle des transactions : Les transactions doivent être gérées manuellement avec JDBC, ce qui peut conduire à des erreurs de programmation et à des problèmes de cohérence des données si elles ne sont pas correctement gérées.



Absence de mapping objet-relationnel : JDBC ne fournit pas de mécanisme natif pour faire la correspondance entre les Objets Java et Les tables du base donnée , obligeant les développeurs à écrire du code supplémentaire pour transformer les résultats de requêtes SQL en objets Java.

Manque de portabilité : JDBC peut manquer de portabilité entre les différents systèmes de gestion de base de données, ce qui nécessite souvent des ajustements significatifs pour fonctionner avec des bases de données différentes.

En outre, une automatisation considérable de ce processus est désormais possible grâce à des concepts que nous aborderons prochainement, tels que la **persistance des données**, **JPA** (**Java Persistence API**), **Hibernate**, et enfin **Spring Data .**

DEFINITIONS

La Persistance des données est la notion qui traite de la sauvagarde des données contenues dans les objets des applications dans un support informatique comme les fichiers , les bases de données

Lors de lancement d'une application ce dernier créer en mémoire tous les objets dont elle a besoin mais une fois qu'on arrête l'application alors tous les données contenus dans les objets seront perdues.Pour éviter ca La persistence va nous permettre de sauvegarder l'état d'un objet à un instant quelconque sur un support informatique.
Les supports informatiques sont entre autres : Les fichiers , les Bases de Données Relatonnelles comme Oracle Database, Mysql, PostegreSQL ; Les Bases de Données non Relationnelles MongoDB, Cassandra,ou tout autre dispositif de stockage .Dans ce cours on va plus mettre l'accent sur les bases de données relationnelles .
Toutefois cette liaison entre le monde Relationnelle et le monde Objet pour le developpement des applications nécessite des techniques : **ORM**

Le Mapping Objet/Relationnel (Oject-Relational Mapping) ou **ORM** pour les intimes consiste à réaliser la correspondance entre une application modélisée en conception orienté objet et une base de données relationnelle

Les ORM sont des frameworks qui ont pour objectif principale L'ORM a pour but d'établir la correspondance entre : une table d'une base de donnée et une classe du modèle objet

LES FONCTIONNALITÉS DES ORM

Un ORM fournit généralement les fonctionnalités suivantes :

- 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
- Éventuellement permettre l'héritage des mappings.
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée.

- Supporter différentes formes d'identifiants générés automatiquement par les bases de données (identity, sequence, ...).
- prise en charge des dépendances entre objets pour la mise en jour en cascade de la base de données
- Support des transactions.
- Gestion des accès concurrents (verrou, deadlock, ...).
- Fonctionnalités pour améliorer les performances (cache, lazy loading, ...).

Toutes les implémentations des ORM en Java sont basées sur JDBC. Comme on l'a présenté précedemment JDBC est l'API utiliser au début pour faire une connexion entre les bases de données et les applications Java mais au prix de l'écriture d'un grand nombre de lignes de code (parfois complexes) et c'est la raison pour lesquelles les frameworks de ORM ont été crée.
Pour utiliser un ORM en Java, nous aurons simplement besoin de comprendre comment configurer une application utilisant JDBC

LES AVANTAGES DES ORM

Les ORM de part les fonctionnalités multiples offrent plusieurs avantages aux développeurs:

- Les ORM libérent le développeur de 95 pourcent des tâches de programmation liées à la persistance des données communes.
- Les ORM assurent la portabilité de votre application si vous changez de SGBD. En effet avec l'API JDBC on écrivait du code sql dans le code Java ceux qui n'assurait pas la portabilité de l'application si on change de SGBD compte tenue des dialectes
- Les ORM proposent au développeur des méthodes d'accès aux bases de données plus efficaces, ce qui devrait rassurer les développeurs on a pas besoin d'être un expert pour l'utilisation de ces outils ORM contrairement à JDBC il fallait savoir gérer les transactions le cache et beaucoup d'autre chose

LES FRAMEWORKS DES ORM

Différentes solutions sont apparues :

- TopLink est un framework d'ORM pour le développement Java. TopLink Essentials est la version open source du produit d'Oracle. Au mois de mars 2008, Sun Microsystems fait succéder EclipseLink à TopLink comme implémentation de référence de la JPA 2.0.
- EclipseLink est un framework open source de ORM pour les développeurs Java. Il fournit une plateforme puissante et flexible permettant de stocker des objets Java dans une base de données relationnelle et/ou de les convertir en documents XML. Il est développé par l'organisation Fondation Eclipse.
- Hibernate est un framework open source gérant la persistance des objets en base de données relationnelle, développé par l'entreprise JBoss.
- Apache OpenJPA est une implémentation open source de la JPA. Il s'agit d'une solution d'ORM pour le langage Java simplifiant le stockage des objets dans une base de données relationnelle. Il est distribué sous licence Apache.

Le problème qui se pose maintenant est pour chaque framework est associé le code Java propre à lui. Si on veut changer de frameorks pour une application donnée on sera dans l'obligation de changer le code source.Alors il se pose un problème de portabilité de notre application si on veut migrer d'un framework ORM vers un autre. Alors c'est dans cette problématique que SUN creér un API qui est souvent considérer comme une interface qui va nous permettre de standariser le **mapping objet relationnel : JPA**

JAVA PERSISTENCE API : JPA

DEFINITION

Java Persistence API (JPA) est un standard essentiel faisant partie intégrante de la plateforme Java EE. Il définit une spécification permettant la gestion de la correspondance entre les objets Java et une base de données, facilitant ainsi la gestion de la persistance des données

JPA propose un ensemble d'interfaces décrivant comment respecter ce standard. Cependant, pour mettre en œuvre JPA, il est nécessaire d'utiliser un framework ou une solution qui réalise cette implémentation. JPA fait partie intégrante de la plateforme JAVA JEE (JSR 220) et est défini dans le package javax.persistence.
Tous les frameworks qu'on a vu précedement implémente JPA. SUN quand il a construit la spécification a également crée le framework qui sert d'implémentation de référence : **eclipselink**

LES ENTITES JPA

JPA permet de définir des entités (entities). Une entité est simplement une instance d'une classe qui sera persistante (que l'on pourra sauvegarder dans / charger depuis une base de données relationnelle).Pour créer les entités on a deux méthodes principales :

- Par Fichiers XML de Configuration (ORM XML Mapping)
- Par les annotations JPA

Cependant, la plupart de développeurs préfèrent utiliser des annotations.

FICHIERS XML DE CONFIGURATION(ORM XML MAPPING)

Cette méthode était plus courante avant l'introduction des annotations en JPA, et elle reste utile pour certaines situations où les configurations doivent être séparées du code source, comme dans des environnements hérités ou lorsque des modifications dynamiques sont nécessaires sans recompilation du code. La qtructure générale de telles fichiers ressemble à ceci:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
    http://xmlns.jcp.org/xml/ns/persistence/orm_2_2.xsd"
    version="2.2">
    <!-- Déclaration de l'entité -->
    <entity class="chemin.vers.MaClasse">
        <!-- Définition de la table associée -->
        <table name="NOM_DE_LA_TABLE"/>
        <!-- Définition des attributs de l'entité -->
        <attributes>
            <!-- Attribut ID -->
            <id name="id">
                <generated-value strategy="IDENTITY"/>
            </id>
            <!-- Attribut simple -->
            <basic name="nomAttribut">
                <column name="NOM_COLONNE"/>
            </basic>
            <!-- Attribut Many-to-One -->
            <many-to-one name="nomAttributManyToOne" fetch="LAZY">
                <join-column name="NOM_COLONNE_JOINTURE"/>
            </many-to-one>
            <!-- Autres attributs... -->
        </attributes>
    </entity>
</entity-mappings>
```

La liste ci-dessous résume les propriétés utilisés en générale dans ces fichiers les plus simples et les plus utiles pour définir une entité et ses attributs :

<entity-mappings> :

- Description : L'élément racine qui encapsule les configurations de mapping pour les entités JPA. Il définit l'espace de noms et la version du schéma XML utilisé pour le mapping.
- xmlns : L'espace de noms XML utilisé pour les mappings JPA.
- xmlns:xsi : L'espace de noms XML Schema Instance.
- xsi:schemaLocation : L'emplacement du schéma XML pour la validation du document.
- version : La version du schéma JPA utilisée.

<Entity> : Définit une entité JPA et spécifie la classe Java correspondante.

- Attributs :
  - class : Le nom complet de la classe Java représentant l'entité (par exemple, com.example.Employee).

<Table> : Spécifie le nom de la table de base de données à laquelle l'entité est mappée.
<attributes> : Contient les mappings des attributs de l'entité, y compris les champs d'identifiant, les champs de base et les relations.
<Id> : Spécifie le champ de clé primaire de l'entité.
<GeneratedValue> : Spécifie comment la valeur de la clé primaire doit être générée.
<Column> : Spécifie le mapping du champ de l'entité vers une colonne de base de données.
<JoinColumn> : Spécifie le mapping d'une colonne de clé étrangère entre deux entités.
<NamedQuery> : Définit une requête nommée pour une entité.
<NamedQueries> : Définit un ensemble de requêtes nommées pour une entité.
<Transient>: Spécifie qu'un champ d'entité ne doit pas être persisté dans la base de données.
<Version> : Spécifie le champ de version de l'entité, utilisé pour le verrouillage optimiste.
<Temporal >: Spécifie le type temporel d'un champ d'entité date ou horodatage.

Il faut définir un fichier de mapping pour chaque classe nommé du nom de classe suivi de hbm.xml

ce fichier doit être situé dans le même repertoire classe que la classe correspondante ou dans le même archive pour les applications packagées

EXEMPLE :

```java
package entity;

public class Personne {
    private int id;
    private String nom;
    private String prenom;
    private int age;
    public Personne() {}
// les getters et les setters
}
```

Table Personne			
	Name	Type	
1	Id_Person	Int(11)	Primary Key
2	person_nom	varchar(45)	
3	person_prenom	varchar(45)	
4	person_age	Int(11)	

Le mapping de la classe Personne en table Personne avec le fichier personne.hbm.xml

personne.hbm.xml

Mapping par fichier XML

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 24 d?c. 2018 00:19:34 by Hibernate Tools 3.5.0.Final -->
<hibernate-mapping>
 <class name="ma.boukouchi.tp.entity.Personne" table="PERSONNE">
        <id name="id" type="int">
            <column name="id_Person" />
            <generator class="assigned" />
        </id>
        <property name="nom" type="java.lang.String">
            <column name="person_nom" />
        </property>
        <property name="prenom" type="java.lang.String">
            <column name="person_prenom" />
        </property>
        <property name="age" type="int">
            <column name="person_age" />
        </property>
 </class>
</hibernate-mapping>
```

Pour ce qui concerne la méthode d'annotations on le verra en détail dans la section suivante

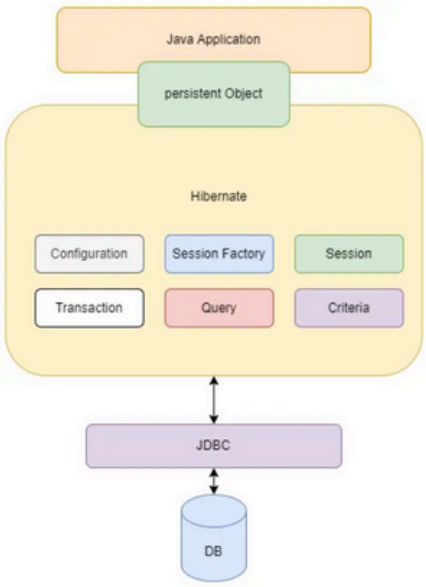
LES INTERFACES DE JPA

Pour gérer les entités JPA ainsi crée JPA possède d'un ensemble d'interface à cet effet. Parmis celles-ci trois sont principales:

- **EntityManage**r : Interface principale pour interagir avec le contexte de persistance. Elle permet de créer, lire, mettre à jour et supprimer des entités. L'EntityManager gère également les transactions et les requêtes JPQL.
- **EntityManagerFactory** : Interface pour créer des instances d'EntityManager. Elle est responsable de la configuration et de la gestion de l'usine d'EntityManager. En général, une instance d'EntityManagerFactory est créée par application.
- **EntityTransaction** :Interface pour gérer les transactions de l'EntityManager dans un environnement de gestion de transactions local. Elle permet de commencer, valider et annuler des transactions.

- Chaque framework ORM possède sa propre implémentation de l'interface **EntityManager**.
- **EntityManage**r définit les méthodes qui permettent de gérer le cycle de vie de la persistance des Entity.
  - La méthode **persist()** permet de rendre une nouvelle instance d'un EJB Entity persistante. Ce qui permet de sauvegarder son état dans la base de données.
  - La méthode **find()** permet de charger une entité sachant sa clé primaire.
  - La méthode **createQuery()** permet de créer une requête EJBQL qui permet de charger une liste d'entités selon des critères.
  - La méthode **remove()** permet de programmer une entité persistante pour la suppression.
  - La méthode **merge()** permet de rendre une entité détachée persistante.

On verra également comment les utiliser en détail dans la section suivante avec des exemples.



DEFINITION

Hibernate est une implémentation populaire de **JPA** qui offre des fonctionnalités avancées telles que la gestion des transactions, la mise en cache et la récupération paresseuse. Il facilite le développement d'applications basées sur des objets en offrant une interface de programmation plus simple et plus expressive pour interagir avec la base de données.

LE MAPPING AVEC LES ANNOTATION JPA

Le mapping avec les **annotations JPA** offre une approche efficace pour définir la correspondance entre les **entités Java** et les **tables de la base de données**. En utilisant les annotations fournies par la spécification JPA, les développeurs peuvent déclarer les entités, leurs attributs, les relations entre les entités, ainsi que les contraintes de base de données telles que les clés primaires et étrangères et les relations entres les entités .

Cette méthode permet de décrire la structure de la **base de données** directement dans le **code source Java**, ce qui améliore la lisibilité et facilite la maintenance. Elle est largement préférée et couramment utilisée par rapport à d'autres approches, offrant ainsi une manière plus concise et intuitive de gérer la correspondance entre les **entités Java** et les tables de la base de données.

Voici 15 annotations JPA populaires :

@Entity : Marque une classe Java comme une entité JPA, pouvant être persistée dans une base de données.
@Table : Spécifie le nom de la table de base de données à laquelle l'entité est mappée.
@Id : Spécifie le champ de clé primaire de l'entité.
@GeneratedValue : Spécifie comment la valeur de la clé primaire doit être générée.
@Column : Spécifie le mapping du champ de l'entité vers une colonne de base de données.
@JoinColumn : Spécifie le mapping d'une colonne de clé étrangère entre deux entités.
@NamedQuery : Définit une requête nommée pour une entité.
@NamedQueries : Définit un ensemble de requêtes nommées pour une entité.
@Transient : Spécifie qu'un champ d'entité ne doit pas être persisté dans la base de données.
@Version : Spécifie le champ de version de l'entité, utilisé pour le verrouillage optimiste.
@Temporal : Spécifie le type temporel d'un champ d'entité date ou horodatage.

Ce ne sont que quelques-unes des nombreuses annotations JPA disponibles. Selon votre cas d'utilisation, vous pourriez avoir besoin d'utiliser des annotations supplémentaires.

Voici quelques annotations Hibernate** qui ne font pas partie de la spécification JPA :

@Proxy(lazy = false) : Spécifie qu'une entité ne doit pas être chargée de manière paresseuse.
@Type : Spécifie le type d'un type personnalisé spécifique à Hibernate pour un champ d'entité.
@Fetch : Spécifie le mode de récupération pour une entité ou une collection liée.
@BatchSize : Spécifie la taille du lot pour la récupération des entités ou collections liées.
@SQLInsert, @SQLUpdate, @SQLDelete : Spécifie les instructions SQL pour l'insertion, la mise à jour ou la suppression d'une entité.
@Formula : Spécifie une propriété dérivée d'une entité à l'aide d'une expression SQL.
@DynamicUpdate, @DynamicInsert : Spécifie que seules les propriétés modifiées doivent être incluses dans l'instruction SQL de mise à jour ou d'insertion.

****l'utilisation d'annotations spécifiques à Hibernate peut rendre le code moins portable, car il est lié à l'implémentation Hibernate. Pour maximiser la portabilité, il est recommandé d'utiliser uniquement les annotations JPA et d'éviter d'utiliser les annotations spécifiques à Hibernate sauf si nécessaire.**

Configuration de HIBERNATE

- **Ajout les dépendances Hibernate et JPA :** L'ajout des dépendances Hibernate , JPA et Base de données utilisé dans le fichier **pom.xml** (pour pour le projet Maven).

- **Définition les entités** : Création des classes **Java Bean** pour représenter les entités de votre modèle de données. Utilisez les annotations JPA telles que **@Entity, @Id**, @GeneratedValue, @Column, etc., pour mapper ces classes aux tables de la base de données.

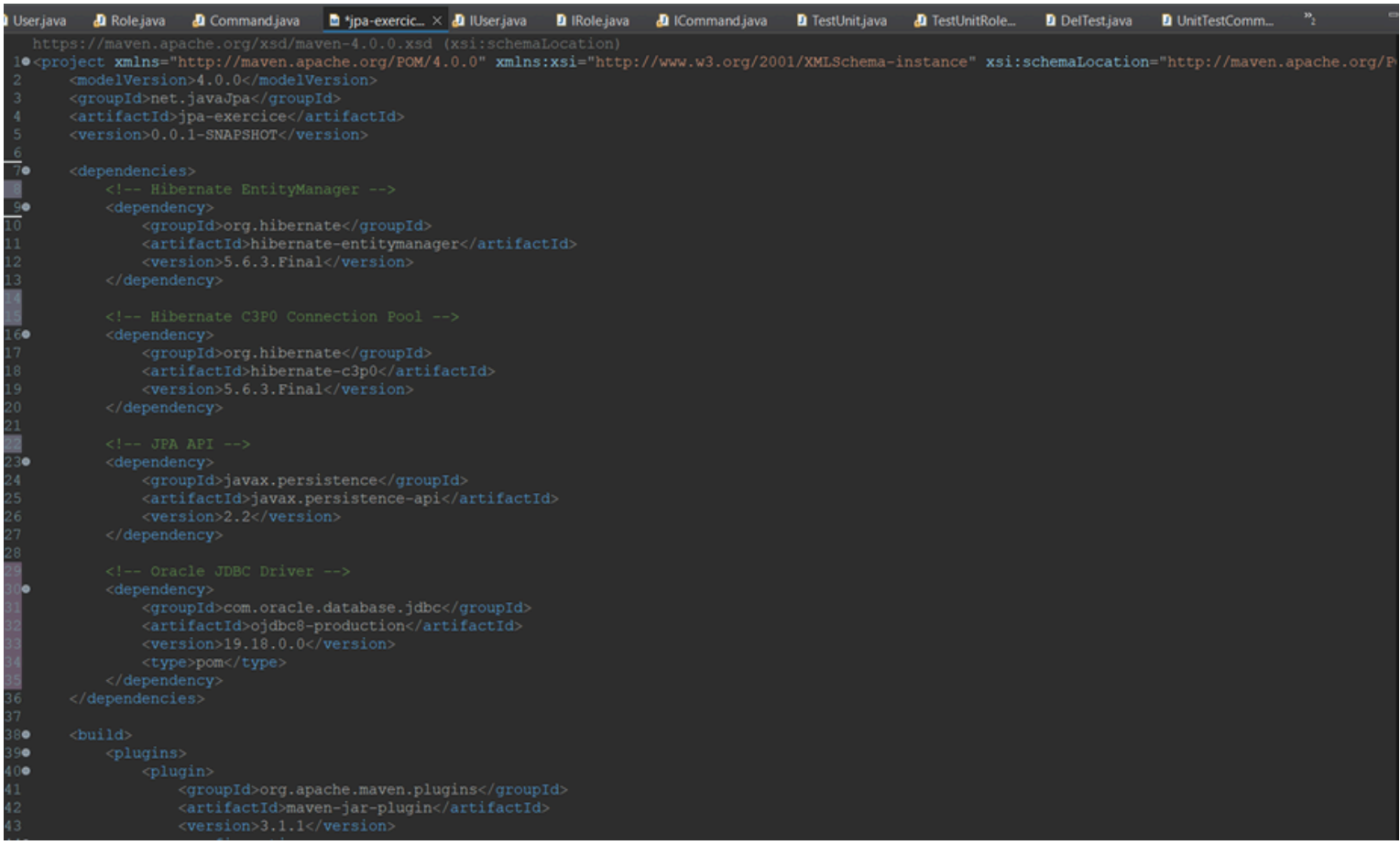
- **Configuration de fichier persistence.xml :** Création et la configuration du fichier persistence.xml dans le répertoire **META-INF** de votre dossier de ressources **(src/main/resources).** Ce fichier contiendra les informations de configuration de l'unité de persistance **JPA**, telles que le nom de l'unité, le fournisseur de persistance (Hibernate dans ce cas), et les propriétés de connexion à la base de données.

- **Écriture des requêtes JPA :** L'emploi des méthodes de **l'EntityManager** (ou Session dans le cas de **Hibernate** pur) pour exécuter des opérations de persistance telles que l'insertion, la mise à jour, la suppression et la récupération des entités **(Operation CRUD)**.

1 ERE ETAPE : AJOUT LES DÉPENDANCES

- **Qu'est-ce que le fichier pom.xml ?**

Le fichier **pom.xml** est un fichier de configuration utilisé par Maven pour gérer un projet Java. Il spécifie les dépendances, les plugins et d'autres configurations nécessaires à la construction et à l'exécution du projet.



- **Dans ce fichier pom.xml :**

- La dépendance **hibernate-entitymanager** intègre Hibernate avec JPA.
- La dépendance **hibernate-c3p0** ajoute un pool de connexions pour gérer les connexions à la base de données de manière efficace.
- La dépendance **javax.persistence-api** inclut l'API JPA pour les annotations et les interfaces nécessaires.
- La dépendance **ojdbc8-production** fournit le pilote JDBC pour se connecter à une base de données Oracle.

2 EME ETAPE : DÉFINITION DES ENTITÉS

Création des classes **JavaBean** pour représenter les entités de votre modèle de données : **Les entités JPA** sont des classes Java **(qui respect les critères du java bean)** qui représentent les tables dans une base de données relationnelle. Pour créer ces entités, nous utilisons des **JavaBeans** et les annotons avec des annotations JPA pour spécifier la manière dont elles doivent être mappées aux tables de la base de données.

Qu'est-ce qu'un JavaBean ?

Un **JavaBean** est une classe Java qui suit certaines conventions spécifiques :

- **Constructeur sans argument :** La classe doit avoir un constructeur public sans argument.
- **Propriétés** : Les attributs de la classe doivent être privés et accessibles via des méthodes publiques getter et setter.
- **Sérialisabilité (optionnel mais recommandé)**: La classe doit implémenter l'interface **java.io.Serializable** ou la **sérialisation** c'est la conversion d'un objet java en un flux de bytes pour stockage ou transmission.

Critères et annotations obligatoires

Pour qu'une classe **JavaBean** soit considérée comme une **entité JPA** valide, les critères suivants doivent être respectés :

- **@Entity :** Obligatoire pour indiquer que la classe est une entité.
- **@Id :** Obligatoire pour définir le champ qui sert de clé primaire.
- Les autres annotations comme **@Table, @GeneratedValue, et @Column** sont facultatives mais souvent nécessaires pour des configurations spécifiques et pour un mapping précis entre les entités et les tables de la base de données.

3 EME ETAPE : CONFIGURATION DU PERSISTENCE.XML

Le fichier **persistence.xml** est utilisé pour configurer **l'unité de persistance JPA** dans une application Java , on peut configurer plusieurs unité de persistance pour la configuration de différents base de données .
Voici une explication détaillée basée sur votre exemple :

STRUCTURE XML DE BASE:

```
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd (xsi:schemaLocation)
<?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">
```

DÉFINITION DE L'UNITÉ DE PERSISTANCE:

- L'élément **<persistence-unit>** contient les informations spécifiques à l'unité de persistance, telles que l'**ORM** utilisé et pour mettre cette unité de persistance unique on doit ajouter l'attribue **name** .

```
<persistence-unit name="WebStoreOrc">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
```

PROPRIÉTÉS DE CONNEXION À LA BASE DE DONNÉES:

- L'élément **<properties>** contient les propriétés de connexion à la base de données, telles que le pilote **JDBC**, **l'URL** de connexion, le **nom d'utilisateur** et le **mot de passe**.

```
    <properties>
        <property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver" />
        <property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@localhost:1521/FREE" />
        <property name="javax.persistence.jdbc.user" value="SYSTEM" />
        <property name="javax.persistence.jdbc.password" value="oracle" />

        <property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect" />
        <property name="hibernate.format_sql" value="true" />
    </properties>
```

- **<property name="javax.persistence.jdbc.driver" value="oracle.jdbc.OracleDriver" /> :** Cette ligne spécifie le pilote **JDBC** à utiliser pour la connexion à la base de données Oracle. Dans notre cas , il s'agit du pilote **Oracle JDBC**.

- **<property name="javax.persistence.jdbc.url" value="jdbc:oracle:thin:@localhost:1521/FREE" />** : Cette ligne spécifie l'**URL de connexion** à la base de données. Dans notre cas , il s'agit d'une connexion locale à Oracle sur le **port 1521** avec la **base de donnée FREE**.

- **<property name="javax.persistence.jdbc.user" value="SYSTEM" />** : Cette ligne spécifie le **nom d'utilisateu**r pour la connexion à la base de données.

- **<property name="javax.persistence.jdbc.password" value="oracle" />** : Cette ligne spécifie **le mot de passe** pour la connexion à la base de données.

- **<property name="hibernate.dialect" value="org.hibernate.dialect.OracleDialect" /> :** Cette ligne spécifie le **dialecte Hibernate** à utiliser pour la génération de requêtes SQL spécifiques à Oracle. Dans notre cas, le dialecte utilisé est **OracleDialect**.

- **<property name="hibernate.format_sql" value="true" />** (**optionnel**): Cette ligne spécifie si Hibernate doit formater les requêtes SQL générées. Lorsque cette propriété est définie sur true, les requêtes SQL sont joliment formatées pour une meilleure lisibilité dans les journaux de débogage.

- **<property name="hibernate.hbm2ddl.auto" value="update"/> :** permet à Hibernate de mettre à jour automatiquement **(sans écraser les donnes contrairement a l'autre option "create")** le schéma de la base de données en fonction des modifications apportées aux entités JPA

4 EME ETAPE : ÉCRITURE DES REQUÊTES JPA

L'écriture des requêtes JPA implique l'utilisation des méthodes fournies par **l'EntityManager** pour effectuer des opérations de persistance, notamment l'insertion, la mise à jour, la suppression et la récupération des entités, communément désignées par l'acronyme **CRUD (Create, Read, Update, Delete)**. Ces opérations permettent de manipuler les données stockées dans la base de données en utilisant les entités Java correspondantes.

- ****Insertion (Create)**** : Utilisation de la méthode **`persist()`** de l'EntityManager pour ajouter une nouvelle entité à la base de données.
- ****Récupération (Read)**** : Utilisation de méthodes telles que **`find()`** ou **`createQuery()`** pour récupérer des entités existantes à partir de la base de données en fonction de critères spécifiés.
- ****Mise à jour (Update)**** : Modification des attributs d'une entité puis appel de la méthode `merge()` de l'EntityManager pour mettre à jour cette entité dans la base de données.
- ****Suppression (Delete)**** : Utilisation de la méthode **`remove()`** de l'EntityManager pour supprimer une entité de la base de données.

L'utilisation de ces méthodes permet d'interagir avec la base de données de manière efficace et sécurisée, en garantissant la cohérence et l'intégrité des données. Il est également possible d'écrire des requêtes JPA plus complexes en utilisant le **langage JPQL (Java Persistence Query Language) / HQL (Hibernate Query Language)** pour effectuer des opérations de filtrage, de tri et de jointure sur les données. et Les transactions, utilisées pour encapsuler ces opérations, assurent la cohérence des données lors de leur exécution.

EXEMPLE DES REQUETES CRUD

```
public void addProduit(Produit p) {
/* Création d'une transaction */
EntityTransaction transaction=entityManager.getTransaction();
/* Démarrer la transaction */
transaction.begin();
try {

/* enregister le produit p dans la base de données */
entityManager.persist(p);
/* Valider la transaction si tout se passe bien  */

transaction.commit();
} catch (Exception e) {
/* Annuler la transaction en cas d'exception  */
transaction.rollback();
e.printStackTrace();
}
}
```

```
public List<Produit> listProduits() {
  Query query=entityManager.createQuery("select p from Produit p");
  return query.getResultList();
}
```

- cette requête est spécifiée en utilisant le langage de requêtes JPA appelé **HQL** ou **JPA QL.**
- **HQL** ressemble à SQL, sauf que au lieu de des tables et des relations entre es tables, on utilise les classes et les relations entre les classes.

En fait, avec JPA, quant on fait la programmation orientée objet, on n'est pas sensé connaitre la structure de la base de données, mais plutôt on connait le diagramme de classes des différentes entités.

- C'est Hibernate qui va traduire le HQL en SQL. Ceci peut garantir à notre application de fonctionner correctement quelque soit le type de SGBD utilisé

```
public Produit getProduit(Long idProduit) {
  Produit p=entityManager.find(Produit.class, idProduit);
  return p;
}
```

- Pour sélectionner un objet sachant son identifiant (clé primaire), on utilise la méthode **find()** de l'objet entityManager.
- Si l'objet n'existe pas, cette méthode retourne null.

```
public void updateProduit(Produit p) {
      entityManager.merge(p);
}
```

- Pour mettre à jour un objet édité et modifier, on peut utiliser la méthode **merge()** de entityManager.

```
public List<Produit> listProduits() {
  Query query=entityManager.createQuery("select p from Produit p");
  return query.getResultList();
}
```

cette requête est spécifiée en utilisant le langage de requêtes JPA appelé **HQL** ou **JPA QL.**

HQL ressemble à SQL, sauf que au lieu de des tables et des relations entre es tables, on utilise les classes et les relations entre les classes.

En fait, avec JPA, quant on fait la programmation orientée objet, on n'est pas sensé connaitre la structure de la base de données, mais plutôt on connait le diagramme de classes des différentes entités.

C'est Hibernate qui va traduire le HQL en SQL. Ceci peut garantir à notre application de fonctionner correctement quelque soit le type de SGBD utilisé

```
public Produit getProduit(Long idProduit) {
  Produit p=entityManager.find(Produit.class, idProduit);
  return p;
}
```

- Pour sélectionner un objet sachant son identifiant (clé primaire), on utilise la méthode **find()** de l'objet entityManager.
- Si l'objet n'existe pas, cette méthode retourne null.

```
public void updateProduit(Produit p) {
        entityManager.merge(p);
}
```

- Pour mettre à jour un objet édité et modifier, on peut utiliser la méthode **merge()** de entityManager.

LES RELATIONS AVEC JPA

L'une des fonctionnalités majeures des **ORM (Object-Relational Mapping)** est de **gérer** les relations entre objets comme des relations entre tables dans un modèle de base de données relationnelle. **JPA (Java Persistence API)** définit des modèles de relation qui peuvent être déclarés par annotations pour faciliter cette correspondance. Les relations sont spécifiées par les annotations suivantes : **@OneToOne, @ManyToOne, @OneToMany, @ManyToMany**.

LA RELATION 1:1 (ONE TO ONE)

L'annotation **@OneToOne** définit une relation **1:1** entre deux entités. Si cette relation n'est pas forcément très courante dans un modèle relationnel de base de données, elle se rencontre très souvent dans une modèle objet.

Exemple de relation OneToOne

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToOne
  @JoinColumn(name = "abonnement_fk")
  private Abonnement abonnement;

  // getters et setters omis ...
}
```

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Abonnement {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

L'annotation **@OneToOne** implique que la table Individu contient une colonne qui est une clé étrangère contenant la clé d'un abonnement. Par défaut, JPA s'attend à ce que cette colonne se nomme **ABONNEMENT_ID**, mais il est possible de changer ce nom grâce à l'annotation @JoinColumn :

LA RELATION N:1 (MANY TO ONE)

L'annotation **@ManyToOne** définit une relation **n:1** entre deux entités.

Exemple de relation ManyToOne

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToOne;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne
  // déclaration d'une table d'association
  @JoinTable(name = "societe_individu",
             joinColumns = @JoinColumn(name = "individu_id"),
             inverseJoinColumns = @JoinColumn(name = "societe_id"))
  private Societe societe;

  // getters et setters omis ...
}
```

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Societe {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

L'annotation **@ManyToOne** implique que la table Individu contient une colonne qui est une clé étrangère contenant la clé d'une société. Par défaut, JPA s'attend à ce que cette colonne se nomme **SOCIETE_ID**, mais il est possible de changer ce nom grâce à l'annotation @**JoinColumn**.

Plutôt que par une colonne, il est également possible d'indiquer à JPA qu'il doit passer par une table d'association pour établir la relation entre les deux entités avec l'annotation @**JoinTable .**

LA RELATION 1:N (ONE TO MANY)

L'annotation **@OneToMany** définit une relation **1:n** entre deux entités. Cette annotation ne peut être utilisée qu'avec une collection d'éléments **"ArrayList ,List , etc"** puisqu'elle implique qu'il peut y avoir plusieurs entités associées.

Exemple de relation OneToMany

```java
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Societe {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToMany
  private List<Individu> employes = new ArrayList<>();

  // getters et setters omis ...
}
```

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

L'annotation **@OneToMany** implique que la table Individu contient une colonne qui est une clé étrangère contenant la clé d'une société. Par défaut, JPA s'attend à ce que cette colonne se nomme **SOCIETE_ID**, mais il est possible de changer ce nom grâce à l'annotation **@JoinColumn**. Il est également possible d'indiquer à JPA qu'il doit passer par une table d'association pour établir la relation entre les deux entités avec l'annotation @**JoinTable**. Ainsi, **@ManyToOne** fonctionne exactement comme **@OneToMany** sauf que cette annotation porte sur l'autre côté de la relation.

LA RELATION N:N (MANY TO MANY)

L'annotation **@ManyToMany** définit une relation **n:n** entre deux entités. Cette annotation ne peut être utilisée qu'avec une collection d'éléments puisqu'elle implique qu'il peut y avoir plusieurs entités associées.

Exemple de relation ManyToMany (Relation ternaire)

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToMany
  private List<Individu> enfants = new ArrayList<>();

  // getters et setters omis ...
}
```

```java
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToMany
  @JoinTable(name = "ParentEnfant",
             joinColumns = @JoinColumn(name = "parent_id"),
             inverseJoinColumns = @JoinColumn(name = "enfant_id"))
  private List<Individu> enfants = new ArrayList<>();

  // getters et setters omis ...
}
```

L'annotation **@ManyToMany** implique qu'il existe une table d'association **Individu_Individu** qui contient les clés étrangères **INDIVIDU_ID** et **ENFANTS_ID** permettant de modéliser la relation. Il est possible de décrire explicitement la relation grâce à l'annotation **@JoinTable**

JPA AVEC SPRING DATA

Spring Data est un projet Spring qui a pour objectif de simplifier l'interaction avec différents systèmes de stockage de données : qu'il s'agisse d'une base de données relationnelle, d'une base de données NoSQL, d'un système Big Data ou encore d'une API Web.
Le principe de Spring Data est d'éviter aux développeurs de coder les accès à ces systèmes. Pour cela, Spring Data utilise une convention de nommage des méthodes d'accès pour exprimer la requête à réaliser.Spring Data fournit une abstraction au dessus des technologies de persistence de données , on aura plus besoin d'implementer les interfaces de JPA pour gérer les entités et aussi la persistence de ces entités

AJOUT SPRING DATA JPA DANS UN PROJET MAVEN

Spring Data se compose d'un noyau central et de plusieurs sous modules dédiés à un type de système de stockage et une technologies d'accès. Dans un projet Maven, pour utiliser Spring Data pour une base de données relationnelles avec JPA, il faut déclarer la dépendance suivante dans le **fichier pom.xml**

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>2.2.2.RELEASE</version>
</dependency>
```

Spring Data s'organise autour de la notion de repository. Il fournit une interface marqueur générique **Repository<T, ID>**. Le type T correspond au type de l'objet géré par le repository. Le type ID correspond au type de l'objet qui représente la clé d'un objet.
L'interface **CrudRepository<T, ID>** hérite de **Repository<T, ID>** et fournit un ensemble d'opérations élémentaires pour la manipulation des objets.
Pour une intégration de Spring Data avec JPA, il existe également l'interface **JpaRepository<T, ID>** qui hérite indirectement de **CrudRepository<T, ID>** et qui fournit un ensemble de méthodes pour interagir avec une base de données.

Un exemple d'implémentation de l'interface JPARepository

```java
package com.oracle.dev.jdbc.springboot3.jpa.ucp;

import java.util.List;

interface UserRepository extends JpaRepository<User, Integer> {
    List<User> findbyrole(Role roles);
    void addRoleToUser(int idUser,int idRole);
}
```

L'interface **JpaRepository<T, ID>** déclare beaucoup de méthodes mais elles suffisent rarement pour implémenter les fonctionnalités attendues d'une application. Spring Data utilise une convention de nom pour générer automatiquement le code sous-jacent et exécuter la requête. La requête est déduite de la signature de la méthode (on parle de query methods).
La convention est la suivante : Spring Data JPA supprime du début de la méthode les prefixes find, read, query, count and get et recherche la présence du mot By pour marquer le début des critères de filtre. Chaque critère doit correspondre à un paramètre de la méthode dans le même ordre.

```java
interface UserRepository extends JpaRepository<User, Integer> {
    List<User> findbyrole(Role roles);
    User getByLogin(String login);

    long countByEmail(String email);

    List<User> findByNameAndEmail(String name, String email);

    List<User> findByNameOrEmail(String name, String email);
}
```

Spring Data JPA générera automatiquement sans avoir besoin d'implémenter le corps des méthoses une implémentation pour chaque méthode de ce repository contrairement précedemment où on était obligé de définir une interface où on implémentera nos propres méthodes

IMPLÉMENTATION DES MÉTHODES DE REPOSITORY

Il est parfois nécessaire de fournir une implémentation d'une ou de plusieurs méthodes d'un repository. Dans ce cas, il faut isoler les méthodes que l'on souhaite implémenter dans une interface spécifique. Par exemple, on peut créer l'interface **UserCustomRepository :**

```java
package com.oracle.dev.jdbc.springboot3.jpa.ucp;

public interface UserCustomRepository {
    //déclaration des méthodes personalisés
    //AUutant que vous voulez
    void doSomethingComplicated(User u);

}
```

Comme Spring Data JPA détecte une interface parente qui n'hérite pas elle-même de l'interface **Repository<T, ID>**, il recherche une classe Java implémentant l'interface spécifique portant le même nom que l'interface avec le suffixe **Impl** dans le même package ou un sous-package. Si une telle classe existe alors Spring Data JPA tente de créer un bean de cette classe.

ATTENTION :
La classe d'implémentation ne doit pas porter de stéréotype Spring comme @Component ou @Repository. Par contre, elle peut utiliser toutes les autres annotations autorisées par le Spring Framework si le contexte d'application est configuré correctement.

```java
package com.oracle.dev.jdbc.springboot3.jpa.ucp;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;

public class UserCustomRepositoryImpl implements UserCustomRepository {
    @PersistenceContext
     private EntityManager em;

    @Override
     public void doSomethingComplicated(User u) {
       // ...implémentation de le methode
     }

}
```

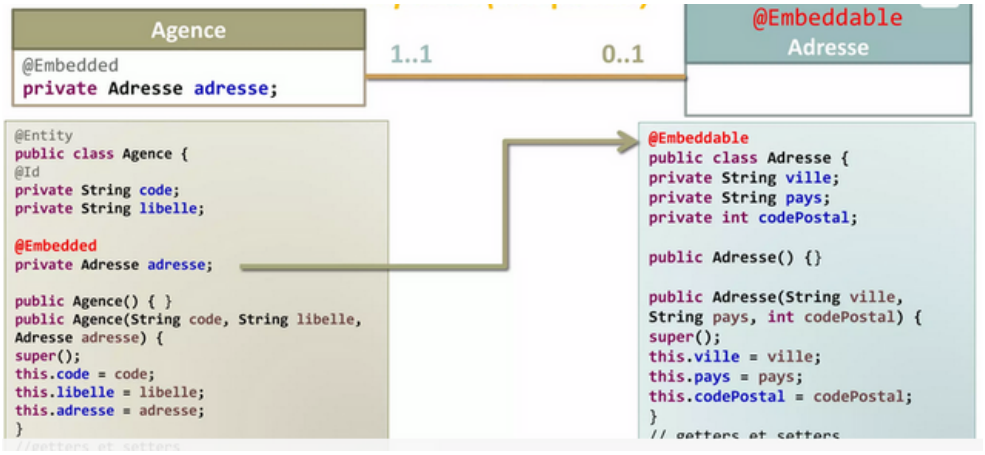
Le repository fonctionnera ainsi par délégation. Lorsque la méthode UserRepository.doSomethingComplicated sera appelée, elle déléguera le traitement à la méthode UserCustomRepositoryImpl.doSomethingComplicated.

Après modifier la classe **UserRepository** pour qu'il hérite aussi de l'interface Spécifique

```java
/*  Copyright (c) 2021, 2022, Oracle and/or its affiliates.

package com.oracle.dev.jdbc.springboot3.jpa.ucp;

import java.util.List;

interface UserRepository extends JpaRepository<User, Integer>,UserCustomRepository {
    List<User> findbyrole(Role roles);
    void addRoleToUser(int idUser,int idRole);
}
```

EXEMPLE DE MAPPING D'UNE RELATION ONETOONE AVEC HIBERNATE

L'application consiste à la gestion des agences d'une banque. On a deux classes : Agence et Addresse . Une addresse est associé à 0 ou une seule agence et une agence de son côté est associée à une et une seule addressese



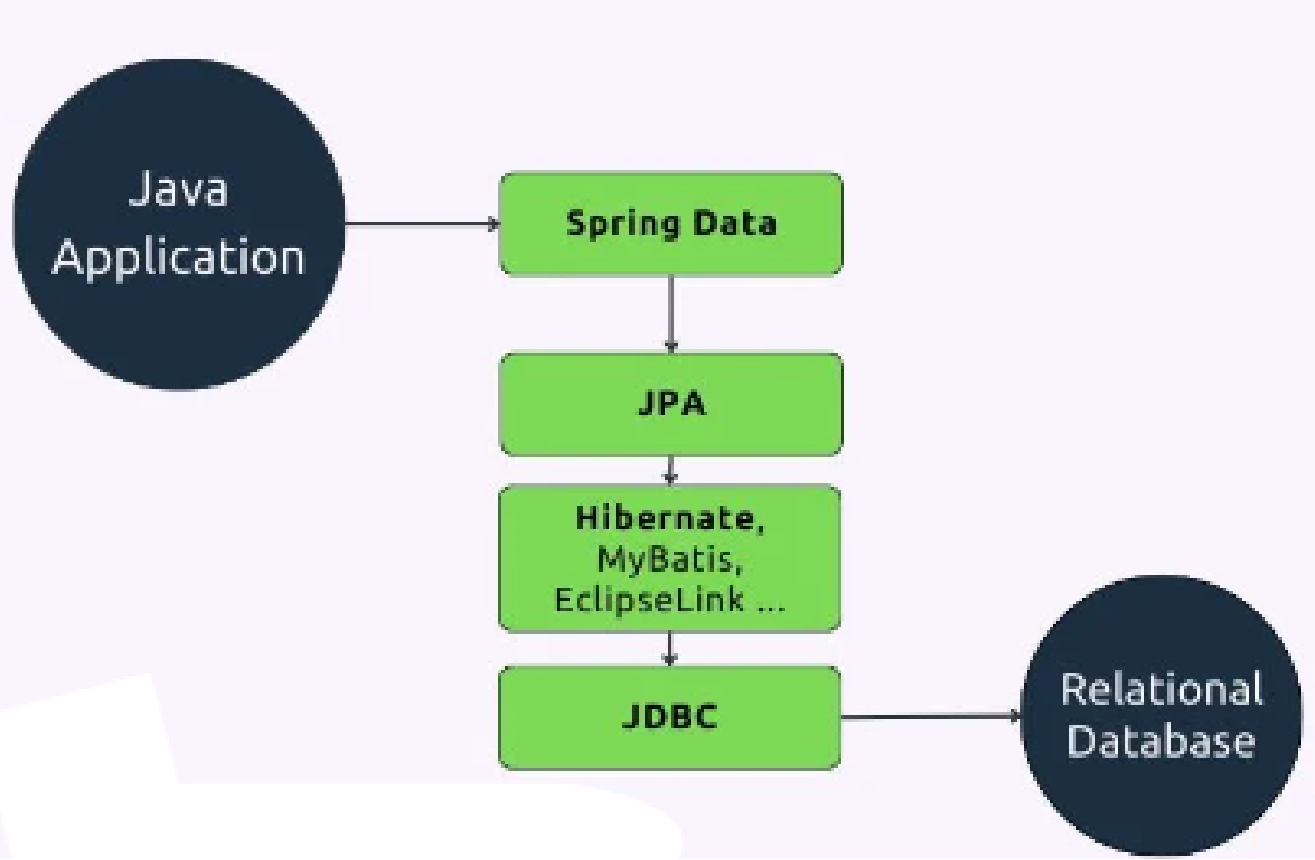
Les composants (component)

- Une entité existe par elle-même indépendamment de toute autre entité, et peut être rendue persistante par insertion dans la base de données, avec un identifiant propre.
- Un composant est un objet sans identifiant, qui ne peut être persistant que par rattachement à une entité.
- La notion de composant résulte du constat qu'une ligne dans une base de données peut parfois être décomposée en plusieurs sous-ensembles dotés chacun d'une logique autonome. Cette décomposition mène à une granularité fine de la représentation objet, dans laquelle on associe plusieurs objets à une ligne de la table.
  - **@Embedded** nous indique que ce composant est embarqué.
  - **@Embeddabl**e nous indique que cette classe sera utilisée comme composant.

CONCLUSION

Pour conclure ce sujet là , on a cette schéma qui illustre l'architecture typique d'une **application Java** pour la gestion des données, mettant en évidence l'intégration de divers **frameworks** et technologies pour faciliter l'accès aux **bases de données relationnelles**. Au cœur de cette architecture, une application Java utilise **Spring Data** pour simplifier l'interaction avec les services de stockage de données. Spring Data repose sur la spécification **JPA (Java Persistence API)**, qui standardise la gestion des entités et des relations dans une base de données. Les frameworks de persistance tels que **Hibernate**, **MyBatis**, et **EclipseLink** implémentent JPA, offrant des fonctionnalités avancées pour le **mapping objet-relationnel (ORM)**. Enfin, **JDBC (Java Database Connectivity)** assure la communication directe avec la base de données relationnelle.

Cette architecture modulaire permet aux développeurs de se concentrer sur la logique métier tout en bénéficiant d'une gestion **efficace** et **sécurisée** des données, améliorant ainsi la **maintenabilité**, la **réutilisabilité** et la **performance** des applications.



2024/2023

MERCI POUR VOTRE LECTURE