Synnaxium Studio

JPA - Hibernate

Introduction

Qu’est ce que JPA ? (Java Persistence API)

JPA -> ORM

… Qu’est ce qu’un ORM ?

Object-Relational Mapping

Permet de faire la liaison entre notre Base de Donnée Relationnelle et notre code qui est Orienté Objet

Introduction

D’accord mais c’est pas déjà ce qu’on faisait ?

Oui, l’avantage de la plupart des ORM est de permettre l’automatisation des tâches de lecture / écriture de la BDD

Remplacer les SELECT * . . .

Par des new Actor()... de manière rapide et efficaces

Rappel / Introduction au modèle MVC

Les ORM sont utilisés dans beaucoup de types d'applications mais principalement dans les applications dites MVC

MVC -> Modèle - Vue - Contrôleur

Le principe est de séparer la logique dite "métier" de l'affichage du logiciel.

Cette séparation permet d'améliorer la maintenance du code (on séparer la récupération et le traitement des information et l'affichage pur et dur)

Les ORM viennent se placer dans la partie "Modèle" de l'application

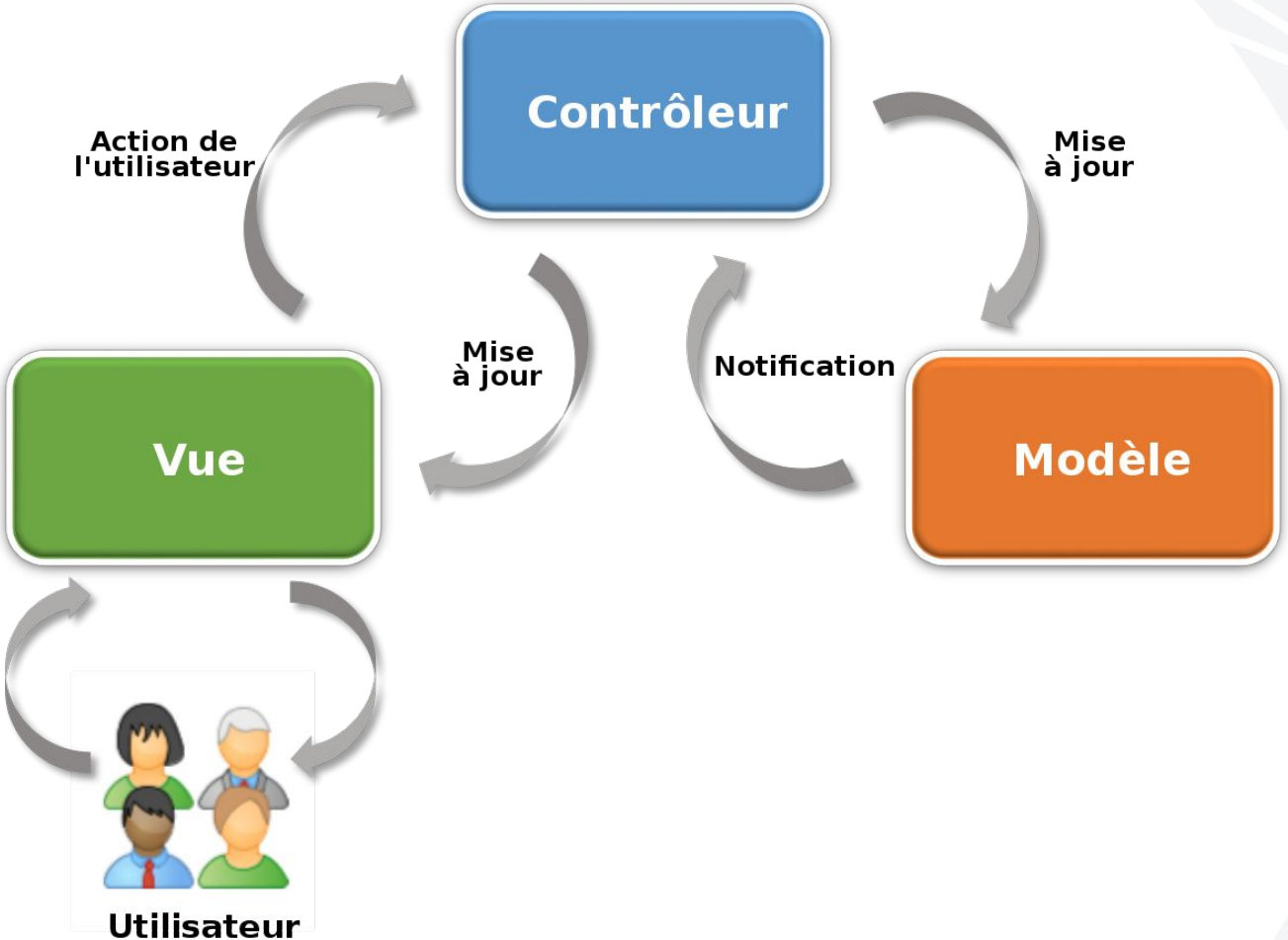
Rappel / Introduction au modèle MVC

Comment ça se présente ?

Modèle : contient les partie stockage des données

Vue : contient la partie graphique de l'application (affichage des données)

Contrôleur : contient la logique permettant de traiter les interactions de l'utilisateur.


Contrôleur
Action de l'utilisateur
Mise à jour
Mise à jour
Notification
Vue
Modèle
Utilisateur

Autres avantages

Permet d’isoler la couche “données”

- On peut la tester indépendamment du reste de l’application
- On peut changer de SGBD sans se soucier du reste de l’application
- Les modèles peuvent évoluer sans perturber le reste de l’application

Hibernate ?

Implémentation de JPA, version actuelle 6.1

D’autres implémentation de JPA sont disponibles :

- EclipseLink
- OpenJPA
- SpringJDBCTemplate

Nous allons donc devoir utiliser deux librairies en parallèles : Hibernate ET JPA

Mise en place de l’environnement

Pour le reste des TPs nous allons voir comment se servir de JPA grâce à la base de données de Test utilisée dans nos précédents TPs (si c’est moi qui ai fait les modules précédents ;))

Setup de l’environnement :

Java 18
Hibernate
Mysql Driver

Base de données : sakila

C’est parti !

Ouvrir le build.gradle et ajouter les lignes nécessaires pour nos dépendances.

```
// On ajoute la dépendance Hibernate
implementation 'org.hibernate:hibernate-core:6.1.2.Final'
// Le connecteur MySQL
implementation 'mysql:mysql-connector-java:8.0.30'
```

Ensuite pour VSCode on Reload le Projet et on Refresh le Workspace (Ctrl+Shift+P)

On est prêts !

Avec Java 18 l'API de JPA est comprise dans le package jakarta
jakarta.persistence

C'est quoi Jakarta ?

Anciennement appelé Java 2 Platform Enterprise Edition (J2EE)
Cela regroupe toutes les librairies pour étendre JSE pour le lancement d'application plus complètes, cela inclus des services web, les orm etc

https://jakarta.ee/specifications/platform/9/apidocs/

Pour d'anciennes version, les packages se trouvent dans javax.persistence

Avant toute chose !

JPA / Hibernate est un sujet vaste, il y a beaucoup de subtilités et d'options disponibles.

Toutes les possibilités ne sont pas abordées dans ce cours, voici quelques ressources pour aller plus loin :
https://www.baeldung.com/learn-jpa-hibernate

Pour Spring + JPA / Hibernate :

- https://www.baeldung.com/the-persistence-layer-with-spring-and-jpa
- http://orm.bdpedia.fr/intro.html

Avant toute chose !

Et bien sûr !

La documentation ;)

https://hibernate.org/orm/documentation/6.1/

C’est parti !

Configuration de JPA

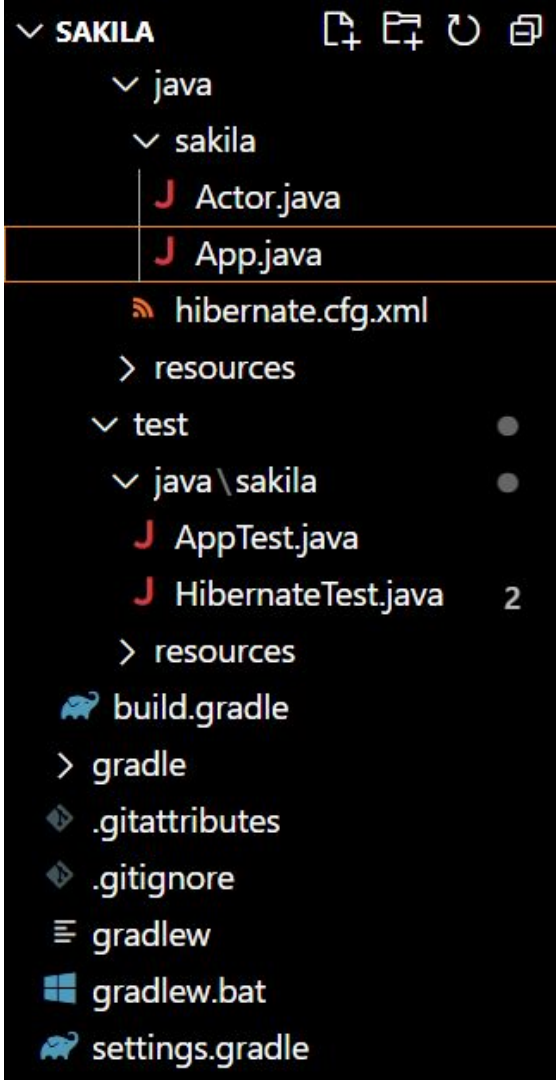
Une fois notre environnement prêt, nous pouvons établir notre première connexion à la base de données.

Pour cela nous allons créer notre fichier de configuration :

- hibernate.cfg.xml

Il devra se trouver à la racine de notre classpath.

Pour mon projet par exemple :



Bases de la configuration

```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>
        <!-- local connection properties -->
        <property name="hibernate.connection.url">jdbc:mysql://mysql.snx.ovh:3306/sakila</property>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.username">M2I</property>
        <property name="hibernate.connection.password">H3ll0M2I</property>
        <property name="hibernate.connection.pool_size">10</property>

        <!-- dialect for MySQL -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <property name="hibernate.show_sql">true</property>
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <property name="cache.use_query_cache">false</property>
    </session-factory>
</hibernate-configuration>
```

Bases de la configuration

On retrouve les principales lignes nécessaires pour la configuration d'une connexion.

D'abord la balise "session-factory" qui permet de définir une connexion à une base de données. Il peut y en avoir plusieurs dans un même fichier de configuration si plusieurs bases de données doivent être connectées.

A l'intérieur de cette balise nous allons trouver plusieurs balises "property" qui permettent de configurer des éléments spécifiques à notre connexion

On retrouvera : le driver utilisé, les identifiants et le "langage" à utiliser

A vous de préparer votre config !

Testons notre connexion !

Pour tester que nous arrivons à nous connecter, nous pouvons écrire un test qui viendra vérifier que la connexion se créée correctement.

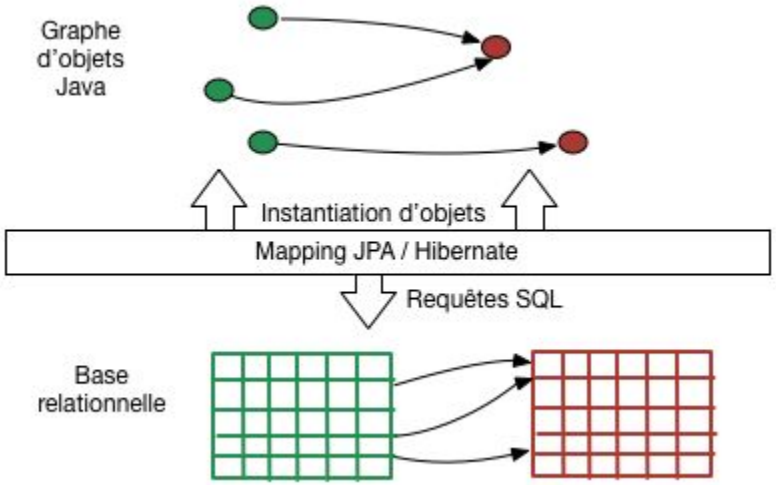
Ici le test sera “fail” si la connexion n’est pas établie

```
@Test void TestsHibernate() {
    // Création d'une connexion vers notre SGBD
    Configuration configuration = new Configuration().configure();
    SessionFactory sessionFactory = configuration.buildSessionFactory();
    Session session = sessionFactory.openSession();
    assertNotNull(session);
}
```

Faire sa première Entity !

Comme vu précédemment, JPA va nous permettre de créer des objets à partir de nos éléments de base de données.

Pour cela nous allons devoir créer nos classes et leur appliquer des tags afin de les lier à notre Base de Données. Afin de créer un “graph” de classes



Exemple d’Entity Simple

Plutôt que de blablater pendant des heures je vous propose d’analyser un peu une Entity simple que j’ai créé

Table utilisée : country

```java
package sakila;

import java.util.Date;

import jakarta.persistence.*;

@Entity
@Table(name="country")
public class Country {
    @Id
    @Column(name = "country_id")
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer countryId;

    @Column(name="country")
    private String country;

    @Column(name="last_update")
    @Temporal(TemporalType.TIMESTAMP)
    private Date lastUpdate;

    //#region get/set···
}
```

Principes de base

Utiliser l'encapsulation : IMPORTANT

Utilisation de différents tags pour nos champs / noms de table etc

Génération des get / set (forcément si on utilise l'encapsulation)

Importer les éléments de **jakarta.persistence**

(Ici j'ai tout importé pour que cela soit plus lisible, mais vous pouvez importer uniquement ce qui vous intéresse)

Testons notre Entity !

```java
public static void country(){
    Session session;

    Configuration configuration = new Configuration().configure();

    // IMPORTANT : On Ajoute bien nos Entités !!
    configuration.addAnnotatedClass(annotatedClass: Country.class);
    // Fin Ajout entités

    SessionFactory sessionFactory = configuration.buildSessionFactory();
    session = sessionFactory.openSession();

    Country a1 = session.getReference(entityType: Country.class, id: 34);

    System.out.println(a1.getCountry());
    session.close();


}
```

Comparons un peu avec l'ancien code

Qu'est ce qui a changé par rapport à l'établissement de la connexion ?

```
configuration.addAnnotatedClass(annotatedClass: Country.class);
```

```
Country a1 = session.getReference(entityType: Country.class, id: 3);

System.out.println(a1.getCountry());
session.close();
```

Que fait chaque ligne ?

Enregistrement de la classe

Hibernate ne connaît le mapping entre la classe et la table que s'il lui est fourni explicitement.

Pour cela deux moyens existent :

Déclarer directement lors de la programmation

```
configuration.addAnnotatedClass(annotatedClass: Country.class);
```

Utiliser le fichier de configuration précédemment créé et ajouter :

```
        <mapping class="package.Class" />
</session-factory>
```

Il faudra bien sûr remplacer package.Class, par le nom de votre package…et de votre classe…

Récupération d’un contenu

```
Country a1 = session.getReference(entityType: Country.class, id: 3);
```

Cette ligne permet de récupérer du contenu, cela va aller chercher l'élément possédant l’id 3 et le “binder” a sa classe.

On peut ensuite se servir de l’objet contenu dans a1 comme un objet classique

Mapping

Nous venons de voir un mapping basique et une lecture de données basique

Voyons un peu ce qui s’est passé en détail :

@Entity : Indique que les instances de cette classe sont persistantes
@ID : Indique que la propriété est une clé primaire
@Column : Indique que la propriété est associé à une colonne de la base
@Table : Indique la table correspondant à la classe

Ça c'est la base ! Bien sûr vous pouvez remarquer que j’ai ajouté quelques petites annotations en plus. Elles permettent de réaliser des opérations un peu plus avancées.

Mapping : partie 2

Dans notre exemple il y a quelques propriétés supplémentaires :

@GeneratedValue : Qui permet de définir la manière dont est généré la clé primaire. Elle se fera selon le GenerationType précisé :

- *GenerationType.AUTO :* La génération de la clé primaire est laissée à l'implémentation. C'est Hibernate qui s'en occupe
- *GenerationType.IDENTITY :* La génération se fait par le SGDB (Mysql : AUTO_INCREMENT)
- *GenerationType.TABLE :* La génération de la clé primaire se fera en utilisant une table dédiée hibernate_sequence qui stocke les noms et les valeurs des séquences.
- *GenerationType.SEQUENCE :* La génération de la clé primaire se fera par une séquence définie dans le SGBD, auquel on ajoutera l'attribut generator.

Mapping : partie 2

@Temporal : Permet de définir les attributs qui seront temporels, tel que les DATE, les TIME, ou les DATETIME. Ils auront besoin d'un TemporalType (DATE, TIME, TIMESTAMP)

@Transiant : Un attribut Transiant est un attribut qui n'est… pas permanent. Il ne sera pas sauvegardé en base de données. Par exemple l'âge d'une personne pour laquelle nous aurions un attribut *dateDeNaissance*.

Mapping : précisions

@Column : En plus de préciser le nom de la colonne nous pouvons lui préciser plusieurs arguments supplémentaires :

nullable, *length*, *name*, *unique*

```
@Column(name="country", length=50, nullable=false, unique=false)
```

@Table : Nous avons aussi la possibilité de préciser un schéma en particulier si nous en utilisons plusieurs

```
@Entity
@Table(name="country", schema="sakila")
public class Country {
```

Exercice

A partir de ce que nous avons appris :

Créer l’Entity Actor

Créer l’Entity Language

Créer l’Entity Film

Un souci ? Une remarque ?

Une langue ?....

Association : Many To One

Nous pouvons voir qu'il est facile de récupérer l'identifiant, mais que dans un contexte Objet il serait intéressant de récupérer un Objet de type "Language"

La classe Film étant composé de deux d'entre eux.

Pour cela nous disposons du mot clé :

@ManyToOne
En combinaison avec le mot clé :
@JoinColumn(name="some_id")
Nous pouvons réaliser simplement une jointure

Association One to Many

```
@ManyToOne
@JoinColumn(name="language_id")
private Language language;
```

Il faut bien sûr réaliser les getters et setters associés.

On peut remarquer que la requête associée lorsque nous avons besoin des données possède maintenant un “left join”

```
Hibernate: select f1_0.film_id,f1_0.description,l1_0.language_id,l1_0.last_update,l1_0.name,f1_0.last_update,f1_0.length,f1_0.
original_language_id,f1_0.rating,f1_0.release_year,f1_0.rental_duration,f1_0.rental_rate,f1_0.replacement_cost,f1_0.special_fe
atures,f1_0.title from film f1_0 left join language l1_0 on l1_0.language_id=f1_0.language_id where f1_0.film_id=?
ADAPTATION HOLES
```

Sauvegarder des données

Maintenant que nous savons faire quelques association

Essayons de sauvegarder aussi quelques données et observons ce qui se passe.

Pour sauvegarder des informations utilisez simplement :

```
session.persist(Objet);
```

Essayez de manipuler les informations et d'associer les objets que nous venons de créer.

Ajoutez une langue ? Ajoutez des films ?

Association : One To Many

Presque aucune différence avec **@ManyToOne**

@OneToMany
En combinaison avec le mot clé :
@JoinColumn(name=”some_id”)
Nous pouvons réaliser simplement une jointure

ATTENTION on ne récupère plus UN SEUL objet mais PLUSIEURS !

Problème de redondance

Le problème lorsque l'on utilise **OneToMany** et **ManyToOne** de chaque côté avec la JoinColumn est la redondance.

Pour régler cela nous devons plutôt utiliser

@OneToMany*(mappedBy="nomAttribut")*

Association : Many to Many

De la même manière que pour le OneToMany et ManyToOne

Il faudra décider d'une table "référente"

La syntaxe sera cette fois ci :

```
@ManyToMany
@JoinTable(
name = "film_actor",
joinColumns = @JoinColumn(name = "film_id"),
inverseJoinColumns = @JoinColumn(name = "actor_id"))
private Set<Actor> actors;
```

Association Many To Many

Et de l'autre côté

```
@ManyToMany(mappedBy = "actors")
private Set<Film> films;
```

Tout simplement !

Limitation ?

Pas de possibilité de récupérer d'autres attributs

A vous !

Exercice : Faire l’association de Films avec Catégories

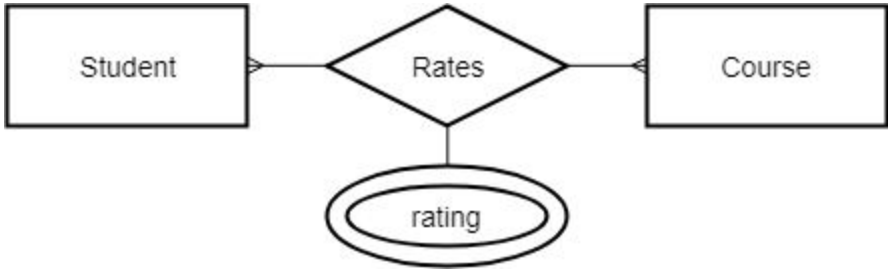
Many To Many : Embeddable

Si nous avons besoin de récupérer d'autres attributs en plus de l'association

@Embeddable

Prenons l'exemple fait par Baeldung :

Nous avons des étudiants et des cours
Les étudiants doivent pouvoir rajouter des notes aux cours suivis



Embeddable

Dans ce cas nous commençons par faire une avec le tag **@Embeddable**

Cette classe va implémenter l'interface *Serializable*

Cette classe ne contient **que la clé !**

C'est une clé composite

Elle devra aussi implémenter les

méthodes *hashCode()* et *equals()*

```
@Embeddable
class CourseRatingKey implements Serializable {

    @Column(name = "student_id")
    Long studentId;

    @Column(name = "course_id")
    Long courseId;

    // standard constructors, getters, and setters
    // hashcode and equals implementation
}
```

Création de l'Entity correspondante

```
@Entity
class CourseRating {

    @EmbeddedId
    CourseRatingKey id;

    @ManyToOne
    @MapsId("studentId")
    @JoinColumn(name = "student_id")
    Student student;

    @ManyToOne
    @MapsId("courseId")
    @JoinColumn(name = "course_id")
    Course course;

    int rating;

    // standard constructors, getters, and setters
}
```

A vous !

Mettre à jour Film et Catégorie pour accéder à last_update

Many To Many : cas spécifique

Il est possible de faire une relation Many To Many avec une nouvelle Entity

Dans le cas ou la clé n’est pas composite (par exemple Inventory)

Dans ce cas nous ferons des relations ManyToOne et OneToMany sur cette Entity particulière

Association : précisions

Sur les association vous pouvez rajouter le paramètre “fetch”

Nous avions vu que nos information n’étaient récupérés que lorsque nous appelons le getter nécessaire

fetch permet de changer ce comportement :

fetch = FetchType.LAZY => Uniquement sur demande

fetch = FetchType.EAGER => A la récupération de la donnée