CHAPITRE 2. LA COUCHE D'ACCES AUX DE DONNEES

1. L'API JPA

Définition
JPA (Java Persistence API) est une partie de l'architecture globale de JavaEE. C'est un ensemble de règles qui constituent un modèle à respecter pour réaliser un **mapping** entre les classes Java et les tables relationnelles. Concrètement, JPA est une bibliothèque (regroupée dans le package **javax.persistence**) qui se constitue d'interfaces, de méthodes (abstraites), d'énumérations et d'annotations. JPA n'est pas un outil ou un framework, c'est une architecture qui définit les principes de mapping des classes Java avec les tables SQL. Cette architecture est implémentée par plusieurs frameworks, certains apportent aussi le support des BD NoSQL comme EclipseLink et Hibernate.

2. Mapping Objet/Relationnel

Définition
Le principe de mapping objet/relationnel (ORM) vise à voir une base de données relationnelle - depuis une application orientée objet - comme une base de données « objet ». Cela signifie que le programmeur ne fera pas d'actions directes (SQL) sur la BD, mais plutôt sur ses propres objets (java, c#, php, …), et ces actions vont se répercuter sur les enregistrements de la BD, à l'aide de frameworks appelés « frameworks ORM ».
Les frameworks d'ORM en Java
▪ Oracle TopLink, l'un des plus anciens frameworks ORM ▪ EclipseLink, constitue l'évolution de TopLink ▪ iBatis, ▪ Hibernate, framework utilisé par défaut dans Spring. ▪ …

3. L'ORM HIBERNATE

1. Introduction & mapping XML

Installation, configuration et connexion de Hibernate avec MySQL en utilisant Maven
▪ Dans Eclipse JavaEE, créer un projet Maven, ▪ Dans la rubrique **dependencies** du fichier pom.xml de Maven, ajouter la dépendance de : o Hibernate, en spécifiant la version, o Connecteur jdbc de MySQL,

```
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>4.3.6.Final</version>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.6</version>
    </dependency>
```

- Dans le dossier `src/main/ressources`, créer le fichier XML de configuration de hibernate, qui doit être nommé **hibernate.cfg.xml** :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://adresse_ip:port_mysql/db</property>
    <property name="hibernate.connection.username">utilisateur</property>
    <property name="hibernate.connection.password">mot_de_passe</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="show_sql">true</property>
  </session-factory>
</hibernate-configuration>
```

Mettre à jour le schéma de la BD en cas de modifications dans les classes Java.

Montrer les actions SQL exécutées par Hibernate lors de l'exécution de la commande maven-install. Doit être définie à **false** en mode production.

Mapping classe ⇔ table : via les fichiers de mapping XML

- Créer la classe Java ordinaire (attributs + accesseurs + constructeur sans paramètres),
- Puis créer le fichier XML de mapping de chaque classe avec sa table SQL :

```
<hibernate-mapping>
   <class name = "Employee" table = "EMPLOYEE">
      <id name = "id" type = "int" column = "id">
         <generator class="native"/>
      </id>
      <property name = "firstName" column = "first_name" type = "string"/>
      <property name = "lastName" column = "last_name" type = "string"/>
      <property name = "salary" column = "salary" type = "int"/>
   </class>
</hibernate-mapping>
```

2. Mapping avec les annotations de JPA

Définitions

Les annotations (*@nom_annotation*) de JPA sont définies (comme tout le contenu de l'api JPA) dans le package javax.persistence. Elles jouent le rôle d'étiquètes qui servent à attribuer une description (table, clef primaire, colonne, auto-incrémentation, valeur par défaut, colonne de jointure, …) à un élément de la classe (classe, attribut ou getter d'un attribut, …).
Les classes destinées à devenir des entités JPA doivent respecter les conditions suivantes :

- Attributs avec getters & setters,
- Constructeur sans paramètres (requis par Hibernate),
- Implémenter l'interface Serializable.

Mapping classe ⇔ table : via les annotations de JPA (la méthode recommandée)

- Designer une classe comme entité JPA :

```
@Entity
public class Client implements Serializable {
```

- Designer la table mappée avec l'entité :

```
import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="client")
public class Client implements Serializable {
```

- Colonne de clef primaire :
  - Clef primaire :

```
@Id
@Column(name="id")
private long id;
```

On peut aussi appliquer les annotations au getter de l'attribut à la place de l'attribut.

  - Clef primaire auto incrémentée :

```
@Id
@Column(name="id")
@GeneratedValue(strategy=GenerationType.IDENTITY)
private long id;
```

Le choix IDENTITY est utilisé quel que soit le SGBD ciblé.

- Colonne ordinaire, en laissant Hibernate choisir le type SQL correspondant au type de l'attribut (dépendamment du SGBD spécifié dans hibernate.cfg.xml), et la taille par défaut du type SQL choisi :

```
@Column(name="nom")
private String nom;
```

- Colonne ordinaire, avec type et taille spécifiques :

Exemple 1 :

```
@Column(name="tel", columnDefinition = "char(13)")
private String tel;
```

Le choix de donner explicitement le type et la taille, implique de connaitre à l'avance les spécificités SQL du SGBD ciblé.

Exemple 2 :

```
@Column(name="anomalie", columnDefinition = "text")
protected String anomalie;
```

- Colonne ordinaire, avec valeur par défaut lors de l'insertion :

```
@Column(name="seuilalerte", columnDefinition = "int(3) default 15")
@Generated(GenerationTime.INSERT)
private Integer seuilAlerte;
```

Fonctionne uniquement avec des attributs de type référence, quand l'attribut égale **null**.

▪ Valeur par défaut + interdire l'insertion :

```
@Column(name="seuilalerte", insertable = false, columnDefinition = "int(3) default 15")
@Generated(GenerationTime.INSERT)
private Integer seuilAlerte;
```

▪ Interdire la mise à jour :

```
@Column(name="code", updatable = false)
protected String code;
```

▪ Interdire la valeur NULL :

```
@Column(name="code", nullable = false)
protected String code;
```

▪ Contrainte d'unicité sur **une** colonne :

```
@Column(name="email", unique=true)
private String email;
```

Si on utilise le mapping via les annotations de JPA, on doit l'indiquer dans le fichier hibernate.cfg.xml avec la balise <mapping> :

```
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3307/spcommandes</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">pass1156</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <property name="show_sql">true</property>

    <mapping class="example.entites.Client"></mapping>
    <mapping class="example.entites.Commande"></mapping>
  </session-factory>
</hibernate-configuration>
```

3. Effectuer des opérations de CRUD sur les objets d'une entité

Créer la session et la transaction (Hibernate 4.3.x)

```
- Partie 0 : les imports nécessaires :
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.Query;

- Partie 1 : créer la SessionFactory
Configuration configuration = new Configuration().configure();
StandardServiceRegistryBuilder builder = new StandardServiceRegistryBuilder()
                .applySettings(configuration.getProperties());
SessionFactory sessionFactory = configuration.buildSessionFactory(builder.build());

- Partie 2 : créer une session
Session session = sessionFactory.openSession();

- Partie 3 : démarrer une transaction
Transaction transaction = session.beginTransaction();

- Partie 4 : effectuer l'une des action CRUD
// (create, retrieve, update, delete)

- Partie 5 : valider la transaction et fermer la session
transaction.commit();
session.close();
```

On peut utiliser la même session pour effectuer plusieurs actions, mais il faut que chaque action soit menée dans une transaction (begin … commit) à part.

Create

```
session.save(objet_entité);
```

Retrieve

```
- get all
Query query = session.createQuery("from Entite");
List<Entite> liste = query.list();
- get by ID
Entite e = (Entite) session.load(Entite.class, id);
Ou bien
Entite e = (Entite) session.get(Entite.class, id);
```

- Langage HQL.
- Utiliser le nom de l'entité au lieu de celui de la table.

load : Lève une exception si l'enregistrement n'est pas trouvé.

get : Retourne null si l'enregistrement n'est pas trouvé.

```
- get avec conditions where, order by, …
Query query = session.createQuery("from Entite where … order by …");
List<Entite> liste = query.list();

- get avec requête paramétrée
Exemple :
Query query = session.createQuery("from Client where id between :min and :max order by name desc");
query.setParameter("min", 10);
query.setParameter("max", 15);
List<Client> liste = query.list();
```

Update

```
session.update(objet_entité);
```

Delete

```
session.delete(session.get(Entite.class,id));
ou bien (de préférence)
session.delete(session.load(Entite.class,id));
```

TP

- Créer un projet **Maven** dans Eclipse.
- En utilisant Maven, intégrer la version **finale** de **Hibernate 4.3.6** dans le projet.
- Dans MySQL, créer une BD nommée spcommandes, et une entité pour les tests. Ex : user(id, uername, password).
- Créer la classe Main qui contiendra la méthode principale, qui va proposer à l'utilisateur les options du menu : afficher, rechercher, ajouter, supprimer et modifier.
- Après chaque exécution d'un sous-menu, on doit retourner au menu principal.
- Pour effectuer les actions du CRUD, la classe Main devra appeler les méthodes (statiques) de GererUser.

4. Spring Data JPA

Définition

Spring Data JPA fait partie de Spring DATA, qui fait partie de l'ensemble d'outils de Spring.

Spring Data JPA constitue une abstraction de Hibernate, car il utilise ce dernier pour implémenter les normes imposées par JPA, mais utilisation est beaucoup plus facile et transparente pour le développeur que celle de Hibernate.

L'utilisation de Spring Data JPA sur des interfaces appelées « repositories », ces interfaces contiennent toutes les fonctionnalités de base, avancées et personnalisables nécessaires pour la persistance en BD des objets des classes entités.

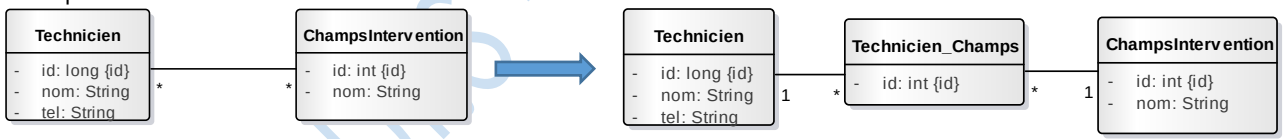
Pour chaque classe entité, on doit créer une interface repository, qui sera utilisée pour effectuer les actions du CRUD, et ce même s'il s'agit d'interfaces, car Spring se chargera de les implémenter et de les instancier automatiquement.

1. Transformation des associations N à N

Transformer les associations plusieurs à plusieurs vers des associations un à plusieurs

Règle :
1- Toutes les tables doivent être mappées à des classes entités, or dans une association plusieurs à plusieurs, il y a 3 tables du côté SQL et 2 classes du côté Java (sauf dans le cas d'une classe association). Dans tous les cas, la meilleure formule est de transformer l'association N à N en deux associations 1 à N.
2- La table intermédiaire prendra comme clefs étrangères les clefs des deux autres tables, mais avec une contrainte d'unicité sur le couple des clefs.

Exemple :

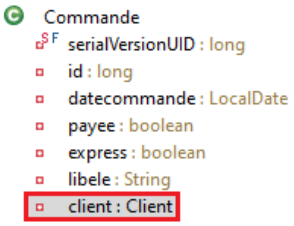


2. Mapping des associations & contraintes

Contrainte d'unicité sur plusieurs colonnes avec @Table

```
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;

@Table(name = "nom_table", uniqueConstraints = @UniqueConstraint(columnNames = {"col1", "col2"}))
```

Contrainte de clef étrangère UN A PLUSIEURS avec @ManyToOne (*meilleure méthode pour mapper une association 1 à plusieurs*)




```
class Commande {
 …
 @ManyToOne(fetch = FetchType.LAZY, optional = false)
 @JoinColumn(name = "idclient")
 @OnDelete(action = OnDeleteAction.NO_ACTION)
 private Client client;
}
```

- **FetchType.*LAZY*** :

L'attribut est chargé uniquement au moment de l'accès, et non lors du chargement initial de l'objet.

- **FetchType.*EAGER*** :

L'attribut est chargé initialement lors du chargement de l'objet.

- **optional = true** :

Décider si la multiplicité est `0..1` (`optional = true`) ou `1..1` (`optional = false`). Dans le cas de la valeur `true`, la colonne de la clef étrangère peut accepter la valeur `null`.

- **@OnDelete(action = OnDeleteAction.*NO_ACTION*)** :

Interdire la suppression en cascade en cas de suppression de l'enregistrement parent. Cette option représente le comportement par défaut. L'inverse est : **OnDeleteAction.*CASCADE***
(Cette annotation se trouve dans `org.hibernate.annotations`)

Pour appliquer les changements au schéma de la BD :
Explorateur des projets → Bouton droit sur le projet Spring Boot → Maven install :

```
2020-01-20 17:44:49.843  INFO 6584 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1
2020-01-20 17:44:49.863  INFO 6584 --- [           main] org.hibernate.dialect.Dialect            : HHH000400: Us
Hibernate: create table commande (id bigint not null auto_increment, datecommande date, express bit, libele varch
Hibernate: create table lignecommande (id bigint not null auto_increment, qtecommandee integer not null, reductio
Hibernate: create table livraison (id bigint not null auto_increment, primary key (id)) engine=InnoDB
Hibernate: create table livreur (id bigint not null auto_increment, nom varchar(255), tel varchar(255) not null,
Hibernate: create table produit (id bigint not null auto_increment, designation varchar(255), prixunit double pre
Hibernate: create table user (id bigint not null auto_increment, password varchar(255), username varchar(255), pr
Hibernate: alter table commande add constraint FKa46ev0lkf78wul2iyo6t3iphp foreign key (idclient) references clie
Hibernate: alter table commande add constraint FKhps2jkj7ouxdx6oman5sfcu8x foreign key (iduser) references user (
Hibernate: alter table lignecommande add constraint FKi74xx560oitpmqemu191biu94 foreign key (idcommande) referenc
Hibernate: alter table lignecommande add constraint FKus68u3o4i3ldild2qtk7c25f foreign key (idproduit) references
2020-01-20 17:44:51.091  INFO 6584 --- [           main] o.h.e.t.i.p.i.JtaPlatformInitiator       : HHH000490: Us
```

Contrainte de clef étrangère UN A PLUSIEURS bidirectionnelle avec @ManyToOne + @OneToMany

Buts (par rapport à la 1iere méthode) :
- Permettre à l'objet parent (multiplicité 1) d'avoir une liste d'objets enfants (multiplicité *).
- Pouvoir contrôler le cycle de vie des objets enfants à partir du parent.

Exemple : Client ↔ Commande
Entité Commande :
Idem que l'exemple précèdent.

Entité Client :

```
@Entity
@Table(name="client")
public class Client implements Serializable {
  …
  @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "client")
  private ArrayList<Commande> commandes;
  …
}
```

- **mappedBy = "client"** :

mappedBy est utilisé pour spécifier l'entité qui contrôle (*owner*) la relation un à plusieurs.
Non nécessaire en cas d'association 1 à plusieurs **unidirectionnelle**.
Le nom spécifié dans **mappedBy** doit être celui que porte l'attribut « un » dans l'entité « plusieurs ».

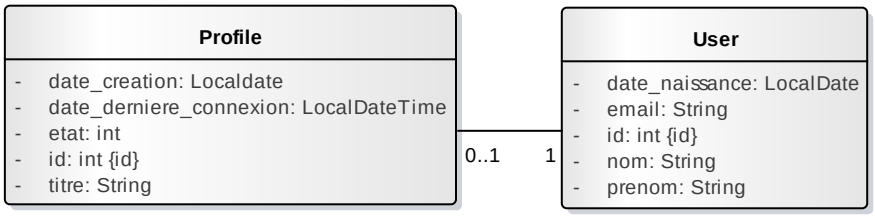
- **cascade = CascadeType.*ALL*** :

Définir l'action à entreprendre en cascade concernant les objets *enfants* en cas de manipulation du *parent*.

- **CascadeType.*PERSIST*** : sauvegarder (insérer) la liste des objets enfants si le parent est sauvegardé,
- **CascadeType.*MERGE*** : sauvegarder avec le même ID l'objet parent après des modifications, la liste des objets enfants est aussi sauvegardée,
- **CascadeType.*DETACH*** : ***DETACH*** a pour but de détacher l'objet entité ciblé du manager d'entité qui le gère. Cette option est utilisée quand on a l'intention d'exclure un objet entité d'une opération (DELETE par exemple). Dans ce cas **CascadeType.*DETACH*** signifie que les objets enfants aussi seront détachés.
- **CascadeType.*REFRESH*** : ***REFRESH*** constitue l'inverse de ***MERGE***, l'objet parent est restitué à son état initial après des modifications, ce qui veut dire que ces modifications sont annulées. Dans ce cas **CascadeType.*REFRESH*** signifie que les modifications des objets enfants aussi seront annulées.
- **CascadeType.*ALL*** : signifie le regroupement de toutes les options précédentes.
- Aucune action n'est prise par défaut, ce qui veut dire que si aucune option cascade n'est spécifiée, les objets enfants ne seront pas affectés par les actions subies par le parent.

Contrainte de clef étrangère UN A UN avec @OneToOne

Exemple : user → profile



2 techniques possibles :

- 1iere technique (la plus utilisée) : une clef étrangère dans l'une des tables, **avec contrainte d'unicité**.
- 2ieme technique : une table intermédiaire, ce qui veut dire que l'association sera traitée comme une association plusieurs à plusieurs.

→ Mapping en utilisant la 1iere technique :

- L'entité ayant la multiplicité 0..1 doit recevoir la clef étrangère de l'autre entité. Si la multiplicité est 1 des deux côtés, alors le choix importe peu.
- Comme pour une association un à plusieurs, on peut choisir un mapping unidirectionnel ou bidirectionnel (l'option choisie dans cet exemple).

Entité User :

```
@Entity
@Table(name = "user")
public class User {
	public User() { }
	@Id
	@Column(name = "id")
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private long id;

	// autres attributs

	@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "user")
	private Profile profile;
}
```

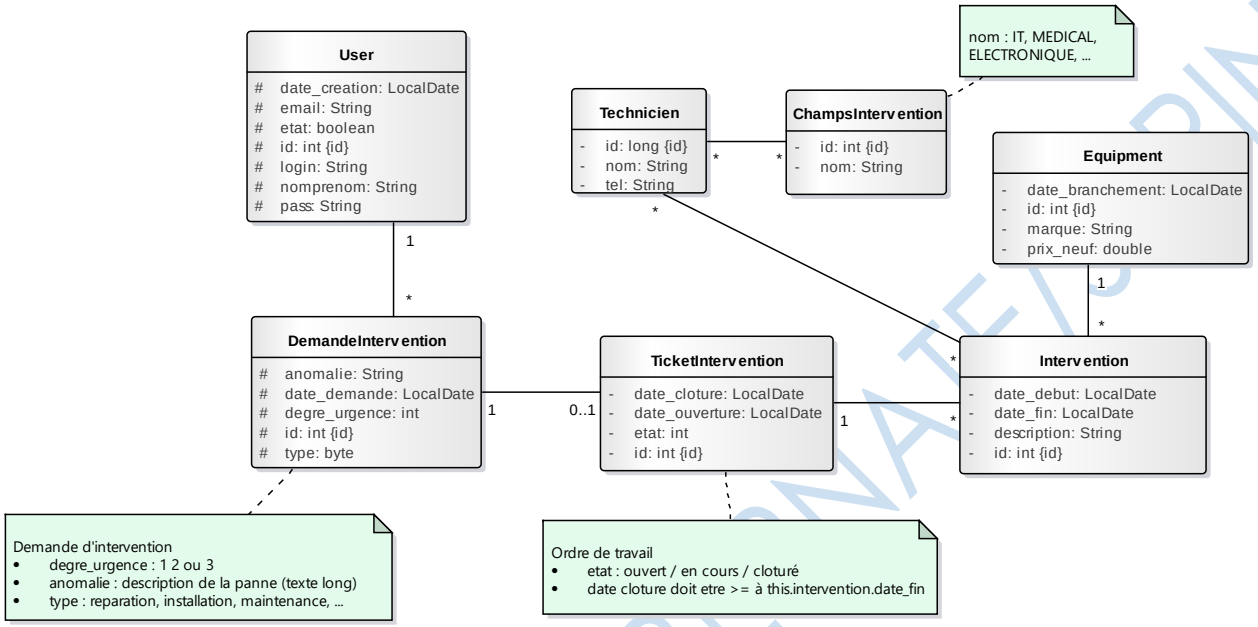
Entité Profile :

```
@Entity
@Table(name="profile")
public class Profile {
	public Profile() { }

	@Id
	@Column(name="id")
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private long id;

	// autres attributs

	@OneToOne(fetch = FetchType.LAZY)
	@JoinColumn(name = "user", nullable = false, unique = true)
	private User user;
}
```

```
Hibernate: create table profile (id bigint not null auto_increment, user bigint not null, primary key (id)) engin
Hibernate: alter table profile drop index UK_g1i4q9q8d0bmwhqqv4o6t551
Hibernate: alter table profile add constraint UK_g1i4q9q8d0bmwhqqv4o6t551 unique (user)
Hibernate: alter table commande add constraint FKhps2jkj7ouxdx6oman5sfcu8x foreign key (iduser) references user (
Hibernate: alter table profile add constraint FKryplhbam423p6jd25epftnqvh foreign key (user) references user (id)
2020-01-23 16:12:36.159  INFO 13404 --- [           main] o.h.e.t.j.p.i.JtaPlatformInitiator       : HHH000490: U
```

TP

- Créer un projet **Maven - SpringBoot - Spring Data JPA - Spring Rest Repositories - MySQL 5**.
- Dans le fichier de configuration de l'application SpringBoot (*application.properties*), utiliser la base de données **medical**.
- Créer les classes entités correspondantes au diagramme UML suivant :



- Mettre à jour le schéma de la BD.