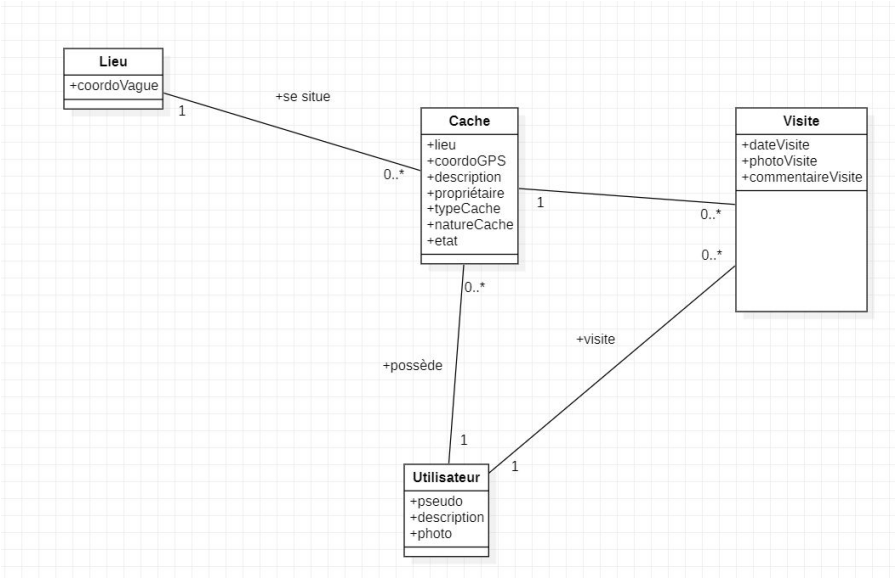
Sacha Lesueur
Coquerelle Loïc

Sommaire :

I] Introduction

Le but de ce compte rendu est de présenter notre solution pour l'application console Geocache comptant pour le module Programmation orientée objet.
Cette application doit permettre la manipulation de données (Create, Read, Update, Delete) depuis une base de données MySQL.

II] Réalisation du diagramme de classe



Pour représenter le concept de cette application, nous avons réalisé un diagramme UML selon le schéma ci-dessus.

On en a donc déduit 4 entités:

- Utilisateur qui va stocker les informations de l'utilisateur tel qu'un pseudo, une description et une photo.

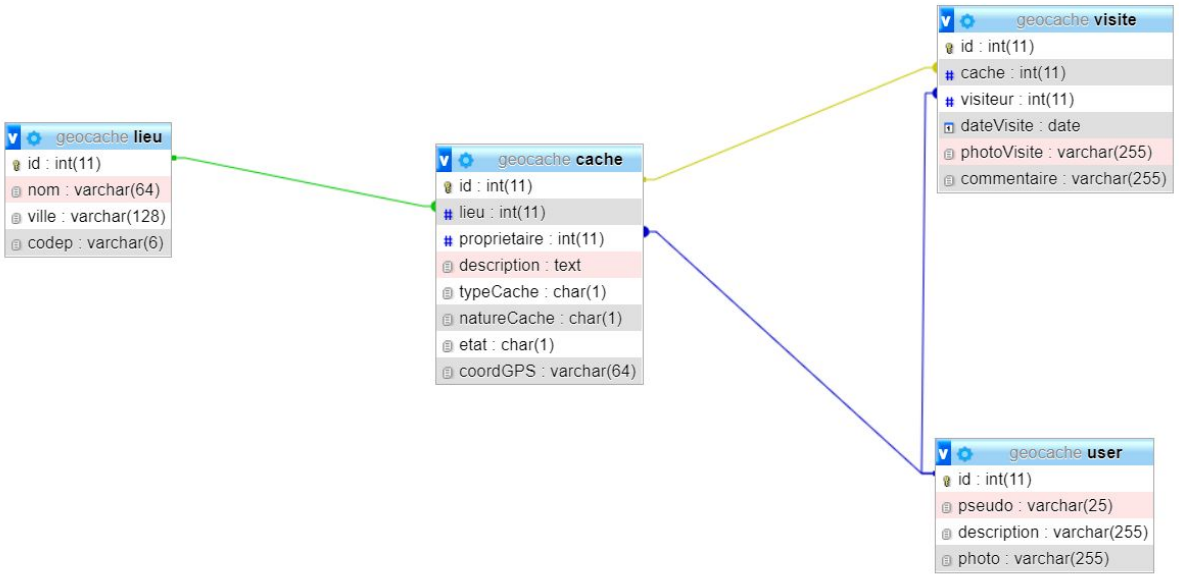
- Cache qui va stocker les informations des différentes caches.(lieu, description, type, nature, propriétaire et son état.)

- Lieu qui va stocker les coordonnées GPS du lieu de la cache.

- Visite qui va stocker une date, un commentaire et une photo de la visite

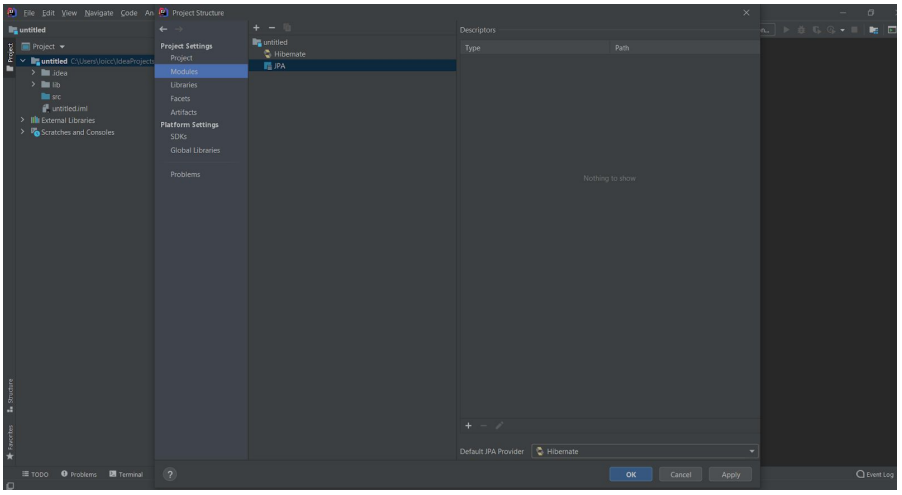
Sacha Lesueur
Coquerelle Loïc

III] Base de données



IV] Mise en place du projet

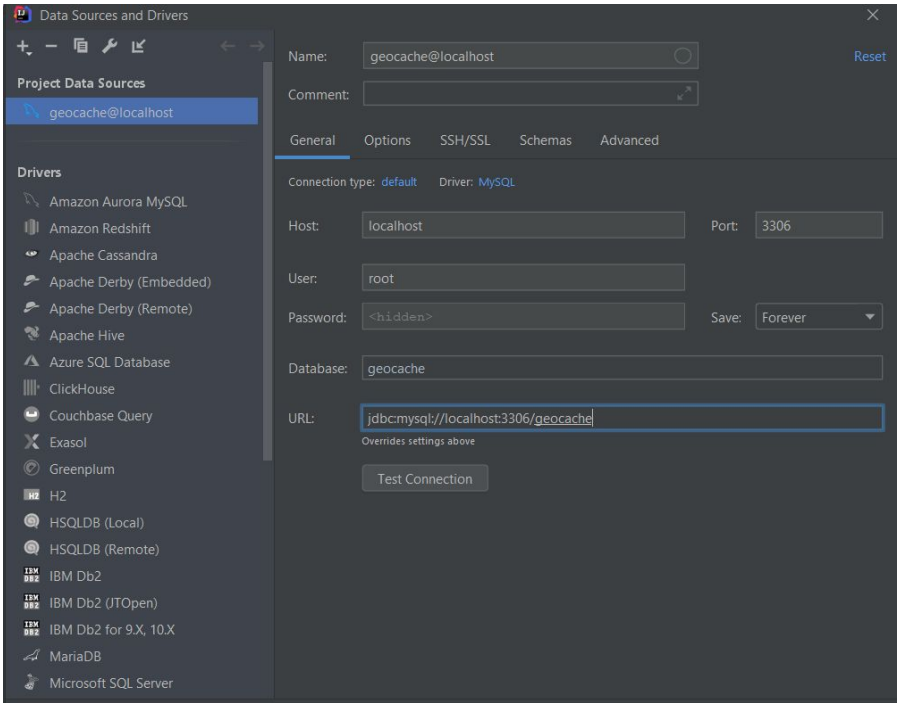
Pour initialiser le projet nous avons créé un nouveau projet sur intellJ.
Aller dans l'onglet file-> project structure et ajouter les modules hibernate et JPA.



Il faut également

On initialise ensuite la base de données, pour ceci on utilise l'onglet Database d'IntelliJ et MySQL.

On réalise la configuration suivante afin de se connecter à notre base de données geocache.



Une fois la base de données bien associée, il faut générer les entités représentant les tables de notre base de données. On utilise l'outil de persistance d'IntelliJ. Une fois générée, il ne faut pas oublier de rajouter le mapping entre nos entités et notre base de données.

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
  <property name="connection.url">jdbc:mysql://localhost:3306/geocache?serverTimezone=UTC</property>
  <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
  <property name='connection.username'>root</property>
  <property name='connection.password'></property>

  <mapping class="geocache.CacheEntity"/>
  <mapping class="geocache.LieuEntity"/>
  <mapping class="geocache.UserEntity"/>
  <mapping class="geocache.VisiteEntity"/>
  </session-factory>
</hibernate-configuration>
```

V] Mise en place des relations entre entités et base de données

Une fois les entités générées par l'outil de persistance, il faut rajouter les différentes relations (oneToMany / ManyToOne et OneToOne).
Ces relations vont permettre de réaliser le lien entre nos entités et les différentes clés étrangères vers les autres entités.
On prend par exemple l'exemple ici de la classe CacheEntity avec les relations vers le "lieu" et le "propriétaire" qui sont les clés étrangères des autres tables.

```java
@JoinColumn(name = "lieu")
@ManyToOne(cascade = CascadeType.DETACH)
private LieuEntity lieu;

@JoinColumn(name = "proprietaire")
@ManyToOne(cascade = CascadeType.DETACH)
private UserEntity proprietaire;
```

VI] Mise en place du CRUD (Create, Read, Update, Delete)

Nous implémentons dans un premier temps une interface DAOInterface, qui définit les méthodes du CRUD qui seront implémentés dans chacune des classes DAO par la suite.

```
package modele;


public interface DAOInterface<T>{

    void create(T obj);

    T read(Integer id);

    void update(T obj);

    void delete(T obj);

}
```

On réalise ensuite une classe abstraite JpaDao implémentant cette interface qui sera la classe mère de tous nos objets DAO (UserDao, VisiteDao, CacheDao, LieuDao).

C'est dans cette classe que nous ouvrons une session qui nous permettra de récupérer l'entityManager qui nous permettra de gérer le CRUD, on initialise cette session ainsi que cet entityManager selon l'exemple suivant.

```
public JpaDao(Class<T> entityClass){
    try {
        this.entityClass = entityClass;
        Configuration configuration = new Configuration();
        configuration.configure("hibernate.cfg.xml");

        ourSessionFactory = configuration.buildSessionFactory();
    } catch (Throwable ex) {
        System.out.println(ex);
        throw new ExceptionInInitializerError(ex);
    }
}

public void openSession(){
    session = ourSessionFactory.openSession();
    em = session.getEntityManagerFactory().createEntityManager();
}
```

On implémente comme prévu les différentes méthodes : read, update, create and delete. Pour faire le create, il faut commencer la transaction, le "persist" permet de réaliser l'action de création.

```
public void create(Object obj) {
    try {
        em.getTransaction().begin();
        em.persist(obj);
        em.getTransaction().commit();
        System.out.println("Entity " + obj.getClass().getName() + " persisted successfully");
    } catch (HibernateException e) {
        e.printStackTrace();
        em.getTransaction().rollback();
    }
}
```

Pour le delete, il suffit d'utiliser la fonction remove de l'entityManager comme ci dessous.

```
public void delete(Object obj) {
    try {
        em.getTransaction().begin();
        em.remove(obj);
        em.getTransaction().commit();
        System.out.println("Entity " + obj.getClass().getName() + " deleted successfully");
    } catch (HibernateException e) {
        e.printStackTrace();
    }
}
```

On suit la même démarche pour réaliser l'update, pour le read on le fait comme ci suivant :

```
public T read(Integer id) { return em.find(this.entityClass, id); }
```

VII] Mise en place du modèle DAO

Pour réaliser le modèle DAO nous ajoutons l'attribut entityClass en paramètre du constructeur du JpaDao. Cet ajout de paramètre nous permet d'instancier les VisiteDao, LieuDao, UserDao et CacheDao.

Les classes UserDao, CacheDao, LieuDao et VisiteDao étant sensiblement les mêmes nous ne présenterons ici qu'une seule de ces classes.

```
public class CacheDao extends JpaDao<CacheEntity> implements DAOInterface{

    private static CacheDao instance;

    public CacheDao() { super(CacheEntity.class); }

    public static CacheDao getInstance(){
        if(instance == null) {
            instance = new CacheDao();
        }
        return instance;
    }
```

Le constructeur de cette classe appelle le constructeur de la classe mère JpaDao.
On implémente également une méthode getInstance qui nous renvoie une instance de notre classe.
Cette méthode va permettre d'avoir des instances de cette classe même depuis d'autres classes.

Dans ces classes on redéfinit la méthode toString afin d'avoir un affichage de nos données de manière plus lisible.

VIII] Exécution du programme en console

Pour exécuter cette application console il suffit d'exécuter le fichier Main.java.

On effectue un test de lecture sur le Lieu d'id 1, l'utilisateur d'id 1 afin de tester le bon fonctionnement de nos méthodes de lecture.

```
LieuEntity{id=1, nom='Gros Horloge', ville='Rouen', codep='76000'}
```

```
UserEntity{id=1, pseudo='Loic', description='Une mec beau et très très hot', photo='null'}
```

On obtient les résultats ci-dessus pour la lecture.

On ajoute ensuite un utilisateur d'id 57 sans aucune autre information. On obtient le résultat ci-dessous.

```
User -> read :
UserEntity{id=1, pseudo='Loic', description='Une mec beau et très très hot', photo='null'}
Entity geocache.UserEntity persisted successfully
UserEntity{id=57, pseudo='null', description='null', photo='null'}
```

```
Entity geocache.UserEntity persisted successfully
UserEntity{id=57, pseudo='null', description='null', photo='null'}
Entity geocache.UserEntity updated successfully
UserEntity{id=57, pseudo='Bernard', description='null', photo='null'}
```

On modifie ensuite cet user juste créé en modifiant son pseudo, on obtient ce résultat.

```
UserEntity{id=1, pseudo='Loic', description='Une mec beau et très très hot', photo='null'}
Entity geocache.UserEntity deleted successfully
```

On peut également supprimer un utilisateur, comme le montre le screen précédent, Il n'est après vérification plus présent dans la base de données.

IX] Liste complète et liste filtrée

```
public static List getListFiltre(UserEntity user){
    Query query = instance.em.createNamedQuery("listVisiteFiltre",CacheEntity.class);
    query.setParameter("proprietaire", user);

    return query.getResultList();
}
```

Pour les listes nous avons décidé d'implémenter une liste qui nous renverra toutes les caches que possède un utilisateur.
On utilise pour cela l'instance de la classe CacheDao dont nous avons parlé précédemment, ainsi qu'une Query permettant de récupérer notre liste.

X] Bascule vers MongoDB

Cette partie n'a pas été traitée par manque de temps.

XI] Conclusion

Pour conclure nous avons dans un premier temps un diagramme UML qui nous a permis de mieux comprendre le but et le concept même de l'application qui nous était demandé. Ce diagramme nous a permis de concevoir un modèle de données stable qui nous a servi de base pour débuter notre projet. Ce modèle de données étant le cœur même de cette application. Nous avons ensuite repris la structure mise en place durant les cours sur le TP Hôpital. Cette structure nous a permis d'avoir de solides bases quant à l'initialisation du projet. De cette base nous avons suivi une nouvelle fois la démarche vu durant les cours afin d'associer notre base de données à notre projet grâce aux outils d'intelliJ.
Une fois tout cela fait, nous avons donc réalisé les différentes relations, nous avons connu quelques difficultés par la suite lors de l'implémentation des DAO en lien avec ces relations. La partie du DAO fut délicate pour nous puisque nous avons perdu beaucoup de temps sur une erreur que nous ne comprenions pas. Cette erreur venait d'une mauvaise inclusion d'une librairie JEE qui prenait le pas sur la configuration écrite dans le fichier hibernate.xml. De ce fait, aucun de nos modèles DAO ne semblait communiquer avec la base de données.
Une fois ce problème résolu nous avons commencé à nous pencher sur les listes, nous avons également connu des difficultés dans cette partie, qui venait comme dit précédemment de nos relations ManyToOne, OneToMany qui n'était pas correctement initialisé.
Par manque de temps nous n'avons pas pu commencer la partie sur le basculement vers MongoDb.