Labo Spring 6
- *Hibernate* -

Objectif
- Accéder à une base de données MySQL en utilisant Hibernate

Dans la suite de l'exercice, le code magique va être recherché dans la base de données (accès en lecture) et l'enfant qui aura introduit ses coordonnées via le formulaire d'inscription sera enregistré dans la BD (accès en écriture).

NB. Attention, ne testez votre application qu'à l'étape 9.

Etape 1 – Installer MySQL Server

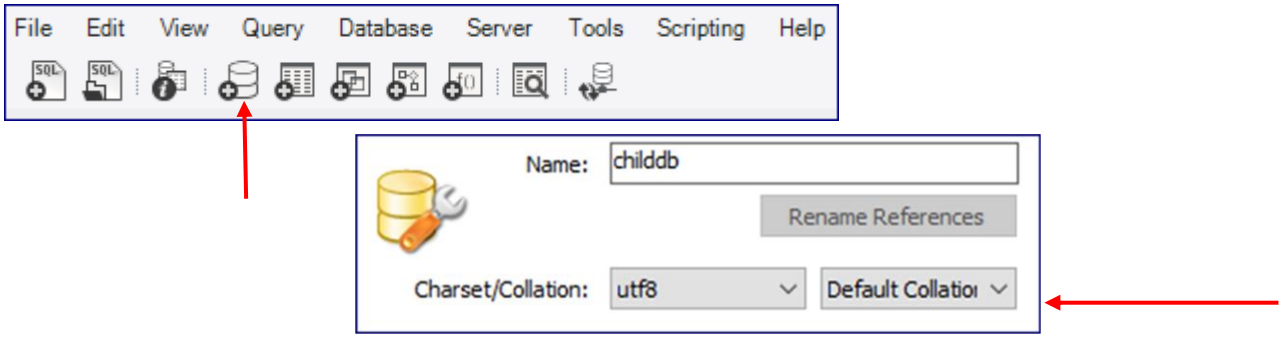
Pour rappel, ne pas installer Workbench seul car il ne contient pas le serveur MySQL !

Attention, vous aurez besoin dans cet exercice du nom d'utilisateur associé à votre base de données ("root" ou autre) et du mot de passe associé.

Etape 2 – Créer un schéma MySQL

Assurez-vous que le serveur MySQL tourne :
Task Manager ⇨ Services

Créez un schéma MySQL appelé *childdb*.



Pour éviter tout problème de type "*timezone*", effectuez les deux actions suivantes (**à réexécuter chaque fois que le serveur MySQL redémarre**) :

⇨ Sélectionnez la base de données ⇨ clic droit ⇨ Set as Default Schema

⇨ Puis exécutez les instructions suivantes :

```
set @@global.time_zone  = '+00:00' ;
set @@session.time_zone = '+00:00' ;
```

Etape 3 – Configurer Spring Boot pour accéder à la base de données

Hibernate

Hibernate est un framework *ORM (Object Relational Mapping*) open source qui comble le fossé entre l'orienté objet et les bases de données relationnelles en convertissant les données d'une base de données relationnelle en objets manipulables en programmation orientée-objet, et inversement.

Il propose de remplacer les accès directs aux bases de données relationnelles (SQL) par des fonctions de haut-niveau manipulant des objets, et ce, via un "mapping" entre des tables et des classes.

Hibernate fournit, outre des instructions d'accès en écriture et en lecture prédéfinies, un système aisé d'écriture de requêtes personnalisées.

Déclarez une spring datasource dans *application.yml*.

Ajoutez deux dépendances dans le *pom.xml*.

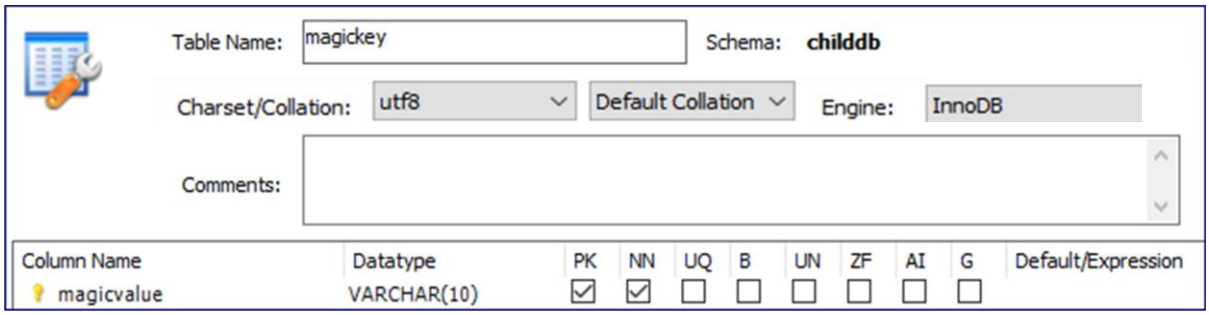
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.17</version>
</dependency>
```

Etape 4 – Créer les tables

Créez la table *magickey*. N'y prévoyez qu'une seule colonne appelée magicvalue.

Cette colonne est identifiante.

N.B. Charset/Collation : *utf8* - *Default Collation* et Engine : *InnoDB*

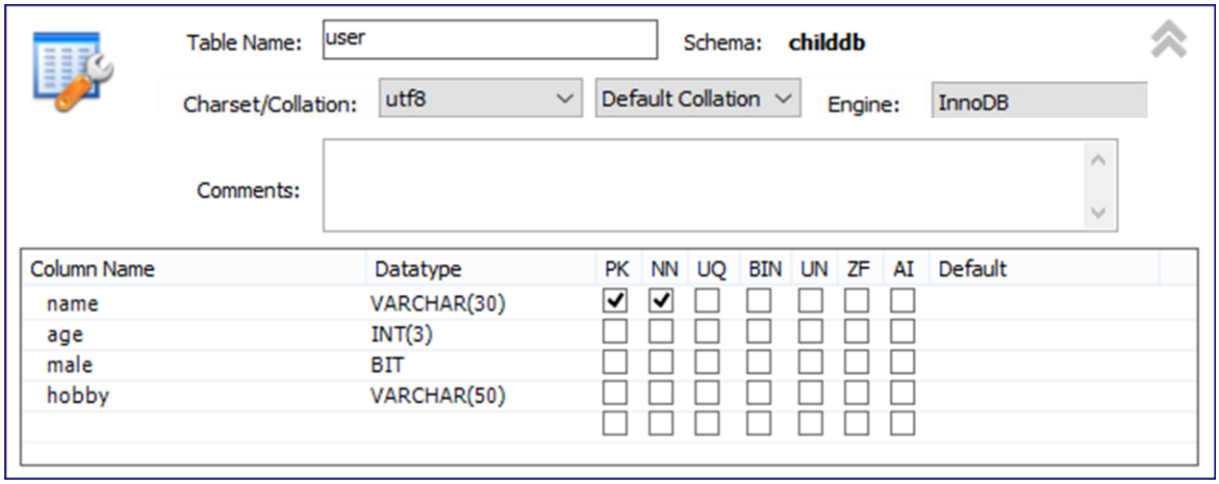


Insérez au moins 5 lignes dans cette table.

Créez la table *user*. Prévoyez les colonnes suivantes :

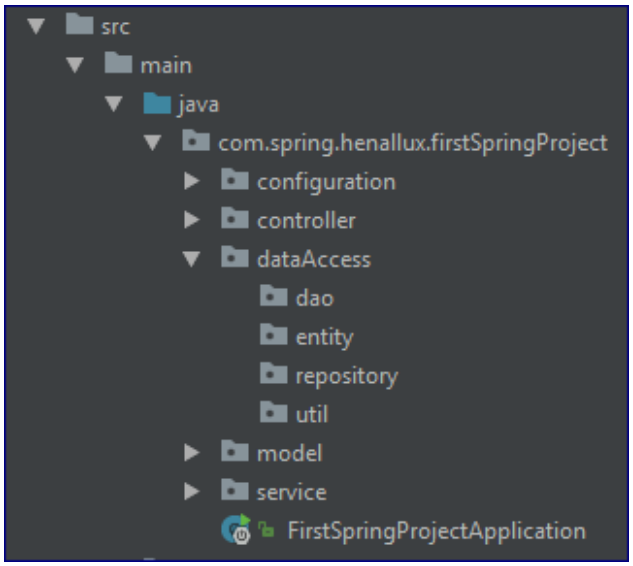
- name (identifiant)
- age
- male (ex : type BIT pour booléen – valeurs true ou false)
- hobby

Rappel : Charset/Collation : *utf8* - *Default Collation* et Engine : *InnoDB*



Etape 5 – Créer les packages de gestion de la persistance des données

Créez dans le package *com.spring.henallux.firstSpringProject* 4 nouveaux packages : *dataAccess.entity*, *dataAccess.repository*, *dataAccess.util* et *dataAccess.dao*. Ce qui donnera l'architecture suivante :



Etape 6 – Créer la classe de type entité *MagicKeyEntity*

Entity Class

Les classes de type entité sont des composants qui représentent les données persistées dans une base de données. Chaque classe entité est le **miroir d'une table** de la base de données ; chaque colonne de la table correspond à une variable d'instance de la classe entité.

Grâce à un framework ORM comme Hibernate, le programmeur peut manipuler des objets de ces classes plutôt que d'accéder directement à la base de données.

Une classe de type entité est un Plain Old Java Object (POJO) annoté *@Entity*, avec (au moins) un constructeur sans argument et des variables d'instance déclarées privées (+ getters/setters publiques). A noter que si des objets de ces classes doivent être transférés d'une machine distante à une autre (via le réseau), la classe doit être en outre déclarée sérialisable (*implements Serializable*).

L'annotation *@Table* sur la classe de type entité permet de préciser le nom de la table de la base de données dont la classe est le miroir.

Chaque variable d'instance correspondant à une colonne doit être annotée *@Column* (en précisant le nom de la colonne dans la table).

La variable d'instance correspondant à l'identifiant est annotée *@Id*.

Créez la classe *MagicKeyEntity* dans le package *entity*.

Cette classe doit être le miroir de la table *magickey* : utilisez les annotations *@Entity* et *@Table* pour lier cette classe à la table *magickey* (cf. Module 12 – Entity Bean).

Définissez-y une variable d'instance correspondant à la colonne identifiante *magicvalue* de la table (utilisez pour ce faire les annotations *@Id* et *@Column*).

N'oubliez pas les getters/setters et au moins un constructeur sans argument

Etape 7 – Créer l'interface *MagicKeyRepository*

Data Repository

Une interface (Repository) est fournie pour chaque classe de type entité. Cette interface contient des méthodes **CRUD** (Create - Read - Update - Delete) d'accès classiques en écriture et en lecture sur la table correspondant à cette classe entité.

Pour utiliser ces instructions prédéfinies, le programmeur doit créer une sous-interface à partir de l'interface *JpaRepository* en veillant à définir correctement les types que sont la classe entité d'une part et le type de l'identifiant d'autre part. Cette sous-interface doit être annotée *@Repository*. Attention, **aucune implémentation de l'interface de type *@Repository* ne doit être fournie** par le programmeur ; c'est le framework Hibernate qui s'en charge.

Quelques méthodes prédéfinies dans l'interface *JpaRepository<T,ID>* (et donc disponibles dans toutes ses sous-interfaces) :

<S extends T> S **save**(S entity)
void **delete**(T entity)
T **findOne**(ID primaryKey)
List<T> **findAll**()
Long **count**()
boolean **exists**(ID primaryKey)

Créez l'interface *MagicKeyRepository* dans le package *repository*. Cette interface offrira les méthodes CRUD de base pour lire et écrire dans la table *magickey*.

Cette interface hérite de l'interface générique *JpaRepository*. Soyez attentif à bien définir les paramètres de l'interface que vous créez : donnez le nom de l'entity class correspondante et préciser le bon type d'identifiant.

Utilisez l'annotation *@Repository*.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.spring.henallux.firstSpringProject.dataAccess.entity.MagicKeyEntity;

@Repository
public interface MagicKeyRepository extends JpaRepository<MagicKeyEntity, String>{
}
```

Pour rappel, vous devez définir l'interface mais pas donner d'implémentation. C'est Hibernate qui s'en charge !

Etape 8 – Créer la classe *MagicKeyDAO*

Design Pattern DAO

Le Design Pattern DAO tel que vu au bloc 2 doit être implémenté.

L'objectif est de séparer l'accès au système de persistance de données des autres couches, de sorte que si le système de persistance des données change (par exemple, des fichiers xml au lieu d'une base de données relationnelle), les couches supérieures ne soient pas impactées.

Le package *dao* va contenir les interfaces DAO ainsi que les classes qui implémentent ces interface DAO. Ce sont les méthodes implémentées dans le package *dao* qui pourront être appelées par les couches supérieures (par exemple, des contrôleurs ou des services). Les méthodes du package *dao* vont faire appel aux méthodes déclarées dans les interfaces de type *repository*.

Les couches supérieures manipulent des objets de type modèle (package model) et ne peuvent en aucun cas manipuler des objets de classes entités. Les classes entités sont des miroirs de tables et font donc partie de la couche Data Access. Aucun objet de type entité ne peut être remonté aux couches supérieures. Les méthodes du package *dao* reçoivent donc des arguments de type modèle (classes de la couche model) et retournent des objets de type modèle.

Implémentez le **Design Pattern DAO** dans votre projet.

Créez dans le package *dao* une interface *MagicKeyDataAccess*. Cette interface doit proposer toutes les méthodes que vous voulez exposer aux couches supérieures. Ces méthodes ne peuvent donc ni recevoir en argument ni retourner des objets de type *MagicKeyEntity*, car liées au type de persistance des données choisi, en l'occurrence ici une base de données relationnelle.

Créez ensuite dans le package *dao* la classe *MagicKeyDAO* qui implémente cette interface.

Utilisez l'annotation *@Service* sur la classe.

Transaction

Une transaction est un ensemble d'opérations considérées comme formant une unité logique : ou bien l'entièreté des opérations est exécutée (*commit*) ou bien aucune opération ne l'est (*rollback*).

L'objectif des transactions est de garder la base de données dans un état cohérent en cas de panne ou d'échec au milieu d'une transaction.

Si une méthode contenant plusieurs instructions d'accès à une base de données doit être considérée comme constituant une transaction, elle est annotée *@transactional.* Cela signifie qu'en cas d'erreur dans la transaction, toutes les opérations d'accès à la base de données prévues dans la méthode seront annulées.

N.B. Si l'on souhaite qu'une méthode soit considérée comme une transaction, il faut utiliser l'annotation *@Transactional* sur la méthode. Si chaque méthode de la classe DAO doit être considérée comme une transaction, il suffit de placer l'annotation *Transactional* sur la classe DAO. Pour que Hibernate applique le mécanisme des transactions, il faut ajouter l'annotation *@EnableTransactionManagement* sur la classe Application

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication
@EnableTransactionManagement
public class FirstSpringProjectApplication {

    public static void main(String[] args)
    {
      SpringApplication.run(FirstSpringProjectApplication.class, args);
    }
}
```

Dans votre projet, prévoyez dans l'interface *MagicKeyDataAccess* la méthode *getMagicKeys()* qui doit retourner la liste des clés (ArrayList<String>) contenues dans la table *magickey*.

Implémentez cette méthode dans *MagicKeyDAO.* Cette méthode devra appeler la méthode *findAll()* disponible dans le repository. Récupérez une référence vers le repository correspondant par injection de dépendance (@*Autowired*).

Voici un début de code (en supposant que toutes les méthodes doivent être considérées comme des transactions), à vous de le compléter.

```java
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;

@Service
@Transactional
public class MagicKeyDAO {

    private MagicKeyRepository magicKeyRepository;

    @Autowired
    public MagicKeyDAO(MagicKeyRepository magicKeyRepository) {
        this.magicKeyRepository = magicKeyRepository;
    }

    public ArrayList<String> getMagicKeys()
    {
      List <MagicKeyEntity> magicKeyEntities = magicKeyRepository.findAll();
```

Etape 9 – Adapter la classe *WelcomeController*

Adaptez la méthode *POST* de *WelcomeController* : faites-en sorte que le code magique introduit par l'utilisateur soit maintenant comparé aux codes (clés) de la base de données retournés par la méthode *getMagicKeys()* de la classe *MagicKeyDAO*.

Pour ce faire, récupérez une référence vers une instance de *MagicKeyDAO* par injection de dépendance (utilisez l'annotation @*Autowired*).

Attention, appliquez le Design Pattern DAO : déclarez dans le controlleur une variable d'instance de type **interface** *MagicKeyDataAccess* et non de type *MagicKeyDAO* !

Pour rappel, la méthode *contains* existe dans la classe *ArrayList*.

Testez votre application.

En cas de problème :

Si vous avez un message d'erreur signalant qu'il est impossible de créer un objet de type *Repository*, c'est peut-être que Hibernate n'arrive pas à se connecter à la BD. Vérifiez :

- Il y a peut-être un problème de timezone ⇨ exécutez dans MySQL les instructions suivantes (à réexécuter chaque fois que le serveur MySQL redémarre) :

  set @@global.time_zone = '+00:00' ;
  set @@session.time_zone = '+00:00' ;

- Vérifiez la déclaration de la datasource dans *application.yml* (attention, sensible aux indentations ⇨ 2 ou 4 blancs en début de ligne).

Etape 10 – Enregistrer l'utilisateur dans la base de données

Basez-vous sur les étapes précédentes pour enregistrer l'utilisateur dans la base de données dès qu'il a rempli le formulaire d'inscription.

Pour ce faire, créez la classe *UserEntity*, l'interface *UserRepository*, l'interface *UserDataAccess* et la classe *UserDAO,* en les plaçant dans les bons packages.

Pour rappel, les méthodes de *UserDataAccess* (et donc de *UserDAO*) ne peuvent ni recevoir en argument ni retourner des objets de type *Entity* classes, car liées au type de persistance des données choisi, en l'occurrence ici une base de données relationnelle. Il faut donc recevoir et retourner des objets de type modèle, donc des objets de type *User* et non pas de type *UserEntity*.

Par conséquent, comme vous devrez transformer un objet de type *User* en un objet de type *UserEntity* et inversement, prévoyez la classe *ProviderConverter* dans le package *util* qui proposera les méthodes qui s'en chargeront, à savoir :

public UserEntity userModelToUserEntity(User user)

public User userEntityToUserModel(UserEntity userEntity)

Annotez la classe *ProviderConverter @Component* et implémentez ses méthodes. Vous pouvez aussi utiliser l'outil Bean Mapper ***Dozer*** afin de faciliter l'écriture des ces méthodes de conversion.

Outil de mapping *Dozer*

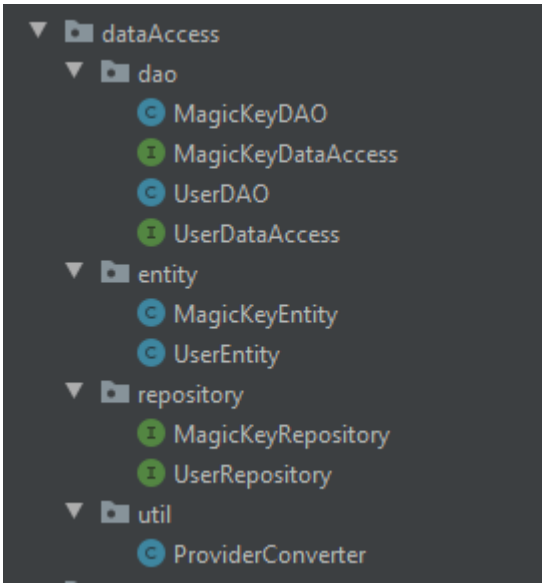
L'écriture du code de la classe *ProviderConverter* est fastidieuse. En effet, il faut recopier ligne par ligne chaque variable d'instance.

L'outil *Dozer* peut se charger de la conversion automatique des variables de même nom dans deux classes. Il suffit d'appeler la méthode *map* sur un objet de type *DozeBeanMapper*, en passant comme paramètre un objet d'une classe et en second paramètre la classe de destination ; la méthode *map* retourne un nouvel objet de la classe de destination dans lequel les variables d'instance sont remplies à partir de l'objet fourni en entrée.

Pour utiliser *Dozer*, ajoutez une dépendance dans le pom.xml :

```
<dependency>
    <groupId>net.sf.dozer</groupId>
    <artifactId>dozer</artifactId>
    <version>5.5.1</version>
</dependency>
```

Structure du package *dataAccess* :



Prévoyez dans *UserDAO* une méthode qui permet d'enregistrer un objet de type *User* dans la base de données : *public User save(User user)*

Adaptez la méthode *POST* de *InscriptionController* : appelez la méthode *save* de *UserDAO* pour enregistrer dans la base de données l'utilisateur correspondant au formulaire.

Etape 11 – Créer des requêtes personnalisées

Vous pouvez ajouter des requêtes personnalisées dans un repository.

Requêtes personnalisées

Hibernate permet l'ajout de requêtes supplémentaires en plus de celles déjà prédéfinies dans l'interface repository.

Deux façons de procéder existent : *Customized Query* ou *Named Query*.

Customized Query

Des requêtes additionnelles peuvent être générées sur base du nom de méthode ; ce nom doit alors être construit par le programmeur en combinant des mots clés et des noms de variables d'instance de la classe entité. Le programmeur déclare ces méthodes dans l'intreface repository. Aucune implémentation ne doit être écrite par le programmeur ; c'est à la charge du framework.

Exemple :
public List<BookEntity> ***findByNbPagesAndTitleOrderByIsbn*** *(Integer pages, String title) ;*

Named Query

S'il n'est pas possible de générer la requête souhaitée sur base uniquement du nom de la méthode construit en combinant les mots clés disponibles, il est toujours possible pour le programmeur de donner le code de la requête. La méthode est alors annotée *@Query* et la syntaxe est alors proche du SQL.

Référez-vous au module 13 – Query pour la liste des mots clés disponibles afin de créer et de tester des requêtes personnalisées de type *Customized Query*. Pour rappel, vous ne devez pas donner de code à ces méthodes mais bien construire leur nom en combinant des mots clés.

Ajoutez deux "Customized Queries" dans *UserRepository* en créant le nom des méthodes en combinant les mots clés adéquats afin de :

- Lister les enfants (*List*<*UserEntity*>) qui ont un âge compris entre deux valeurs données et qui aiment un hobby donné (l'âge minimum (de type Integer), l'âge maximum (de type Integer) et le hobby (de type String) sont donnés en arguments) ;

- Lister les enfants (*List*<*UserEntity*>) qui ont choisi un des hobbies donnés (la liste des hobbies est donnée en argument sous forme d'une *ArrayList*<*String*>), le tout trié par ordre alphabétique sur le prénom de ces enfants.

Adaptez l'interface *UserDataAccess* en y ajoutant les deux méthodes suivantes (les noms de méthode proposés ici ne sont pas significatifs, vous pouvez les modifier) :

- ArrayList<User> getChildByAgeAndHobby(Integer ageMin, Integer ageMax, String hobby) ;

- ArrayList<User> getChildByHobbies(ArrayList<String> hobbies).

Implémentez ces méthodes dans la classe *UserDAO* en appelant les méthodes adéquates du repository.

Afin de tester ces méthodes, générez la méthode *toString* dans la classe modèle *User*.

Couche Vue en programmation Web <> affichage à la console

L'affichage à la console n'a pas de sens en programmation Web : la communication avec l'utilisateur se fait via des pages Web. Pour éviter le couplage de couches, les entrées sorties devraient être gérées exclusivement via les pages Web (couche Vue).

L'affichage à la console devrait être évitée. On y fait une entorse juste le temps de tester nos "Customized Queries".

Pour simplifier l'écriture de code et juste le temps de tester ces méthodes, appelez-les dans la méthode post du *InscriptionController* :

- affichez à la console la description (*toString*) des enfants entre 2 et 10 ans qui aiment la musique ;

- affichez à la console la description (*toString*) des enfants qui ont comme hobby soit la nature soit le sport, le tout trié par ordre alphabétique sur le nom.