

SQL dans un langage de programmation

Connexion du programme à une base de données

Introduction

- Les données gérées par un programme sont **volatiles**. Elles sont stockées dans la mémoire vive de l'ordinateur lorsqu'il est actif, mais une fois que le programme s'arrête, ces données disparaissent
- Hors, une application peut avoir besoin de **stocker et charger des données**
- Il existe différents moyens : fichiers simples, fichiers csv, ou bien, une **base de données**
- Une base de données (par exemple, relationnelle) définit un moyen simple d'organiser, stocker et charger des données. Le protocole qu'on utilise pour communiquer avec est le SQL
- En plus d'assurer **l'intégrité des données** selon les différentes **contraintes** définies lors de la création de la base, celle-ci peut également être distante et **partagée par différents programmes** de différentes natures (un programme java, C++, PHP, etc...)
- Normalement, dans le découpage d'une application **client/serveur**, c'est l'application **serveur** qui est chargée de communiquer avec la **base de données**

Architecture en couche

- L'architecture d'un logiciel se découpe en différentes **couches** :
 - La couche **IHM** (Interface **H**omme **M**achine)
 - La couche **Application**
 - La couche **Métier** aussi appelée **BLL** (*business logic layer*)
 - La couche **Stockage** (si besoin) aussi appelée **DAL** (*data access layer*)
 - La couche **Réseau** (si besoin) ou **Transport**

Architecture en couche - IHM, application, métier

- Les couches **IHM**, **application** et **métier** sont les trois couches principales d'une application. Les échanges entre ces trois couches permettent d'implémenter une architecture **MVC** (cela devrait vous rappeler vos cours de Web...)
- La couche **IHM** permet de gérer les différentes fenêtres graphiques et surtout **l'interaction avec l'utilisateur**.
- La couche **métier** contient le cœur de l'application, à savoir les différentes **entités** manipulées (exemple : produit, client..) ainsi que des classes de **services** appelées **manager** qui permettent de manipuler ces entités d'implémenter la partie **logique** de votre application.
- Enfin, la couche **application** permet de faire le lien entre la couche **IHM** et la couche **métier**. Elle contient différents **controllers** dont le rôle est de gérer les **événements** qui surviennent sur l'interface et d'envoyer des requêtes auprès de la couche **métier** et de transmettre les résultats obtenus à **l'IHM**.

Architecture en couche - La couche stockage

- La **couche stockage** contient des classes dont le rôle exclusif est de communiquer avec une **source de données** (BDD ou autre) afin d'assurer la **persistance des données** (les sauvegarder et les charger)
- Lorsqu'un le programme modifie des données en locale, la couche stockage est chargée de les **synchroniser** avec la source de stockage
- C'est également au niveau de cette couche que le programme vient **recupérer les données**
- Dans le contexte de l'utilisation d'une **BDD**, c'est donc ici qu'on retrouvera tout le code chargé d'envoyer des **requêtes SQL** vers la base et de traiter le résultat
- Cette couche est également chargée du **mapping** (conversion) des **données brutes** reçues depuis la source en **objets** de l'application (les entités de la **couche métier**)
- Les couches de **services** (tels que les **managers**) de la couche **métier** vont se servir des classes définies dans la couche **stockage** afin de gérer la **persistance** des données manipulées par l'application.

La couche stockage - Principe CRUD

- Techniquement, chaque type **d'entité** de l'application doit avoir une classe permettant de la gérer de manière persistante.
- Cette classe fournit donc des opérations de **lecture / écriture** sur la **source de données**
- On dit qu'on implémente le **CRUD** quand on peut réaliser les opérations suivantes au niveau d'un type d'entité donné :
 - Création d'une nouvelle entité sur la source (**Create**)
 - Récupération de toutes les entités de ce type (ou d'une entité précise) depuis la source (**Read**)
 - Mise à jour (modification des valeurs) d'une entité sur la source (**Update**)
 - Suppression d'une entité sur la source (**Delete**)
- Bien sûr, il est tout à fait possible d'implémenter des opérations plus précises (ex : récupération de toutes les personnes de + de 40 ans), en plus du **CRUD**

Interactions avec ma couche métier

- Comme nous l'avons vu, la couche métier contient toutes les **entités** de l'application (exemple : Produit, Client, Commande...) ainsi que la **logique de l'application**
- La partie “**logique**” est principalement gérée par des **classes de services** appelées **manager**. Un **manager** gère une entité précise (ManagerClient, ManagerProduit, etc...)
- Chaque **manager** est donc lié à un type d'entité et **gère l'ensemble de ces entités**. On lui envoie des **requêtes** quand on veut récupérer ou modifier des données à l'intérieur de l'application
- Les managers se **synchronisent** avec la couche **stockage**. Par exemple, quand on veut créer une **nouvelle entité**, on envoie une **requête** à un manager qui va instancier un objet (et éventuellement le garder en mémoire). A ce moment là, le manager est donc aussi chargé de communiquer avec la **couche stockage** afin de **sauvegarder l'entité** de manière **persistante**.
- La couche **IHM** et **application** ne doivent **pas communiquer avec la couche stockage**! D'ailleurs, dans une application client / serveur, la couche IHM se trouve côté client et la couche stockage côté serveur

Lazy loading VS Eager Loading

- Les **managers** peuvent adopter deux **stratégies** vis-à-vis des entités gérées :
 - Le **Lazy Loading** : consiste à charger les entités **seulement quand on en a besoin**, depuis la source de données. Les managers **ne gère pas de collection locale** des entités, tout est géré au niveau de la source de données. Quand on veut lire une donnée, on passe donc tout le temps par la source de données (BDD / fichier, etc...). Globalement, c'est la manière de faire d'une application web en PHP
 - Le **Eager Loading** : consiste à charger **toutes les entités** du type géré par un manager dans le programme (au lancement). Le manager gère donc un **ensemble d'entités** sous la forme d'une **collection** (list, set...) en **local**

Pour la lecture, il n'y a pas besoin de lire de nouveau dans la source de données, car **tout est déjà chargé**. Les modifications apportées (mise à jour, suppression) doivent être répercutées en local et au niveau de la couche stockage

Lazy loading VS Eager Loading

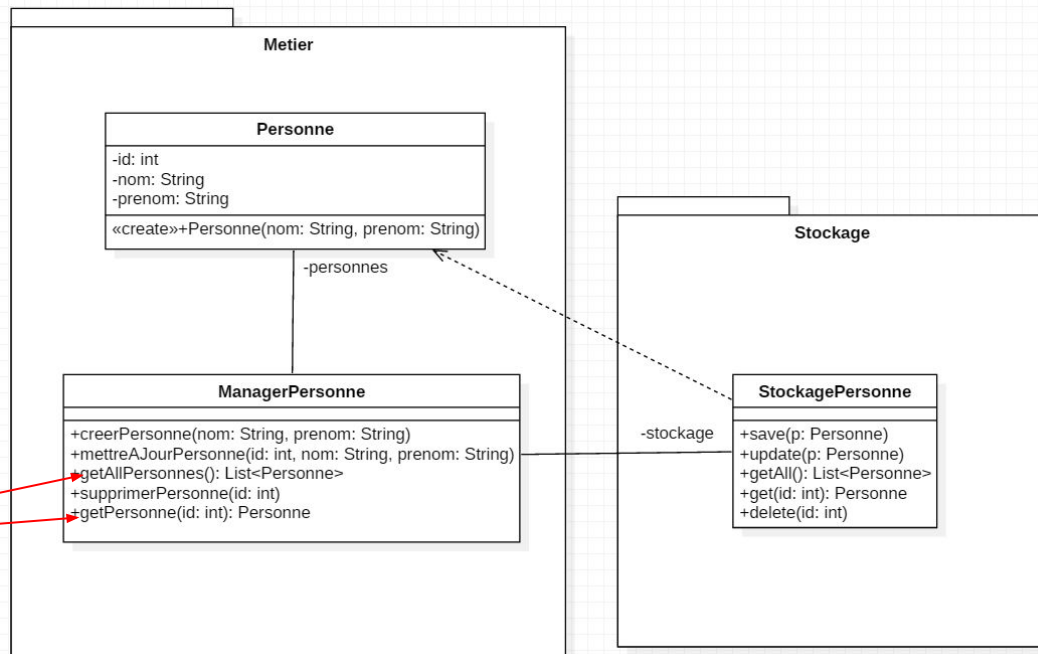
- Le **Lazy Loading** et le **Eager Loading** présentent des **avantages** et des **inconvénients**
- Le **Eager Loading** peut **alourdir le temps de lancement** de l'application, néanmoins, par la suite, les **opérations de lecture seront plus rapides** et moins coûteuses (car faites en locales)
- Le **Eager Loading** a aussi besoin de bien **synchroniser la version locale des objets gérés avec la source des données** (par exemple, si un autre programme modifie la BDD...)
- Le **Lazy Loading** est généralement **plus facile à implémenter** (pas de collections d'objets à gérer en local). Néanmoins, le fait qu'on passe tout le temps par la source (notamment pour la lecture) **peut réduire les performances** de l'application
- La **stratégie** à adopter dépend donc de différents paramètres à évaluer, pour chaque entité : nombres d'entités, fréquence de lecture de ces données, etc...

Lazy loading VS Eager Loading

- La liste “**personnes**” est remplie depuis la source de données lors de l'initialisation du programme
- Il faudrait également une méthode de synchronisation avec la source de données
- Les opérations de création, mise à jour et suppression modifient la liste et mettent à jour la source de données

Utilisent la **liste** locale
“**personnes**” au lieu de faire
une requête à la base

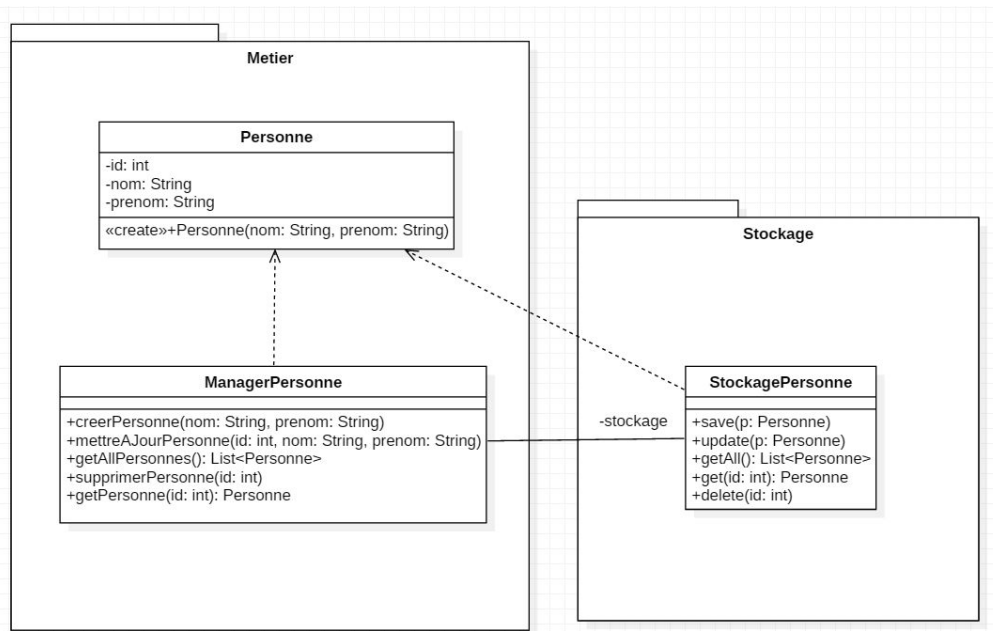
Eager Loading



Lazy loading VS Eager Loading

- Il n'y a pas de liste d'objets
"Personne" géré par le manager
- A **chaque lecture**, on va lire dans la source de données
- Les opérations de création, mise à jour et suppression mettent à jour la source de données

Lazy Loading



Parenthèse - Gradle

- Gradle est un **moteur de production** qui permet facilement de gérer un projet incluant diverses dépendances externes
- Un fichier nommé **build.gradle** permet de lister les adresses des **dépendances** qui devront être téléchargées dans l'environnement de développement
- Ce fichier permet également de définir plus amplement comment l'exécutable (**.jar**) du programme est généré
- **Gradle** génère également différentes tâches qu'il est possible d'exécuter (génération du **.jar**, **tests**, génération de la **documentation**...)
- Il permet également de générer et gérer le **MANIFEST** qui permet, entre autres, de donner des informations sur votre application et d'indiquer quelle est la classe principal à exécuter lors de l'exécution du **.jar**.

Parenthèse - Gradle - Ajours de dépendances

Exemple de l'import de la librairie externe "json" via un bloc dans **build.gradle**

```
dependencies {  
    implementation 'org.json:json:20211205'  
}
```

Une fois les dépendances ajoutées (ou globalement, après toute modification du **build.gradle**)
il faut **synchroniser** le projet (via le bouton ci-dessous)

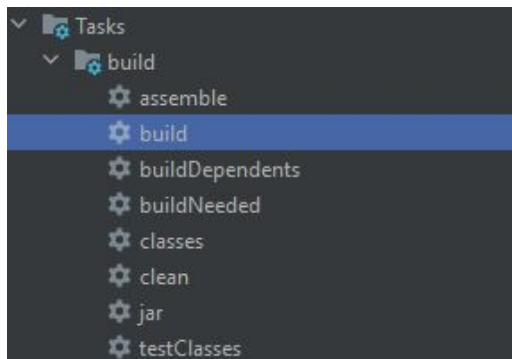


Parenthèse - Gradle - Export du .jar

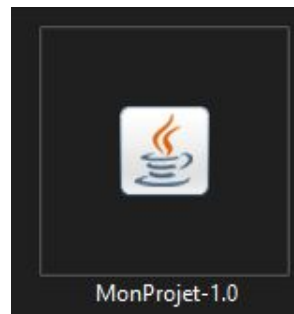
Via un bloc dans le fichier **build.gradle** (en remplaçant par votre **classe principale**)

```
jar {  
    manifest {  
        attributes( "Main-Class": "fr.iutmontpellier.project.Main" )  
    }  
}
```

Pour générer le **.jar** (panneau gradle à droite dans IntelliJ)



Dans **build/libs**



L'API JDBC

- **JDBC** (Java Database Connectivity) est une **API** (Application Programming Interface) qui permet de communiquer avec une **base de données relationnelle**
- **JDBC** est inclus de base dans Java, il n'y a rien à installer (sauf si vous voulez communiquer avec des bases de données qui utilisent une technologie particulière)
- **JDBC** permet d'écrire des **requêtes SQL** à envoyer à la base. On peut ainsi exécuter les différentes fonctionnalités de lecture, création mise à jour et suppression
- **JDBC** ne propose **pas de mapping automatique** des objets, on doit le faire "à la main" en extrayant chaque donnée obtenue en réponse à notre requête, ou bien en optimisant cela en faisant implémenter aux entités l'**interface SQLData**
- Cette **API** permet donc de réaliser la couche "**stockage**" d'une application, en utilisant n'importe quelle base de données relationnelle (MySQL, Oracle...) comme **source de données**

L'API JDBC - Installation d'un driver

- Pour se connecter à un type de base de données, **JDBC** doit utiliser un **driver**. Il faut donc en installer un selon le type de base utilisée dans votre projet
- Pour importer un driver, il suffit simplement d'ajouter une ligne dans le bloc **dependencies** au niveau du fichier **build.gradle** de votre application. Voici un exemple avec le driver **Oracle**:

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'  
    implementation group: com.oracle.database.jdbc, name: 'ojdbc8', version:  
    '12.2.0.1'  
}
```


L'API JDBC - Paramétrage

- **JDBC** s'utilise au travers d'un objet appelé **Connection**
- Afin d'instancier cet objet, on doit spécifier divers **paramètres** :
 - Le **driver** utilisé (drivers différents selon le type de base, MySQL, Oracle, PostgreSQL...)
 - **L'adresse** (ip / url) de la base
 - Le **port** de connexion
 - Le **nom** de la base
 - Le **login** et le **mot de passe** pour accéder à la base (compte utilisé sur le SGBD)
- Grâce à ces informations, l'objet **Connection** peut être instancié et utilisé pour envoyer et traiter des **requêtes** vers la base
- Quand on a terminé les opérations qui nécessitent l'objet **Connection**, on le ferme avec sa méthode **close**

L'API JDBC - Exemple de paramétrage pour Oracle

```
public class SQLUtils {

    private static SQLUtils instance = null;

    private Connection connection;

    private SQLUtils() {
        String url = "jdbc:oracle:thin:@ip:port:nombase";
        String driver = "oracle.jdbc.driver.OracleDriver";
        String user = "login";
        String pass = "password";
        try {
            Class.forName(driver);
            connection = DriverManager.getConnection(url,user,pass);
        }
        catch(ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
public Connection getConnection() {
    return this.connection;
}

public static SQLUtils getInstance() {
    if(instance == null) {
        instance = new SQLUtils();
    }

    return instance;
}
}
```

L'API JDBC - Les objets manipulés

- Les requêtes peuvent s'exécuter au travers deux types d'objets : **Statement** ou **PreparedStatement** (nous verrons la différence entre les deux), tous deux obtenus via l'objet **Connection**.
- Une requête renvoie généralement un **ResultSet**. Il s'agit d'un objet similaire aux **curseurs** en **PL/SQL**. A chaque fois, on a accès à une ligne de résultat. On peut alors récupérer les données, par exemple, en utilisant leur **index** de positionnement dans le **SELECT** de la requête suivante. La méthode **next()** permet alors de passer à la ligne de résultat suivante. Elle renvoie un **booléen** qui renvoie **false** si il n'y a plus de données à traiter. Il est donc utile de vérifier ce résultat s'il y a plusieurs lignes de résultats à traiter, dans une boucle, par exemple.
- Les **Statement**, **PreparedStatement** et **ResultSet** doivent être **fermés** après utilisation (même en cas d'erreur). Pour cela on utilise un mécanisme de java appelé le **try-with-resources** qui consiste à définir les ressources qui devront être fermées dans une section spéciale du bloc **try** (délimité par des **parenthèses**). A l'issue du bloc **try/catch**, ces ressources sont alors **fermées automatiquement** (il faut pour cela que la classe de la ressource implémente l'interface **AutoCloseable**, c'est le cas-ici)

try-with-resources

```
try (  
    //Ressources à initialiser qui seront fermées à la fin du try/catch  
) {  
    ...  
}  
catch (...) {  
  
}
```


L'API JDBC - Mise en contexte

- Nous allons nous appuyer sur l'exemple suivant :
 - Une **personne** est définie par un **id** (int), un **nom** (String) et un **prénom** (String)
 - Une **ville** est définie par un **id** (int) et un **nom** (String)
 - Une **personne** habite dans une **ville** (association)
- Au niveau de la **BDD**, on a le **schéma relationnel** suivant :



Personne(id, nom, prenom, idVille#)

Ville(id, nom)

L'API JDBC - Mise en contexte

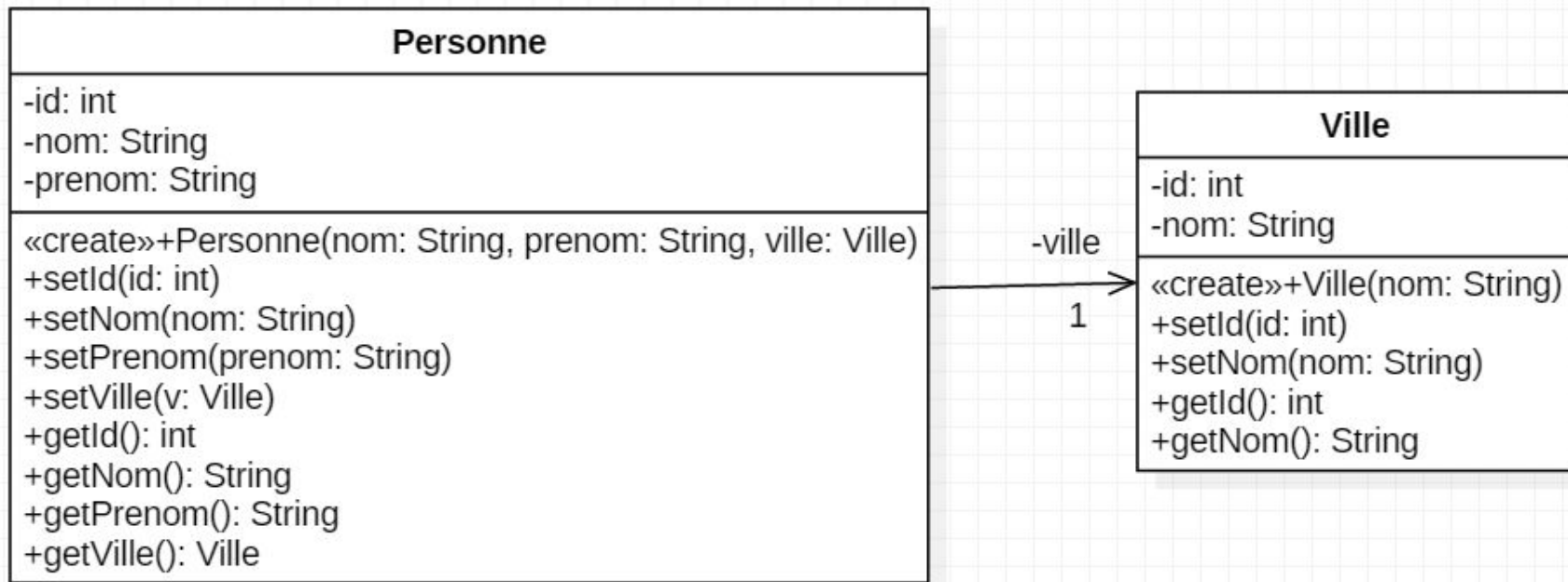
#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Extra
1	id 	int(11)			Non	Aucune	AUTO_INCREMENT
2	nom	varchar(50)	latin1_swedish_ci		Non	Aucune	

Avec MySQL

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Extra
1	id 	int(11)			Non	Aucune	AUTO_INCREMENT
2	nom	varchar(100)	latin1_swedish_ci		Non	Aucune	
3	prenom	varchar(100)	latin1_swedish_ci		Non	Aucune	
4	idVille 	int(11)			Non	Aucune	

Sous **Oracle**, l'option d'auto-incrémentation n'existe pas, on utilise des séquences et des curseurs à la place

L'API JDBC - Mise en contexte



L'API JDBC - Utilisation simple

- Voici un exemple simple permettant de récupérer une ville grâce à son id

```
public Ville getVille(int idVille) {
    Ville ville = null;
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String requete = "SELECT id, nom FROM ville WHERE idVille = " + idVille;
    try {
        Statement st = connection.createStatement();
        ResultSet result = st.executeQuery(requete) ;
    }
    {
        result.next() ; //On accède à la prochaine ligne
        int id = result.getInt(1);
        String nom = result.getString(2);
        ville = new Ville(nom) ;
        ville.setId(id) ;
    } catch (SQLException e) {
        e.printStackTrace() ;
    }
    return ville;
}
```


L'API JDBC - Utilisation simple

- Voici un exemple simple permettant de récupérer toutes les villes de la base :

```
public List<Ville> getVilles() {
    List<Ville> listeVilles = new ArrayList<>();
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String req = "SELECT id, nom FROM ville";
    try (
        Statement st = connection.createStatement();
        ResultSet result = st.executeQuery("SELECT id, nom FROM ville");
    ) {
        while(result.next()) {
            //On accède à la prochaine ligne
            int id = result.getInt(1);
            String nom = result.getString(2);
            Ville ville = new Ville(nom);
            ville.setId(id);
            listeVilles.add(ville);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return listeVilles;
}
```

L'API JDBC - Faille - Injection SQL

- Méthode simple mais qui présente une énorme **faille de sécurité : l'injection SQL**
- **L'injection SQL** est possible quand des données externes doivent être passées de la requête (un id, un nom, un login, un mot de passe...)
- Nous n'entrerons pas dans les détails mais en résumé, la faille consiste à passer des données sous la forme de commentaire sql ce qui annule une partie de la requête et permet aussi d'exécuter du code SQL non désiré initialement
- Cette faille permet de, par exemple, se connecter à un compte sans mot de passe ou bien d'envoyer des requêtes vers la base pour changer ses privilèges...

L'API JDBC - Sécurité - Requêtes préparées

- Il existe un mécanisme permettant de contrer cette faille : **les requêtes préparées**
- La requête est **préchargée** et les données externes sont **injectées** dans la requête au moment de l'exécution. Ainsi, on ne manipule plus le chaîne de caractère de la requête directement
- Pour cela, on utilise, avec JDBC, un objet appelé **PreparedStatement**
- On place des “?” dans la requête à l'endroit où les données devront être placées
- Par la suite, on indique quelle donnée placer à quelle endroit, en suivant l'ordre des “?” dans la requête. attention, **l'index** commence à **1!** (et non pas 0)

L'API JDBC - Récupérer une entité précise

```
public Ville getVille(int idVille) {
    Ville ville = null;
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String requete = "SELECT id, nom FROM ville WHERE idVille = ?" ;
    try (PreparedStatement statement = connection.prepareStatement(requete ,
ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);)
    {
        statement.setInt(1, idVille);
        try (ResultSet result = statement.executeQuery() ;)
        {
            result.next() ; //On accède à la prochaine ligne
            int id = result.getInt(1);
            String nom = result.getString(2);
            ville = new Ville(nom);
            ville.setId(id);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return ville;
}
```

L'API JDBC - Récupérer toutes les entités

```
public List<Ville> getVilles() {
    List<Ville> listeVilles = new ArrayList<>();
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String req = "SELECT id, nom FROM ville";
    try (
        PreparedStatement statement = connection.prepareStatement(req, ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        ResultSet result = statement.executeQuery();
    ) {
        while(result.next()) {
            int id = result.getInt(1);
            String nom = result.getString(2);
            Ville ville = new Ville(nom);
            ville.setId(id);
            listeVilles.add(ville);
        }

    } catch (SQLException e) {
        e.printStackTrace();
    }

    return listeVilles;
}
```

L'API JDBC - Créer une entité

```
public void createVille(Ville ville) {
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String req = "INSERT INTO ville (nom) VALUES (?)";
    try (PreparedStatement statement = connection.prepareStatement(req);)
    {
        statement.setString(1, ville.getNom());
        statement.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

L'API JDBC - Mettre à jour une entité

```
public void updateVille(Ville ville) {
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String req = "UPDATE ville SET nom = ? WHERE id = ?";
    try (PreparedStatement statement = connection.prepareStatement(req);)
    {
        statement.setString(1, ville.getNom());
        statement.setInt(2, ville.getId());

        statement.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

L'API JDBC - Supprimer une entité

```
public void deleteVille(int idVille) {  
    SQLUtils utils = SQLUtils.getInstance();  
    Connection connection = utils.getConnection();  
    String req = "DELETE FROM ville WHERE id = ?";  
    try (PreparedStatement statement = connection.prepareStatement(req);)  
    {  
        statement.setInt(1, idVille);  
  
        statement.executeUpdate();  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
}
```


Lien entre la couche métier et stockage

- On va définir un **ManagerVille** qui permettra de gérer les différentes villes
- La stratégie adoptée sera le **Lazy Loading**
- Toutes les méthodes vues dans les diapositives précédentes sont placées dans une classe “**StockageVille**” dans la couche stockage
- On va créer différentes **services** (méthodes) permettant à la couche application de manipuler les villes de l'application

Lien entre la couche métier et stockage

```
public class ManagerVille {  
  
    private StockageVille stockage = new StockageVille();  
  
    public void creerVille(String nom) {  
        Ville ville = new Ville(nom);  
        stockage.createVille(ville);  
    }  
  
    public void mettreAJourVille(int idVille, String nom) {  
        Ville ville = this.stockage.getVille(idVille);  
        ville.setNom(nom);  
        this.stockage.updateVille(ville);  
    }  
}
```

```
    public List<Ville> getListeVilles() {  
        return stockage.getVilles();  
    }  
  
    public Ville getVille(int idVille) {  
        return this.stockage.getVille(idVille);  
    }  
  
    public void supprimerVille(int idVille) {  
        stockage.deleteVille(idVille);  
    }  
}
```

Lien entre la couche application et métier

- On pourrait imaginer, par exemple, qu'un **controller** de la couche **application** va faire le lien entre la **liste des villes** et une **liste déroulante** (pour faire un choix) de l'IHM

```
public class ControllerVilles {  
  
    @FXML  
    //Liste déroulante  
    private ComboBox<Ville> choixVille;  
  
    private ManagerVille manager;  
  
    @FXML  
    public void initialize() {  
        List<Ville> villes = manager.getListeVilles();  
  
        //Remplit la liste déroulante  
        choixVille.setItems(FXCollections.observableArrayList(villes));  
    }  
  
}
```

L'API JDBC - Personne

- Nous allons maintenant voir comment implémenter le stockage de **Personne**
- Là aussi, on retrouve les notions de **Eager Loading** et de **Lazy Loading** lors du chargement des données qui possèdent des **associations**
- **Personne** possède une relation avec **Ville**. Lors du chargement, on doit faire un choix :
 - Quand on charge une personne, on charge les données de la ville qui lui est associée (**Eager Loading**)
 - On charge seulement la personne, et on chargera les données de la ville plus tard, quand on en aura besoin, via le manager. Il faudra alors rajouter une méthode du style “**getVillePersonne**” dans le **stockage** et le **manager** des villes (**Lazy Loading**)
- Pour notre implémentation, nous opterons plutôt pour une stratégie de **Eager Loading**

L'API JDBC - Récupérer une personne - Eager loading

```
public Personne getPersonne(int idPersonne) {
    Personne personne = null;
    Connection connection = SQLUtils.getInstance().getConnection();
    String requete = "SELECT P.id, P.nom, P.prenom, V.id, V.nom FROM personne P JOIN Ville V on V.id = P.idVille WHERE idPersonne = ?";
    try (PreparedStatement statement = connection.prepareStatement(requete, ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_READ_ONLY);)
    {
        statement.setInt(1, idPersonne);
        try (ResultSet result = statement.executeQuery();)
        {
            result.next();
            int id = result.getInt(1);
            String nom = result.getString(2);
            String prenom = result.getString(3);
            int idVille = result.getInt(4);
            String nomVille = result.getString(5);
            Ville ville = new Ville(nomVille);
            ville.setId(idVille);
            personne = new Personne(nom, prenom, ville);
            personne.setId(id);
        }
    } catch (SQLException e) {e.printStackTrace();}
    return personne;
}
```

L'API JDBC - Création d'une personne

```
public void createPersonne(Personne personne) {
    SQLUtils utils = SQLUtils.getInstance();
    Connection connection = utils.getConnection();
    String req = "INSERT INTO personne (nom, prenom, idVille) VALUES (?, ?, ?)";
    try (PreparedStatement statement = connection.prepareStatement(req);)
    {
        statement.setString(1, personne.getNom());
        statement.setString(2, personne.getPrenom());
        statement.setInt(3, personne.getVille().getId());

        statement.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

L'API JDBC - ManagerPersonne

```
public class ManagerPersonne {  
  
    private StockagePersonne stockage = new StockagePersonne();  
  
    //Initialisé autre part...  
    private ManagerVille managerVille;  
  
    public void creerPersonne(String nom, String prenom, int idVille) {  
        Ville v = managerVille.getVille(idVille);  
        Personne p = new Personne(nom, prenom, v);  
        this.stockage.createPersonne(p);  
    }  
  
    public Personne getPersonne(int id) {  
        return this.stockage.getPersonne(id);  
    }  
  
}
```

Les ORM

- Un **ORM** (Object-Relational Mapping) est un outil qui va simplifier la **communication** avec une **base de données**
- On donne directement les informations utiles pour la base de données au niveau des **classes** (entités) avec des **annotations**
- La **création** et la **gestion** de la base de données et des tables est **automatique**
- Simplification de l'accès aux fonctionnalités **CRUD** pour chaque entité, sans écrire beaucoup de code
- Le **mapping** des objets est fait **automatiquement**
- S'utilise généralement à travers un **framework** (par exemple, **Spring** pour Java)
- **L'ORM** le plus connu de **java** est **Hibernate**

L'ORM Hibernate - Installation

- Pour installer **Hibernate**, on l'importe simplement dans le bloc **dependencies** au niveau du fichier **build.gradle**
- Il faut également importer un (ou plusieurs) **driver** selon le type de base utilisée (pareil que pour JDBC). Ici, on importe **Hibernate** et un **driver** pour les bases de données **Oracle** :

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'  
    implementation 'org.hibernate:hibernate-core:5.6.5.Final'  
    implementation group: com.oracle.database.jdbc, name: 'ojdbc8', version:  
'12.2.0.1'  
}
```

L'ORM Hibernate - Configuration de l'ORM

- **Hibernate** se configure à l'aide d'un fichier **hibernate.cfg.xml** placé à la racine du dossier **resources**
- Comme pour **JDBC**, on doit aussi indiquer :
 - Le **driver** utilisé (drivers différents selon le type de base, MySQL, Oracle, PostgreSQL...)
 - **L'adresse** (ip / url) de la base (en précisant bien le nom de la base utilisé)
 - Le **port** de connexion
 - Le **nom** de la base
 - Le **login** et le **mot de passe** pour accéder à la base (compte utilisé sur le SGBD)
 - Le **dialect** est une classe "pont" qui correspond à la version du SGBD utilisé, ce qui permet à Hibernate de générer efficacement les requêtes SQL adéquates.
- En plus de cela, on doit préciser à **Hibernate** quelles classes correspondent aux entités que l'on souhaite stocker dans la base

L'ORM Hibernate - Configuration de l'ORM

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd" >
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:oracle:thin:@ip:port:nombase </property>
    <property name="connection.username">login</property>
    <property name="connection.password">password</property>
    <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver </property>
    <property name="dialect">org.hibernate.dialect.Oracle10gDialect </property>

    <property name="format_sql">true</property>
    <property name="hbm2ddl.auto">update</property>

    <property name="connection.pool_size">5</property>
    <property name="current_session_context_class">thread</property>

    <mapping class="com.exemple.projet.metier.entite.Personne" />
  </session-factory>
</hibernate-configuration>
```

L'ORM Hibernate - Configuration d'une entité

- Pour configurer une **entité** (classe) on utilise différentes **annotations** :
 - **@Entity** puis **@Table(name = "nom")** : se placent au niveau de la déclaration de la classe. Le nom indiqué dans **@Table** correspond au nom de la table qui sera généré dans la base
 - **@Id** : se place au-dessus de l'attribut qui correspond à la clé primaire de la table. Si cet attribut est numérique, on peut en plus utiliser les deux annotations :
@GeneratedValue(name="nomCustom", strategy = "increment")
@GeneratedValue(generator="nomCustom")
si l'on souhaite que la clé primaire s'incrémente automatiquement
 - **@Column** : se place aux niveau des autres attributs de la classe (qui ne sont pas d'autres entités de l'application). On peut également préciser un nom si besoin (si on veut qu'il y ai un nom différent entre l'attribut dans la classe et dans la table)
- Il faut **impérativement** que la classe définisse un **constructeur vide** (elle peut avoir en plus d'autres constructeurs, bien entendu)
- Il faut bien penser à l'ajouter au **fichier de configuration** d'hibernate

L'ORM Hibernate - Annotations de clé primaire

- Comme nous l'avons vu dans le slide précédent, on utilise l'annotation **@Id** pour préciser une clé primaire.
- Si la clé primaire est **composée d'un seul attribut**, il est donc possible de définir une **stratégie d'affectation automatique** de cette clé pour ne pas à avoir à la spécifier lors de l'insertion d'un nouveau tuple dans la table.
- Pour cela, on utilise les deux annotations :

@GenericGenerator(name="nomCustom", strategy = "strategie")

@GeneratedValue(generator="nomCustom")

- **"nomCustom"** est un nom unique que vous donnez à votre générateur (en rapport avec le nom de la table, par exemple). Il faut que cela soit le même dans les deux annotations (la première définit le générateur, la seconde l'utilise)
- **"strategie"** correspond à la stratégie de génération à adopter, par exemple : **increment** (auto-incrémentation), **sequence**, **identity**...(nous utiliserons principalement **increment**)
- Si la clé primaire est **composée de plusieurs attributs**, on spécifie simplement l'annotation **@Id** au dessus de chacun des attributs (on ne doit pas définir de stratégie automatique d'affectation dans ce cas)

L'ORM Hibernate - Configuration d'une entité

```
@Entity
@Table(name = "Ville")
public class Ville {

    @Id
    @GenericGenerator(name="villesAuto", strategy = "increment")
    @GeneratedValue(generator="villesAuto")
    private int id;

    @Column
    private String nom;

    public Ville(String nom) {this.nom = nom;}

    public Ville() {}

    public int getId() {return id;}

    public void setId(int id) {this.id = id;}

    public String getNom() {return nom;}

    public void setNom(String nom) {this.nom = nom;}

}
```

```
. . .
<hibernate-configuration>
    <session-factory>
        . . .
        <mapping
class="com.exemple.projet.metier.entite.Ville"/>
        . . .
    </session-factory>
</hibernate-configuration>
```

L'ORM Hibernate - Liens entres entités

- Il est possible d'utiliser des **annotations spéciales** pour créer des liens entre les entités
- Cela permet à la base de gérer des **associations** en créant les **clés étrangères** (et tables éventuelles) adéquates
- Pour l'entité, cela permet de réaliser (si besoin) du **eager loading** pour charger automatiquement des données qui lui sont liées
- Il existe 4 types de **relations** qui correspondent aux **cardinalités** de l'association :

@OneToOne : Relation aux cardinalités 1 de chaque côté (l'attribut correspond à une seule entité)

@ManyToOne : Relation aux cardinalités * et 1 (l'attribut correspond donc à une seule entité)

@OneToMany : Relation aux cardinalités 1 et * (l'attribut correspond donc à une liste d'entités)

@ManyToMany : Relation aux cardinalités * et * (l'attribut correspond donc à une liste d'entités)

L'ORM Hibernate - Liens entres entités - ManyToOne

```
@Entity
@Table (name = "Personne")
public class Personne {

    @Id
    @GenericGenerator (name="personnesAuto", strategy = "increment")
    @GeneratedValue (generator="personnesAuto")
    private int id;

    @Column
    private String nom;

    @Column
    private String prenom;

    @ManyToOne
    @JoinColumn (name="idVille")
    @OnDelete (action = OnDeleteAction.CASCADE)
    private Ville ville;

    public Personne (String nom, String prenom, Ville ville) {
        this.nom = nom;
        this.prenom = prenom;
        this.ville = ville;
    }

    public Personne () {

    }

}
```

```
@ManyToOne
@JoinColumn (name="idVille")
@OnDelete (action = OnDeleteAction.CASCADE)
private Ville ville;
```

- Dans **@JoinColumn**, on place le nom de la clé étrangère qui sera générée
- On peut également préciser l'action à effectuer si la valeur référencée est supprimée (cascade = si la ville est supprimée, la personne sera supprimée)

L'ORM Hibernate - Liens entres entités - OneToOne

```
@Entity
@Table(name = "Chat")
public class Chat {

    @Id
    @GenericGenerator(name="chatsAuto",
strategy = "increment")
    @GeneratedValue(generator="chatsAuto")
    private int id;

    @Column
    private String nom;

    @OneToOne
    @JoinColumn(name = "idMaitre")
    @OnDelete(action = OnDeleteAction.CASCADE)

    private Personne maitre;
}
```

```
@Entity
@Table(name = "Personne")
public class Personne {

    . . .

    @OneToOne
    @JoinColumn(name="idChat")
    private Chat chat;

    . . .
}
```

- Une personne a un seul chat et un chat a un seul maître
- Si le maître est supprimé, le chat est supprimé

L'ORM Hibernate - Liens entres entités - OneToMany

```
@Entity
@Table(name = "Ville")
public class Ville {

    @Id
    @GenericGenerator(name="villesAuto", strategy =
"increment")
    @GeneratedValue(generator="villesAuto")
    private int id;

    @Column
    private String nom;

    @OneToMany(mappedBy="ville", fetch = FetchType.EAGER,
, cascade = CascadeType.ALL)
    private List<Personne> habitants = new
ArrayList<>();
}
```

- Au niveau de **mappedBy**, on indique le nom de l'attribut qui fait le lien dans l'autre classe (ici, "**ville**" dans **Personne**)
- Ici, **Personne** doit donc avoir une relation **@ManyToOne** avec ville définie auparavant
- Stratégie : **Eager**. Les données des habitants seront chargées quand une ville est chargée.

L'ORM Hibernate - Liens entres entités - ManyToMany

```
@Entity
@Table(name = "Personne")
public class Personne {
    . . .

    @ManyToMany(cascade = { CascadeType.ALL }, fetch =
FetchType.EAGER)
    @JoinTable(
        name = "Personne Club",
        joinColumns = { @JoinColumn(name = "idPersonne") },
        inverseJoinColumns = { @JoinColumn(name = "idClub") }
    )
    private List<Club> clubs;

    . . .
}
```

- Dans **name**, on indique le nom de la table qui sera créé dans la BDD pour gérer l'association
- Dans **joinColumns** et **inverseJoinColumns**, on indique le noms des deux clés primaires (et à la fois étrangères) qui composeront la table

L'ORM Hibernate - Liens entres entités - ManyToMany

```
@Entity
@Table(name = "Club")
public class Club {

    @Id
    @GenericGenerator (name="clubsAuto", strategy =
"increment")
    @GeneratedValue (generator="clubsAuto")
    private int id;

    @Column
    private String nomClub;

    @ManyToMany (mappedBy = "clubs", cascade = CascadeType.ALL,
fetch = FetchType.EAGER)
    private List<Personne> membres;
}
```

- De l'autre côté, dans **mappedBy**, on indique simplement le nom de l'attribut qui fait le lien de l'autre côté (comme pour **OneToMany**)

L'ORM Hibernate - Utilisation

- Une fois les **entités** configurées (et correctement ajoutées dans le fichier de configuration) au lancement, **la base se créera toute seule!**
- Si jamais vous modifiez plus tard les entités, la base et les tables se mettront à jour automatiquement également
- Néanmoins, l'utilisation "native" de **Hibernate** sans framework est un peu complexe, pour manipuler directement la base de données
- Généralement, dans un framework fournit des classes appelées **repository** (équivalent aux classes de "**stockage**") qui permettent de facilement interagir avec **l'ORM**. Habituellement, les opérations **CRUD** élémentaires sont très facilement accessibles sans avoir à fournir beaucoup de code et il est facile d'étendre un **repository** afin d'ajouter des fonctionnalités plus spécifiques
- Dans le cadre de la 1ère année, vous utiliserez une petite **API** développée pour ce cours vous permettant d'utiliser facilement **Hibernate** sans avoir à installer de framework supplémentaire

L'API Hibernate Repositories

- L'API **Hibernate Repositories** permet de facilement interagir avec **Hibernate** au travers de **repositories** simples
- Pour n'importe quelle entité, il est possible, en une seule ligne de code, de récupérer un **repository** implémentant les opérations **CRUD**
- Il est également possible de créer ses propres classes de **repositories**



L'API Hibernate Repositories - Installation

- Comme **Hibernate Repositories** est une **librairie privée** développée pour ce cours, il faut l'importer de manière locale
- Pour cela, on peut créer un dossier "**libs**" à la **racine du projet** et placer le fichier **.jar** à l'intérieur
- Au niveau du **build.gradle**, il suffit alors d'importer la librairie comme cela :

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.1'  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.1'  
    implementation 'org.hibernate:hibernate-core:5.6.5.Final'  
    implementation group: com.oracle.database.jdbc, name: 'ojdbc8', version:  
    '12.2.0.1'  
    implementation files('libs/HibernateRepositories-1.0.jar')  
}
```

L'API Hibernate Repositories - Repository

- Pour récupérer un **repository** (classe de “**stockage**”) il suffit d'utiliser le code suivant :

```
EntityRepository<NomEntite> repository = RepositoryManager.getRepository(NomEntite.class);
```

- Il faut donc simplement remplacer “**NomEntite**” par l'entité dont vous voulez obtenir la classe gérant le stockage, par exemple :

```
EntityRepository<Personne> repository = RepositoryManager.getRepository(Personne.class);
```


L'API Hibernate Repositories - Repository

- De base, un **repository** possède les méthodes suivantes :
 - **findAll** : renvoie une liste de tous les entités du type géré par le repository
 - **findById** : renvoie l'entité correspondante à l'identifiant passé en paramètre
 - **create** : crée l'entité passée en paramètre au niveau de la base
 - **update** : met à jour l'entité passée en paramètre au niveau de la base
 - **delete** : supprime l'entité passée en paramètre au niveau de la base
 - **deleteById** : supprime l'entité correspondante à l'identifiant en paramètre au niveau de la base

L'API Hibernate Repositories - Repository

```
public class ManagerVille {  
  
    private EntityRepository<Ville> repository =  
RepositoryManager.getRepository(Ville.class);  
  
    public void creerVille (String nom) {  
        Ville ville = new Ville(nom);  
        this.repository.create(ville);  
    }  
  
    public void updateVille (int idVille, String nom)  
{  
        Ville ville = repository.findById(idVille);  
        ville.setNom(nom);  
        this.repository.update(ville);  
    }  
  
}
```

```
public List<Ville> getListeVilles () {  
    return this.repository.findAll();  
}  
  
public Ville getVille (int idVille) {  
    return this.repository.findById(idVille);  
}  
  
public void supprimerVille (int idVille) {  
    this.repository.deleteById(idVille);  
}
```

L'API Hibernate Repositories - Repository

```
public class ManagerPersonne {  
  
    private EntityRepository<Personne> repository = RepositoryManager.getRepository(Personne.class);  
  
    //Initialisé autre part...  
    private ManagerVille managerVille;  
  
    public void creerPersonne(String nom, String prenom, int idVille) {  
        Ville v = managerVille.getVille(idVille);  
        Personne p = new Personne(nom, prenom, v);  
        this.repository.update(p);  
    }  
  
    public Personne getPersonne(int id) {  
        return this.repository.findById(id);  
    }  
  
}
```

L'API Hibernate Repositories - Repository custom

- Il est possible de créer son **propre repository** afin d'ajouter d'autres fonctionnalités
- il suffit d'étendre la classe **EntityRepositoryDatabase** en précisant le type d'entité géré
- Les fonctionnalités du **CRUD** sont héritées (il est possible de les redéfinir)

```
public class CustomPersonneRepository extends EntityRepositoryDatabase<Personne> {  
  
    public CustomPersonneRepository () {  
        super(Personne.class);  
    }  
  
    public List<Personne> findPersonnesHabitantANevers () {  
        . . .  
    }  
  
}
```

L'API Hibernate Repositories - Repository custom

```
public class ManagerPersonne {  
  
    private CustomPersonneRepository repository = new CustomPersonneRepository() ;  
  
    //Initialisé autre part...  
    private ManagerVille managerVille ;  
  
    public void creerPersonne (String nom, String prenom, int idVille) {  
        Ville v = managerVille.getVille(idVille) ;  
        Personne p = new Personne(nom, prenom, v) ;  
        this.repository.update(p) ;  
    }  
  
    public Personne getPersonne (int id) {  
        return this.repository.findById(id) ;  
    }  
  
    public List<Personne> getNivernais () {  
        return this.repository.findPersonnesHabitantANevers () ;  
    }  
}
```