Introduction à JavaFX

A travers la création d'une application « TodoList »

Sommaire

Intégration de MySQL au sein d'un projet JAVA

Nous allons intégrer MySQL à notre projet JavaFX en installant JDBC via Maven et en créant une classe Database.java pour gérer la connexion à la base de données (comme en PHP finalement...).

Objectif : Ajouter JDBC et gérer la connexion MySQL

- Fichiers concernés :
  - pom.xml (Ajout de la dépendance JDBC)
  - src/main/java/database/Database.java (Gestion de la connexion)
- JDBC pour : Java DataBase Connector/Connectivity

Installation de JDBC via Maven

Pour connecter Java à MySQL, nous devons ajouter MySQL Connector/J à notre projet.

Étape 1 : Ajouter la dépendance dans pom.xml

- Ouvrez le fichier pom.xml à la racine du projet.
- Ajoutez la dépendance suivante dans la section <dependencies> :

```
<dependency>
   <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <version>9.2.0</version>
</dependency>
```

Actualisez Maven : cliquez sur l'icône de synchronisation (ou exécutez mvn clean install).

Vérifiez que la dépendance est bien prise en compte (les erreurs en rouge doivent disparaître).

Pourquoi utiliser Maven ?

- Automatisation de l'installation des bibliothèques.
- Gestion simplifiée des versions et mises à jour.
- Portabilité

Comment trouver les dépendances à installer ?

- Vous pouvez vous rendre sur le site se nommant « Mavenrepository »
  - Lien :
- Grâce à la barre de recherche, trouver la dernière version de votre dépendant à ajouter dans votre pom.xml
  - Exemple avec notre connecteur : tapez « Mysql Connector »
  - Vous allez retrouver plusieurs versions possibles mais qui correspondent à des utilisations autres que celle que l'on veut.
  - Pour trouver la dépendance qui correspond, internet est votre ami (Ou Chatg...)

Créer un package database et un fichier Database.java

Nous allons maintenant créer une classe qui permettra d'ouvrir et récupérer la connexion à la base de données.

Etape 2 : Créer le package et la classe

- Dans src/main/java, créez un package database.
- Dans database, créez la classe Database.java.

Étape 3 : Ajouter les attributs de connexion

Dans Database.java, ajoutez les informations nécessaires pour se connecter à la base :

```
package database;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class Database {
    private static final String SERVEUR = "localhost";
    private static final String NOM_BDD = "nom_de_la_base";
    private static final String UTILISATEUR = "root";
    private static final String MOT_DE_PASSE = "";

    private static String getUrl() {
        return "jdbc:mysql://" + SERVEUR + "/" + NOM_BDD + "?serverTimezone=UTC";
    }

    public static Connection getConnexion() {
        Connection cnx = null;
        try {
            cnx = DriverManager.getConnection(getUrl(), UTILISATEUR, MOT_DE_PASSE);
            System.out.println("Connexion réussie à la base de données !");
        } catch (SQLException e) {
            System.out.println("Erreur de connexion : " + e.getMessage());
        }
        return cnx;
    }
}
```

Explication du code

Elément	Rôle
`SERVEUR`	Adresse du serveur MySQL (par défaut : `localhost`).
`NOM_BDD`	Nom de la base de données à laquelle se connecter.
`UTILISATEUR`	Identifiant MySQL (par défaut : `root`).
`MOT_DE_PASSE`	Mot de passe MySQL (laisser vide si aucun mot de passe n'est défini).
`getUrl()`	Construit l'URL de connexion JDBC.
`getConnexion()`	Etablit une connexion à MySQL et affiche un message en console.
`try/catch`	Gère les erreurs de connexion et affiche un message d'erreur si nécessaire.

Test de la connexion

Nous allons maintenant tester si la connexion fonctionne.

Étape 4 : Ajouter un main pour tester

Ajoutez ce code temporaire dans Database.java :

```
public static void main(String[] args) {
    Connection cnx = getConnexion();
    if (cnx != null) {
        System.out.println("Connexion établie avec succès !");
    } else {
        System.out.println("Échec de la connexion à la base de données.");
    }
}
```

Étape 5 : Lancer le test

Exécutez le main qui se trouve dans votre fichier Database.java

Résultat attendu en console :

- Succès : "Connexion réussie à la base de données !"
- Échec : Message d'erreur détaillé.

Si la connexion échoue :

- Vérifiez que MySQL (WAMP) est bien démarré sur vos machines.
- Vérifiez que le nom de la base est correct.
- Vérifiez que les identifiants (root, mot de passe) sont exacts.

Résumé des tâches à effectuer

1. Ajouter la dépendance JDBC à pom.xml et l'installer.
2. Créer le package database et la classe Database.java.
3. Configurer les informations de connexion (serveur, base, utilisateur, mot de passe).
4. Tester la connexion en exécutant Database.java.

Création du modèle Utilisateur.java et de son repositoy

Nous allons maintenant créer la classe Utilisateur.java, qui représente la table des utilisateurs dans la base de données.

Objectif : Définir une classe Utilisateur en lien avec la base

Fichier concerné :

- src/main/java/model/Utilisateur.java
- src/main/java/repository/UtilisateurRepository.java

Création du package et du fichier Utilisateur.java

Étape 1 : Créer le package model

- Dans src/main/java, créez un package nommé « model ».
- Ajoutez-y un fichier Utilisateur.java.

Le package model aura pour objectif de contenir l'ensemble de nos classes en lienavec la base de données.

Étape 2 : Compléter le fichier Utilisateur.java

Compléter la classe utilisateur.java avec les informations venant de ce schéma UML :

Vous devez :

- Créer les attribus
- Générer les getter/setter
- Créer trois constructeurs
  - Un avec l'ensemble des informations – Créer un utilisateur complet
  - Un avec l'ensemble des informations excepté l'id – Pour l'inscription
  - Un avec seulement l'email et le mot de passe – Pour la connexion
- Ajouter une méthode toString

Pourquoi passer l'ID en paramètre ?

- L'ID est géré par la base de données (auto-incrémenté).
- Mais on peut en avoir besoin si on récupère un (ou plusieurs) utilisateur(s) existant.

Pourquoi utiliser des getters et setters ? (rappel)

- Encapsulation : on protège l'accès direct aux attributs.
- Flexibilité : on peut ajouter des règles de validation si nécessaire.

Pourquoi toString() est utile ? (rappel)

- Permet d'afficher facilement un objet utilisateur en console.
- Utile pour le debug et le suivi des requêtes.

Résumé des tâches à effectuer

- Créer le package model et ajouter Utilisateur.java.
- Déclarer les attributs de la classe (id, nom, prénom, email, mdp, rôle).
- Ajouter un constructeur prenant en compte tous les attributs.
- Générer les getters et setters pour chaque attribut.
- Ajouter une méthode toString() pour faciliter l'affichage.

Création du Repository UtilisateurRepository.java

Nous allons maintenant créer une classe UtilisateurRepository qui va gérer les interactions entre notre application et la base de données.

Étape 1 : Créer le package repository

- Dans src/main/java, créez un package repository.
- Ajoutez-y un fichier UtilisateurRepository.java.

Étape 2 : Ajouter l'importation de la connexion

- Nous allons utiliser la connexion MySQL définie dans Database.java.
    - Créer un attribut privé de type Connection
    - Créer un constructeur avec aucun paramètre permettant de récupérer la connexion à la base de données
        - Grâce à la méthode présente dans Database.java

Pourquoi utiliser une connexion unique ?

- Évite d'ouvrir plusieurs connexions inutiles.
- Facilite l'exécution des requêtes SQL.

Étape 3 : Ajouter une méthode pour insérer un utilisateur

```
public void ajouterUtilisateur(Utilisateur utilisateur) {
    String sql = "INSERT INTO utilisateurs (nom, prenom, email, mdp, role) VALUES (?, ?, ?, ?,
?)";
    try {
        PreparedStatement stmt = connexion.prepareStatement(sql);
        stmt.setString(1, utilisateur.getNom());
        stmt.setString(2, utilisateur.getPrenom());
        stmt.setString(3, utilisateur.getEmail());
        stmt.setString(4, utilisateur.getMdp());
        stmt.setString(5, utilisateur.getRole());
        stmt.executeUpdate();
        System.out.println("Utilisateur ajouté avec succès !");
    } catch (SQLException e) {
        System.out.println("Erreur lors de l'ajout de l'utilisateur : " + e.getMessage());
    }
}
```

Pourquoi utiliser un PreparedStatement ?

- Protège contre les injections SQL.
- Optimise la performance des requêtes.

Ce que fait cette méthode :

- Elle prépare une requête SQL avec des ? pour éviter les injections SQL.
- Elle affecte chaque valeur de l'objet Utilisateur aux paramètres SQL.
- Elle exécute la requête pour insérer l'utilisateur en base.
- En cas d'erreur, elle affiche un message en console.

Étape 4 : Compléter UtilisateurRepository en ajoutant des méthodes essentielles

Méthode fournie : Ajouter un utilisateur

À vous de jouer ! Complétez ces méthodes :

Récupérer un utilisateur par email

Objectif : Écrire une méthode qui récupère un utilisateur spécifique en fonction de son email.

Aide :

- Utilisez une requête SELECT * FROM utilisateurs WHERE email = ?.
- Remplacez-le ? par l'email fourni en paramètre.
- Retournez un objet Utilisateur avec les données récupérées.

```
public Utilisateur getUtilisateurParEmail(String email) {
    // À implémenter
}
```

Récupérer la liste de tous les utilisateurs

Objectif : Écrire une méthode qui renvoie une liste de tous les utilisateurs présents en base.

Aide :

- Utilisez une requête SELECT * FROM utilisateurs.
- Parcourez les résultats (ResultSet) pour créer une liste d'objets Utilisateur.
- Retournez cette liste.

```
public ArrayList<Utilisateur> getTousLesUtilisateurs() {
    // À implémenter
}
```

Supprimer un utilisateur par email

Objectif : Écrire une méthode qui supprime un utilisateur spécifique via son email.

Aide :

- Utilisez une requête DELETE FROM utilisateurs WHERE email = ?.
- Exécutez la requête et affichez un message confirmant la suppression.

```
public void supprimerUtilisateurParEmail(String email) {
    // À implémenter
}
```

Mettre à jour les informations d'un utilisateur

Objectif : Écrire une méthode qui permet de modifier un utilisateur existant.

Aide :

- Utilisez une requête UPDATE utilisateurs SET nom = ?, prenom = ?, mdp = ?, role = ? WHERE email = ?.
- Assurez-vous que les valeurs sont bien mises à jour.

```
public void mettreAJourUtilisateur(Utilisateur utilisateur) {
    // À implémenter
}
```

Tâches à effectuer

- Compléter la méthode getUtilisateurParEmail() pour récupérer un utilisateur.
- Écrire la méthode getTousLesUtilisateurs() pour obtenir la liste des utilisateurs.
- Implémenter supprimerUtilisateurParEmail() pour permettre la suppression d'un utilisateur.
- Ajouter mettreAJourUtilisateur() pour modifier les informations d'un utilisateur.

Conseil : Testez chaque méthode en affichant les résultats en console avant de les utiliser dans l'interface graphique.

Une fois ces méthodes complétées, vous serez prêts à connecter votre base de données à votre interface JavaFX !

Connexion avec l'interface et validation des données

Gérer l'authentification et l'inscription via la base de données

Fichiers concernés :

- src/main/java/appli/accueil/LoginController.java
- src/main/java/appli/accueil/InscriptionController.java

Modifier LoginController.java pour intégrer la base de données

Étape 1 : Ajouter l'importation du Repository

Ajoutez les imports nécessaires pour utiliser UtilisateurRepository et Utilisateur :

```
import repository.UtilisateurRepository;
import model.Utilisateur;
```

Étape 2 : Modifier les attributs du contrôleur

Ajoutez une instance de UtilisateurRepository pour interagir avec la base :

```
public class LoginController {
    private UtilisateurRepository utilisateurRepository = new UtilisateurRepository();
    [ … ]
}
```

Pourquoi instancier UtilisateurRepository ici ?

- Cela permet d'accéder aux méthodes de la base depuis le contrôleur sans avoir à l'instancier dans nos méthodes

Étape 3 : Modifier la méthode (onAction) de notre bouton « connexion »

Objectif :

- Récupérer, grâce à notre méthode « getUtilisateurParEmail », un potentiel utilisateur
  - Null dans le cas où l'utilisateur est vide
- Vérifier si l'utilisateur a bien été trouvé et si les mots de passe existent
  - Si oui, rediriger la connexion vers une page « accuteilView »

Modifier InscriptionController.java pour enregistrer un utilisateur

Étape 1 : Ajouter les imports nécessaires

Idem que pour le loginController

Étape 2 : Modifier les attributs du contrôleur

Idem que pour le loginController

Étape 3 : Modifier la méthode (onAction) de notre bouton « inscription »

Objectif :

- Récupérer l'ensemble des informations de la page
- Vérifier qu'aucune information est vide
- Vérifier si les deux mots de passe coïncident
- Vérifier si un utilisateur n'existe pas déjà avec le courriel renseigné
- Créer une instance d'utilisateur avec les informations
- Appeler la méthode d'enregistrement de notre Repository avec l'utilisateur précédemment créé

Sécurité : Pour une meilleure protection, il faudra hacher les mots de passe avant de les enregistrer.

Sécurisation des mots de passe

Nous allons maintenant sécuriser les mots de passe en utilisant **BCryptPasswordEncoder** de Spring Security, une méthode de hachage robuste et largement utilisée.

Objectif : Hacher les mots de passe avant stockage et vérifier le hash lors de la connexion

Fichiers concernés :

- pom.xml (Ajout de la dépendance Spring Security)
- src/main/java/appli/accueil/InscriptionController.java
- src/main/java/appli/accueil/LoginController.java

Etape 1 : Installation de BCrypt via Maven

Trouver, sur le site de Maven, la dépendance suivante :

spring-security-crypto

La dépendance est liée à : org.springframework.security

Synchronisez Maven après l'ajout.

Etape 2 : Hachage du mot de passe lors de l'inscription

Modifier la méthode (onAction) de notre bouton « inscription »

- Instancier un objet de type BCryptPasswordEncoder dans la méthode
- Utiliser la méthode « ecode() » pour hasher le mot de passe avant d'instancier l'utilisateur

Pourquoi utiliser BCryptPasswordEncoder ?

- Ajoute un sel unique automatiquement.
- Rend le hachage plus robuste contre les attaques par brute-force.

Etape 3 : Vérification du mot de passe lors de la connexion

Modifier la méthode (onAction) de notre bouton « connexion »

- Instancier un objet de type BCryptPasswordEncoder dans la méthode
- Utiliser la méthode « matches() » pour vérifier que le mot de passe saisi correspond au mot de passe hasher.

Pourquoi passwordEncoder.matches() ?

- Il compare **le mot de passe entré avec le hash stocké en base.**
- Il protège contre les attaques de type **rainbow table.**

Résumé des tâches à effectuer

- Ajouter la dépendance Spring Security à pom.xml.
- Modifier handleRegister() pour hacher les mots de passe avant insertion.
- Modifier handleLogin() pour vérifier les mots de passe de manière sécurisée.
- Tester l'inscription et la connexion avec des mots de passe sécurisés.

Annexes

Introduction à JDBC et Interaction avec une Base de Données

JDBC (Java Database Connectivity) est une API permettant à un programme Java d'interagir avec une base de données relationnelle (MySQL, PostgreSQL, Oracle, etc.). Elle est incluse dans le JDK et ses classes se trouvent dans java.sql et javax.sql.

Déroulé type d'une connexion JDBC

Étape 1 : Charger le driver JDBC

Chaque SGBD possède son propre driver JDBC :

- **MySQL** → MySQL Connector
- **PostgreSQL** → pgJDBC
- **Oracle** → Oracle JDBC Driver

Avec Maven, les drivers sont automatiquement gérés via les dépendances.

Étape 2 : Se connecter à la base de données

On utilise DriverManager.getConnection() pour établir une connexion avec le SGBD.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnexionJDBC {

    private String serveur = "localhost:3306"; // 3307 pour mariaDB
    private String base = "nomDeLaBase";
    private String utilisateur = "root";
    private String mdp = "motDePasse";

    public static void main(String[] args) throws SQLException {
        Connection maConnection = DriverManager.getConnection(
                "jdbc:mysql://"+ serveur +"/"+base, utilisateur, mdp);
        System.out.println("Connexion réussie !");
    }
}
```

Paramètre	Description
url	L'URL de connexion, format : jdbc:mysql://localhost:3306/nomDeLaBase
user	Nom d'utilisateur MySQL (ex : root)
password	Mot de passe de l'utilisateur

Toujours fermer la connexion avec close() après utilisation ! (dans l'idéal)

Exécution de requêtes SQL avec JDBC

Types de requêtes possibles

Méthode	Utilisation
executeUpdate()	INSERT, UPDATE, DELETE
executeQuery()	SELECT (renvoie un ResultSet)

Exécution d'une requête simple (Statement)

```
Statement stmt = maConnection.createStatement();
stmt.executeUpdate("DELETE FROM Utilisateur WHERE actif=0");
```

Exécution d'une requête paramétrée (PreparedStatement)

Utilisé pour éviter les injections SQL et optimiser les performances.

```
PreparedStatement stmt = maConnection.prepareStatement(
        "SELECT * FROM Utilisateur WHERE email = ?");
stmt.setString(1, "test@email.com");
ResultSet rs = stmt.executeQuery();
```

Avantages des requêtes paramétrées :

- Protègent contre les injections SQL.
- Permettent la précompilation des requêtes pour de meilleures performances.

Traitement des résultats (ResultSet)

ResultSet permet de parcourir les résultats d'une requête SELECT.

Méthode	Utilisation
next()	Passe à la ligne suivante
getString(int colonne)	Récupère une valeur sous forme de String
getInt(int colonne)	Récupère une valeur sous forme de int

Exemple de lecture des résultats

```
while (rs.next()) {
    System.out.println("Nom : " + rs.getString("nom") +
                       " - Email : " + rs.getString("email"));
}
```

Bonne gestion des connexions JDBC

Bonnes pratiques

- Toujours fermer la connexion après utilisation (close()).
- Utiliser PreparedStatement pour les requêtes avec paramètres.
- Gérer les exceptions avec try-catch pour éviter les erreurs SQL.

Exemple de connexion complète et sécurisée

```
try (Connection maConnection = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/maBase", "root", "")) {
    PreparedStatement stmt = maConnection.prepareStatement(
            "SELECT * FROM Utilisateur WHERE email = ?");
    stmt.setString(1, "test@email.com");
    ResultSet rs = stmt.executeQuery();

    while (rs.next()) {
        System.out.println("Utilisateur : " + rs.getString("nom"));
    }
} catch (SQLException e) {
    System.out.println("Erreur SQL : " + e.getMessage());
}
```

Conclusion

- **Connexion** : Utiliser DriverManager.getConnection()
- **Préparation des requêtes** : Statement ou PreparedStatement
- **Exécution des requêtes** : ExecuteQuery (LID) ou ExecuteUpdate (LMD)
- **Lecture des résultats** : ResultSet.next() et getString(), getInt()...
- **Fermeture des ressources** : Toujours fermer la connexion avec close()