Application de **Gestion de Produits**

Nous souhaitons faire une solution d'un logiciel pour gérer les achats et ventes des produits [par catégorie] d'un magasin en ligne en se basant sur une architecture JEE.

Les acteurs :

L'application sera utilisé par deux profils

- Administrateur [Propriétaire]
- Client

Fonctionnalités

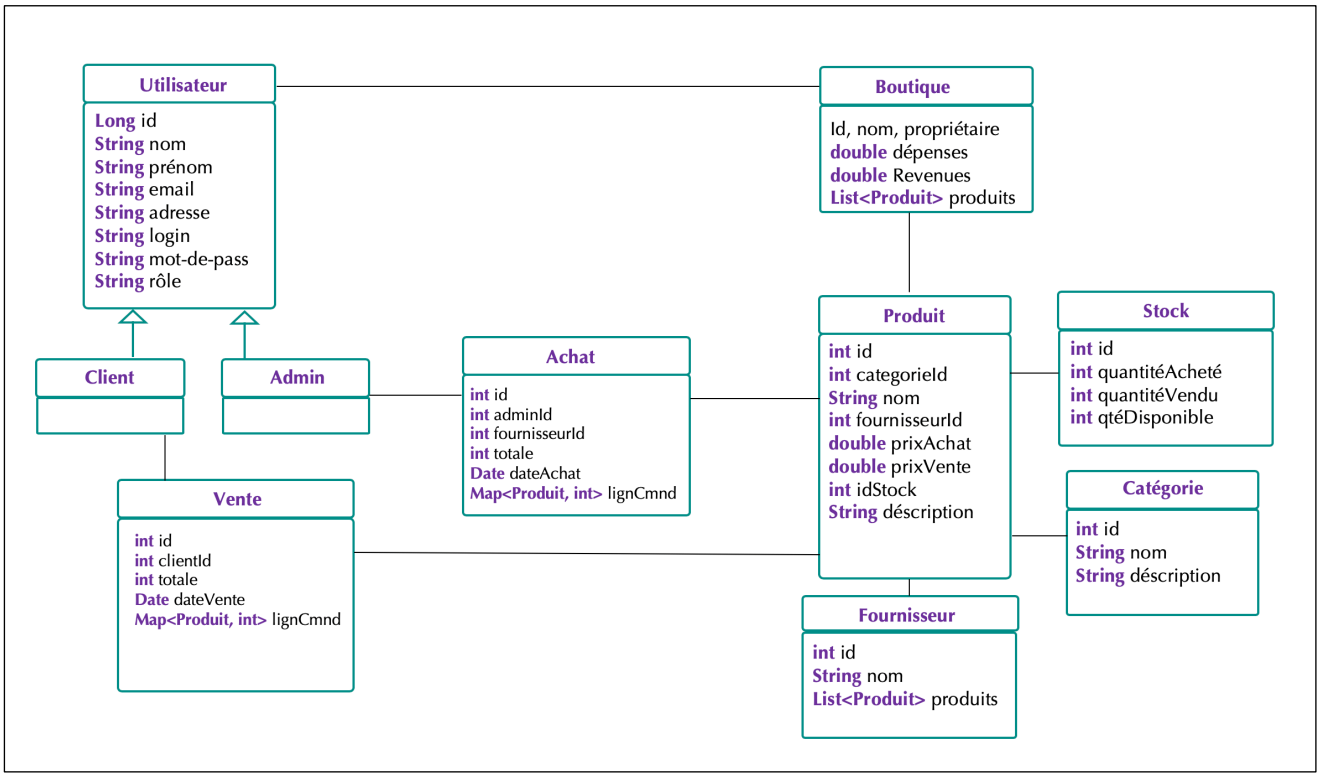
Rôle de l'administrateur

- Gestion des Produits
- Gestion des fournisseurs
- Gestion des Clients
- Gestion des achats et Ventes
- Gestion du stock
- Effectuer des commandes d'achat

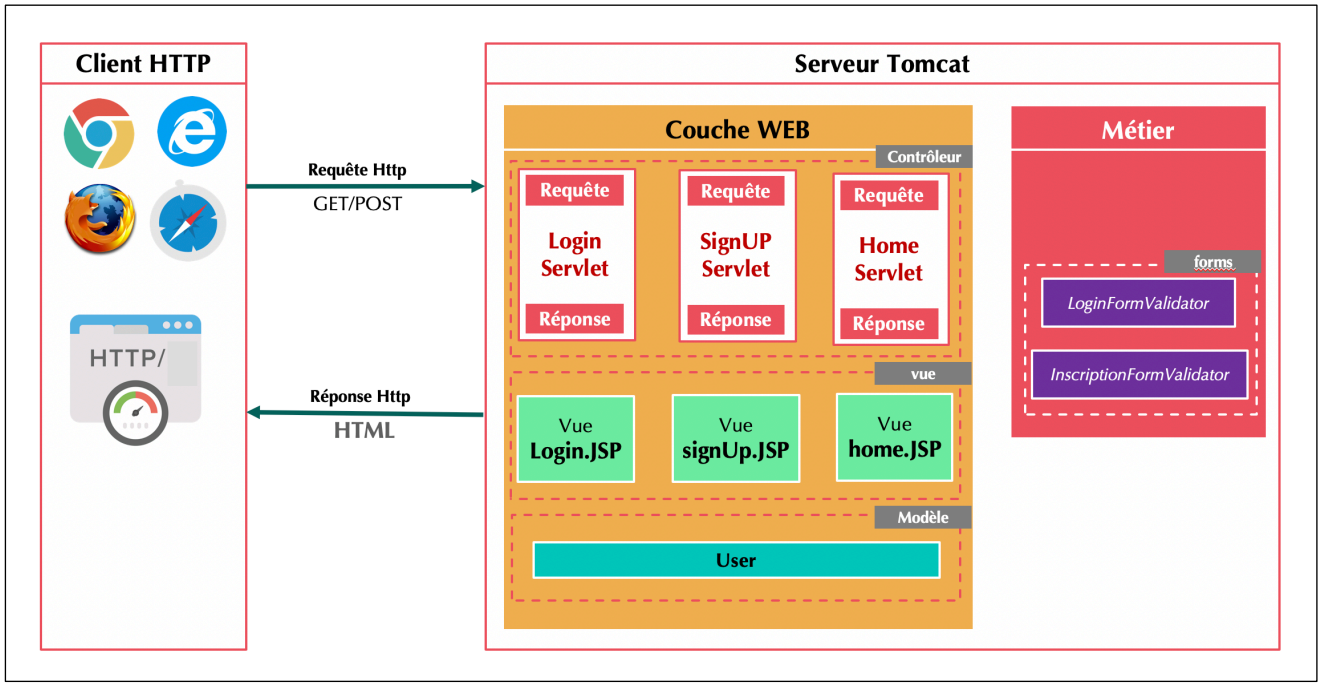
Rôle du Client

- Consultation des Produits
- Effectuer des commandes de vente
- Consulter ses commandes

Diagramme de Classe :



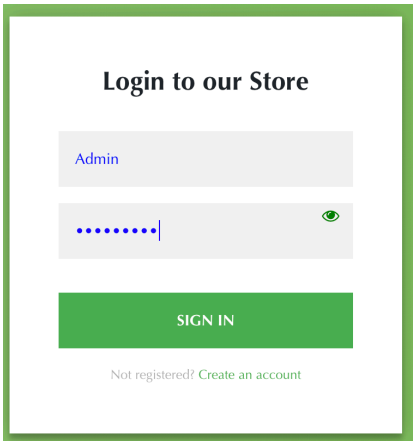
Atelier 1 : Authentification



Procédure d'authentification. :

1. Le déploiement de l'application charge la servlet « **Login-Servlet** » dans le conteneur en passant par une étape d'initialisation de la servlet en appelant la méthode **init()** de cette servlet pour initialiser les variables nécessaires puis un appel à la méthode **service()** qui lie le couple requête/réponse http aux objets HttpServletRequest et HttpServletResponse créé par le conteneur, fait appel la méthode **doGet()** pour se rediriger vers la page « **login.jsp** ».

2. Un utilisateur tape son **login** et son **mot de passe** dans le formulaire proposé par la page **«login.jsp»**



3. Le requête généré est une **requête http** de type **POST**, qui va être intercepté par la Servlet « **Login-Servlet** »

4. La méthode **service()** fait la correspondance une autre fois en appelant la méthode **doPost()** qui va lire les paramètres de la requête (à savoir le **login** et le **mot-de-passe**)
5. Un appel à l'objet **LoginFormValidator** de couche **métier**, est nécessaire pour **vérifier** et **valider** les champs entrés du formulaire pour **établir la connexion** et créer une **session** pour l'utilisateur courant.
6. L'objet **LoginFormValidator** implémente l'interface **ILoginForm** suivante :

```
public interface ILoginForm

String CHAMP_LOGIN  = "login";
String CHAMP_PASS   = "pass";

String getFieldValue( HttpServletRequest request,
                      String fieldName           );

void putErrors(String field, String ErrMsg);

void putValidationMessage(String Msg);

void ValidateLogin(String lg) throws Exception;

void ValidatePassword(String pass) throws Exception;

User ValidateUser(HttpServletRequest request);
```

Fait appel à la fonction **ValidateUser()** qui appel à son tour la fonction **getFieldValue()** pour récupérer les valeurs des deux champs envoyés avec la requête en tant que ses paramètres, puis les valide selon les règles de gestion proposées (via les fonctions **ValidateLogin()** et **ValidatePassword()**), puis fait appel à la base de données pour chercher un enregistrement d'utilisateur ayant les données validées correspondantes.

7. Dans un premier temps on va utiliser une base données orienté objet où on va stocker des objets figés via le code dans une classe BD dans un package ressources juste pour le test de notre application. (après nos données vont être structurées dans une BD relationnelle, dont on va communiquer à notre application via une couche ORM qui va suivre le modèle DAO)

```
public class BD {

 public static List<Utilisateur> TableUser;
 public static void Connection() {

     TableUser = new ArrayList<Utilisateur>();
     TableUser.add(
                 new Utilisateur(
                                 "Admin",
                                 "admin@jee",
                                 "Omar El Midaoui"
                                ));}
                 }
```

Notre classe BD utilise le modèle **Utilisateur**, donc on a besoin de créer notre classe **modèle** pour représenter nos données :

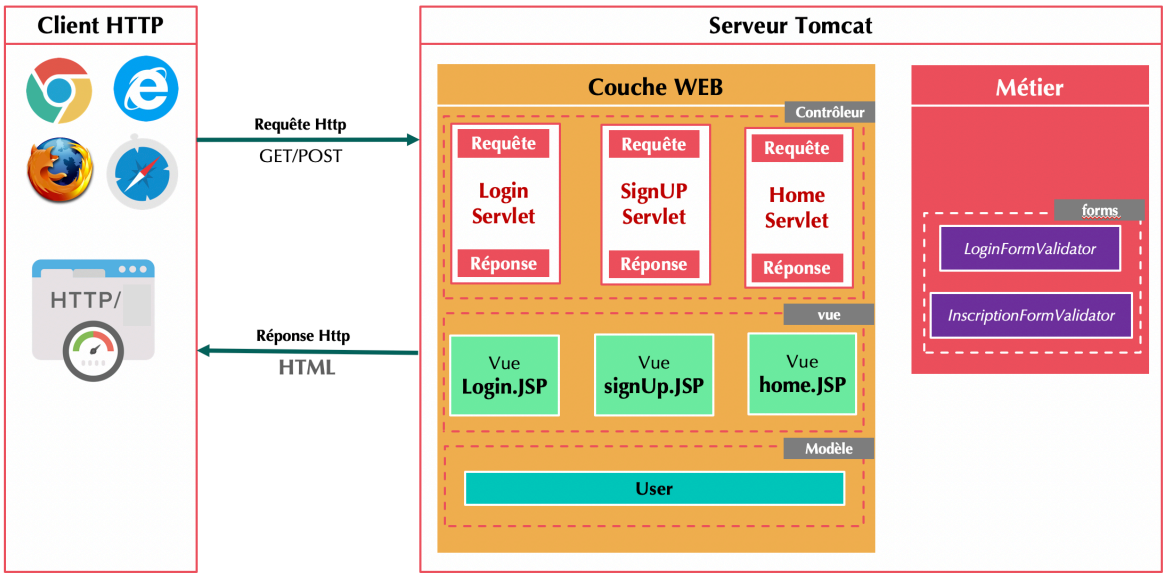
```
public class Utilisateur {

  private String login, motDePasse;
  private String nomComplet;


  public String getLogin()      { return login;      }
  public String getMotDePasse() { return motDePasse; }
  public String getNomComplet() { return nomComplet; }


  public void setLogin(String lg)        { login      = lg;  }
  public void setMotDePasse(String pass) { motDePasse = pass;}
  public void setNomComplet(String nom)  { nomComplet = nom; }

  public Utilisateur() {}

  public Utilisateur(String lg, String pass, String nom) {

                                this.login = lg;
                                this.motDePasse = pass;
                                this.nomComplet = nom;
                                                      }
  public Utilisateur(Utilisateur U) {

                        this.login      = U.login;
                        this.motDePasse = U.motDePasse;
                        this.nomComplet = U.nomComplet;

                                }
                        }
```

8. Fonctionnement de la fonction **ValidateUser()** :
   Si **les champs entées sont valides** et suivent les contraintes fonctionnelles alors on génère pas d'erreurs, et on passe directement à la recherche de l'utilisateur dans la Table Users de la classe BD.
   Dans ce cas deux scénarios sont possible :
   - Si un **utilisateur** avec les données validées existe dans la BD, alors cet utilisateur est retourné et un message de validation contenant la valeur « Connexion réussie » est créé.
   - Si un **utilisateur** avec les données validées n'existe pas dans la BD, alors la fonction retourne **null** et un message de validation contenant la valeur « Connexion échoué» est créé.

   Sinon si les champs entées sont pas valides (ne suivent pas les contraintes fonctionnelles de l'application) alors les erreurs générés via les exceptions levés vont être stocker dans une List ou Map d'erreurs , et la fonction retourne **null** .

9. Une fois la validation est faite, selon le retour La servlet va rediriger la réponse vers la bonne vue.
   - Si **le retour est différent de null** alors on a pu trouver un utilisateur, du coups une session contenant les données de cet utilisateur est créé, le message de validation est stocké dans la requête et la réponse va être rediriger vers « **Home-Servlet** » pour afficher le contenue de la page **« home.jsp »**
   - Sinon si **le retour est null** alors on stock les données générés par l'objet form (erreurs générés et le message de validations) dans la requêtes et on redirige la réponse va la servlet « **Login-Servlet** » même, pour afficher les erreurs sur le formulaire.

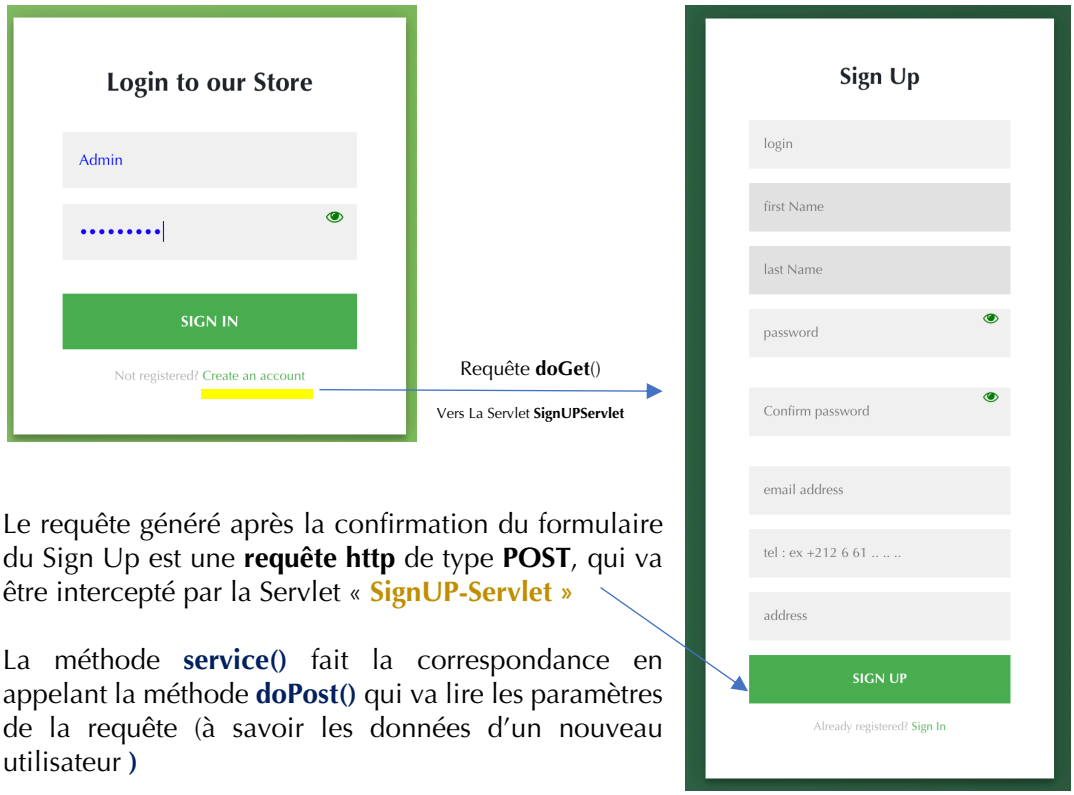
Atelier 2 : Inscription



Procédure d'inscription d'un nouveau utilisateur :

1. Le déploiement de l'application charge aussi la servlet « **SignUP-Servlet** » dans le conteneur. Cette servlet sert à contrôler la vue définit par la page JSP « **iscription.jsp** ».

2. La redirection vers la page **«iscription.jsp»** peut se faire depuis le formulaire proposé par la page **«login.jsp»**



3. Le requête généré après la confirmation du formulaire du Sign Up est une **requête http** de type **POST**, qui va être intercepté par la Servlet « **SignUP-Servlet** »

4. La méthode **service()** fait la correspondance en appelant la méthode **doPost()** qui va lire les paramètres de la requête (à savoir les données d'un nouveau utilisateur)

5. Un appel à l'objet **IscriptionFormValidator** de couche **métier**, est nécessaire pour **vérifier** et **valider** les champs entrés du formulaire pour **établir la création d'un nouveau utilisateur** et créer une **session** pour lui et le connecter.

6. L'objet **IscriptionFormValidator** implémente l'interface **IRegisterForm** suivante :

```
public interface IRegisterForm {

    String CHAMP_LOGIN      = "lg",      CHAMP_EMAIL    = "email";
    String CHAMP_FNAME      = "fname",   CHAMP_LNAME    = "lname";
    String CHAMP_PASS       = "pass",    CHAMP_CPASS    = "cpass";
    String CHAMP_ADDRESS    = "address", CHAMP_TEL      = "tel";

    String  getFieldValue(HttpServletRequest Rq, String field);

    void    putErrors(String field, String error);

    void    putValidationMsg(String Msg);

    void    ValidateLogin(String login) throws Exception;

    boolean ValidatePass(String pass , String cpass) throws Exception;

    void    ValidateEmail(String mail) throws Exception;

    void    ValidateName(String fName, String lName) throws Exception;

    void    ValidateAddress(String address) throws Exception;

    void    ValidateTel(String tel) throws Exception;

    User    RegisterUser(HttpServletRequest Rq);

                        }
```

Fait appel à la fonction **RegisterUser()** qui appel à son tour la fonction **getFieldValue()** pour récupérer les valeurs des champs envoyés avec la requête en tant que ses paramètres, puis les valide selon les règles de gestion proposées (via les fonctions **ValidateLogin()**, **ValidatePass(), …**), La validation du **login** et de **l'email** fait appel à la base de données pour chercher un enregistrement d'utilisateur ayant les même valeurs pour empêcher la violation des contraintes d'unicités de ces deux champs.

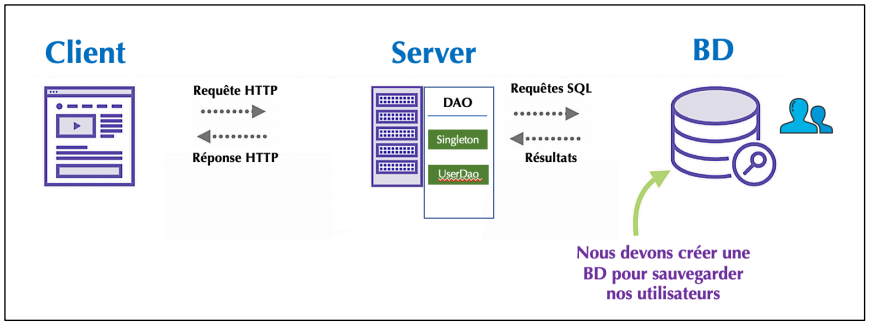
Pour les champs **email** et **téléphone**, on peut les vérifier aussi en utilisant la fonction :
.matches(Expresion)

```
if(!email.matches("([^.@]+)(\\.[^.@]+)*@([^.@]+\\.)+([^.@]+)"))
```

```
if ( !tel.matches( "^\\d+$" ) )
```

Se Connecter à notre modèle : (Pattern **Singleton**)

7. Utilisation de la couche **DAO** comme couche intermédiaire pour l'accès à une base de données **MySQL**



Notre Couche DAO aura pour le moment deux classes :
Une classe « **DbConnection** » pour établir la connexion à notre base de donnée :

Cette classe doit être codé suivant « **le pattern singleton** » pour garantir qu'une unique instance de cette classe sera créée, et bien sûr offrir un point d'accès universel à cette instance.

Il faut donc interdire à tout code extérieur dans les autres couches de l'application d'utiliser l'opérateur "**new**" et de créer des instances supplémentaires. Pour cela, il suffit de déclarer un constructeur de visibilité "**privé**".

Pour cela il faut passer par une méthode utilitaire (**getInstance**()) au lieu du constructeur. Cette méthode sera nécessairement statique, car à cet instant, le code appelant ne dispose encore d'aucune référence sur l'instance du **singleton**, et ne peut donc accéder qu'à ses membres statiques.

La méthode utilitaire étant statique, elle ne peut accéder qu'aux propriétés également statiques de la classe. L'instance unique devra donc être statique aussi.

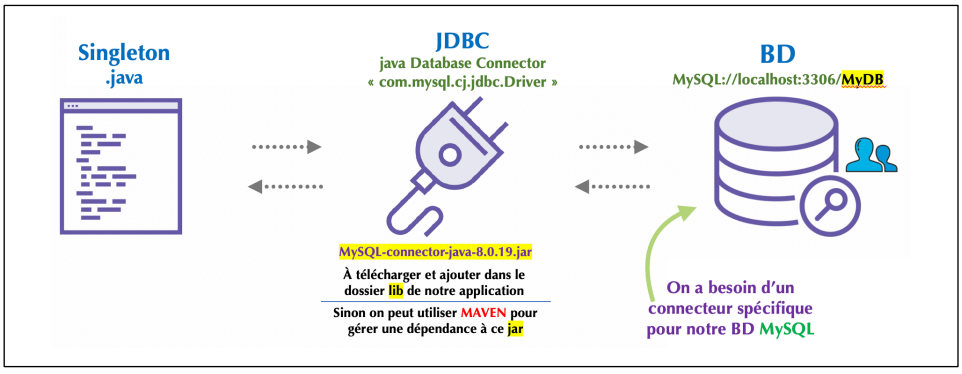
```
public class Singleton {
    // Constructeur privé
    private Singleton()  {}

    // Instance unique pré-initialisée
    private static Singleton INSTANCE = new Singleton();


    // Point d'accès pour l'instance unique du singleton
    public static Singleton getInstance()
    {   return INSTANCE;    }
}
```

Ça donne dans notre cas

```
import java.sql.Connection;

public class DbConnection {
        // Constructeur privé
        private DbConnection()  {}

        // Instance unique pré-initialisée
        private static DbConnection INSTANCE = new DbConnection();

        // La connection qui caractérise notre singleton
        private static Connection connection;

        // Point d'accès pour l'instance unique du singleton
        public static DbConnection getInstance()
        {   return INSTANCE;    }

        // Point d'accès pour l'instance unique de la connexion
        public static Connection getConnection()
        {   return connection;    }

                }
```

Pour se connecter à une base de donnée **MySQL** depuis une application Java, la procédure est simple :

- On a besoin d'abords d'un connecteur (d'un pilote ou driver), et dans notre cas, celui qui est spécifique à une BD **MySQL** → MySQL-connector-java-8.0.19.jar



Retournant Maintenant à notre classe **singleton** : **DbConnection :**

- Pour se connecter et se déconnecter on va ajouter deux fonctions statiques :

```
private static void Connect() {  /* ... */   }

private static void Disconnect() { /* ... */ }
```

- On va s'assurer ne pouvoir créer notre connexion dans la fonction **Connect()**, que si la connexion est **null** ou bien **fermée**
- Pour créer une instance de la connexion, on fait appel au driver « l'objet DriverManager » qu'on a ajouté à notre **buildPath**. Il a besoin de l'adresse vers la DB cible dans le serveur **MySQL** et bien sûr du login et mot de passe aussi pour y accéder.
- Nos fonctions **Connect()** et **Disconnect()** deviennent :

```
private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
private static final String URL    = "jdbc:mysql://localhost:3306/MyDB";
private static final String LOGIN  = "root";
private static final String PASS   = "root";

private static void Connect() {

    try {

            Class.forName(DRIVER); // chargement du driver jdbc

            if (connection == null || connection.isClosed())
                connection = DriverManager.getConnection(URL, LOGIN, PASS);

        }
    catch (Exception e) { e.printStackTrace(); System.exit(0); }
}


private static void Disconnect() {

    try {
            if (connection != null && !connection.isClosed())
                connection.close();

        }
    catch (Exception e) {   e.printStackTrace();    }
}
```

Envoyer des Requêtes SQL à notre Base de Données :

Pour organiser notre code d'une façon similaire à notre schéma de base de donnée.
Nous devons créer une classe **UserDao** à travers la quelle on va pouvoir exécuter l'ensemble de nos requête SQL qui ciblent la table « **User** » de notre BD.

- Pour pouvoir formuler nos requête SQL dans notre classe Dao, on utilise notre instance Connexion pour créer un objet **Statement**. (qu'on va changer par la suite vers un objet **PreparedStatement** qui offre plus de sécurité contre les injections **SQL**)
- Une fois on crée notre **Statement** on l'utilise pour exécuter nos requêtes, on utilise la méthode **executeQuery()** pour les sélections, recherches et suppressions de données et la fonction **executeUpdate()** pour les modifications et insertions.
- Le résultat de l'exécution de notre requête **SQL** est un objet de type **ResultSet** qui peut être vu comme un tableau spéciale (comme une liste de type Set) de résultats, dont chaque élément peut être vu comme une Map clé/valeur où chaque clé correspond à une colonne ou un champ, et chaque valeur correspond à la valeur correspondante dans ligne d'enregistrement.
- Pour parcourir un objet **ResultSet** on utilise un curseur à travers les fonctions suivante : **next()** et **previous()** qui renvoient **vrai** ou **faux** après déplacement du curseur sur la ligne suivante ou précédente. **vrai** si le curseur est positionné sur une ligne, et **faux** si on a dépassé la fin du tableau.

La classe **UserDao** :

```
public class UserDao {

    private Connection connection;

    public UserDao(Connection cnx) { this.connection = cnx; }

    public User Find(String login, String password) {

        User                user     = null;
        PreparedStatement   stmt     = null;
        ResultSet           resultat = null;

        String sql = "Select * From user where login = ? AND password = ? ";
        try {
                        stmt = connection.prepareStatement(sql);
                        stmt.setString(1, login);
                        stmt.setString(2, password);

                        resultat = stmt.executeQuery();

                        if(resultat.next()) {

                            String username = resultat.getString("login");
                            String pass     = resultat.getString("password");
                            String email    = resultat.getString("email");
                            String fname    = resultat.getString("firstName");
                            String lname    = resultat.getString("lastName");

                            user = new User(username, pass, email, fname, lname);

                            }

             } catch (SQLException e) { e.printStackTrace();}
        return user;
    }
}
```

On a besoin de l'instance de la **connexion** pour pouvoir exécuter nos requêtes

Après on implémente les différents requêtes **SQL** sous forme de fonctions **DAO**.
Ici on a commencé par la fonction **FIND** qui traduit en java une requête **SQL** de type **SELECT**.

ORM :
Le résultat de la requête **SQL** (si existe) est mappé à un objet java de notre modèle : ici un objet **User** qui va être retourné par la fonction.

On peut ajouter un bloc finally pour fermer nos objets **Statement**, **ResultSet** et aussi la **Connexion**.

```
finally {

    try {
            resultat.close();
            stmt.close();
        }

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

    DbConnection.Disconnect();

}
```

Modification du code de contrôle appelant les fonctions **ValidateUser()** et **RegisterUser()** :
Le fonctionnement reste le même, sauf notre objet métier a besoin maintenant d'un objet **UserDao** pour pouvoir questionner la base de données.

```
private IFormLogin form;
private UserDao dao;

@Override
public void init() throws ServletException {

    dao = new UserDao(DbConnection.getInstance().getConnection()); }


protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                              throws ServletException, IOException {

    form            = new FormLoginValidator(dao);
    User loggedUser = form.ValidateUser(request);

           if(loggedUser ==null)   {

              request.setAttribute(ATT_FORM, form);
              request.getRequestDispatcher(VUE_LOGIN).forward(request, response);

                                    }

           else                     {

              HttpSession session = request.getSession();
              session.setAttribute(ATT_USER, loggedUser);
              session.setAttribute(ATT_FORM, form);

              request.getRequestDispatcher(VUE_HOME).forward(request, response);

                                    }
                                                          }
```

On a besoin de d'un objet **dao** pour interroger la BD

la récupération de l'instance de la connexion et l'initialisation de notre **dao** se fait dans la fonction **init()** de notre contrôleur

L'instance de notre **dao** est utilisé par notre **objet métier** pour la vérification et la validation de nos données de requête

Même procédure pour l'enregistrement