

Création d'un système de communication Client-Serveur basé sur les Sockets

PROGRAMMATION RESEAU EN JAVA

Eya ZAOUI | Projet Téléinformatique | RT 2 – 1

Table des matières

Introduction	2
Etude théorique	3
Description du travail	5
I. Conception	5
II. Réalisation	6
1. Outils et langages utilisés.....	6
2. Présentation du système	6
Conclusion.....	12
Bibliographie	13
Annexe.....	13

Introduction

La notion de sockets a été introduite dans les distributions de Berkeley (un système UNIX) dans les années 80. On parle parfois de sockets BSD (Berkeley Software Distribution).

Dans le contexte des logiciels, les sockets sont des « connecteurs réseau » ou des « interfaces de connexion ». C'est un mécanisme universel de bas niveau : il permet l'utilisation directe du transport réseau (construit sur TCP ou UDP), depuis tout langage de programmation. Sur cette couche logicielle, le développeur requiert des services en appelant des fonctions, et elle masque le travail nécessaire de gestion du réseau, pris en charge par le système.

Nous allons utiliser les Sockets Java pour établir une connexion entre un programme client et un programme serveur. Le serveur gère une base de données SQL. Nous voulons utiliser l'application cliente pour envoyer un ID d'un étudiant au serveur et récupérer ses données de la BD.

Etude théorique

La communication Client-Serveur est un modèle de communication où une application cliente envoie une requête à un serveur, qui traite cette requête selon un service précis, puis envoie une réponse au client.

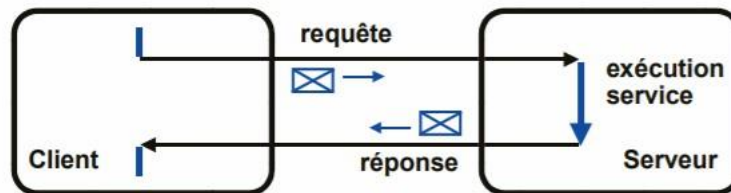
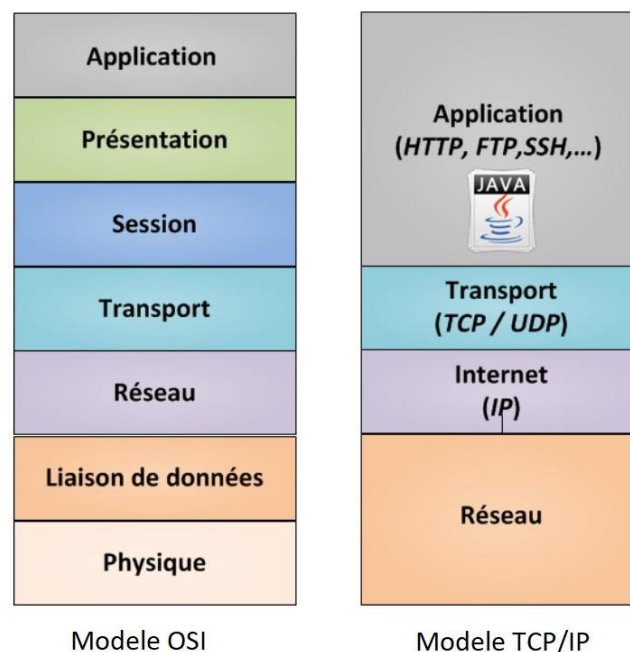


Figure 1- Modèle Client-Serveur

Cette communication peut se faire via les Sockets.

Les sockets forment un mécanisme permettant la communication inter-processus (IPC) sur une même machine ou la communication à travers un réseau TCP/IP.

Les sockets définissent l'interface entre la couche application et la couche transport du modèle TCP/IP.



La couche de Transport requiert la définition au moins d'un port et d'un mode de communication.

Les sockets étant Full-Duplex (flux entrée-sortie), il suffit de déterminer un port sur lequel le client et le serveur vont communiquer. Celui-ci ne doit pas se confondre avec d'autres ports, comme les ports connus ou enregistrés. Pour ce projet, le port sera d'un numéro entre 49152 et 65535.

Il existe 2 modes de communications :

- Mode connecté : Protocole TCP
Ce mode garantit l'ordre, la fiabilité et le contrôle de flux de données. Une connexion durable est établie entre les deux processus. La communication se fait par : Ouverture d'une liaison, suite d'échanges séquentielle, fermeture de la liaison. Si un paquet est perdu, sa transmission est relancée. Le serveur préserve son état entre deux requêtes (toujours connecté).
- Mode déconnecté : Protocole UDP
Ce mode n'a pas de garantie particulière. Les requêtes successives sont indépendantes. Il n'y a aucune liaison établie et il n'y a pas de préservation de l'état entre les requêtes (Le serveur peut être déconnecté). Donc, pour recevoir des réponses, le client doit indiquer son adresse à chaque requête. En fait, il émet des datagrammes : paquets de données chacune comprenant son adresse et sa requête. Puisque le serveur ne connaît pas si un client essaie de lui communiquer, si un paquet est perdu, il ne peut pas être récupéré.
- Points communs : Le client a l'initiative de la communication ; le serveur doit être à l'écoute. Le client doit également connaître l'adresse IP et le numéro de port du serveur. Le serveur peut servir plusieurs clients (1 thread unique ou 1 thread par client).

Dans ce projet, nous traitons la programmation des Sockets en Java. Nous adoptons la communication Client-Serveur en mode connecté. Par ailleurs, nous ne considérons qu'un seul client. Il n'y aura pas de multi-threading pour plusieurs clients.

Le mode de gestion de données choisie est itératif : le serveur traite les requêtes les unes après les autres. Le client ouvre une connexion avec le serveur avant de pouvoir envoyer des requêtes, puis ferme la connexion à la fin de la suite d'opérations. Ceci assure le traitement des pannes et le maintien de la logique d'ordre des opérations.

Description du travail

I. Conception

Notre système se compose par un programme serveur et une base de données SQL qui définit le service fournit par notre serveur. Au cotée client, nous créons deux classes : une classe Client qui établit effectivement la connexion avec le serveur et une classe ClientUI qui instancie un Client et crée une interface graphique (User interface).

Nous utilisons les outils fournis par les bibliothèques Java pour construire le système.

❖ Sockets en Java :

- La classe ServerSocket : socket côté serveur. Ce socket attend les connexions et crée un socket service client pour répondre aux requêtes du client.
- La classe Socket : sockets ordinaires, pour les échanges. Elles fournissent des classes InputStream et OutputStream pour échanger de l'information en entrée-sortie.



Figure 2- Socket client et Socket serveur

❖ Structure de données :

- La classe Etudiant : cette classe Java est l'image de la table Etudiant de la base de données. Le serveur utilise les objets de cette classe pour manipuler les résultats des requêtes SQL.

Etudiant	
-	id : int
-	nom : String
-	dateDeNaissance : String
-	nationalite : String
-	email : String
-	adresse : String

- La classe Server : C'est le programme serveur qui ouvre un server socket et écoute sur son port en attendant une connexion d'un client sur le même port.
- La classe Client : C'est le programme client qui établit une communication avec le serveur. Néanmoins, cette classe ne contient pas une méthode main(), elle est plutôt instanciée par ClientUI.
- La classe ClientUI : La fonction de cette classe est de créer une interface graphique pour l'utilisateur (User Interface). Elle instancie un objet Client qui lance la communication avec le serveur.

- La base de données « etudiants.sql » : elle contient une seule table Etudiant, avec 3 entrées.

Id	Nom	DateDeNaissance	Nationalite	Email	Adresse
1800234	Madeline Dubois	2000-06-06	Française	M.Dubois@hotmail.fr	9 rue Palestine Tunis
1800467	Eya Zaoui	1999-07-20	Tunisienne	zaouieya@gmail.com	14 rue 10469 Ariana
1800568	Ahmed Samir	1998-03-14	Tunisienne	ahmeds@yahoo.fr	2 rue Les Jasmins Manzah 1

Figure 3- La table Etudiant, (depuis phpMyAdmin)

II. Réalisation

1. Outils et langages utilisés

Le langage de développement est **Java**, JDK version 13. L'environnement utilisé est l'IDE **Eclipse**.

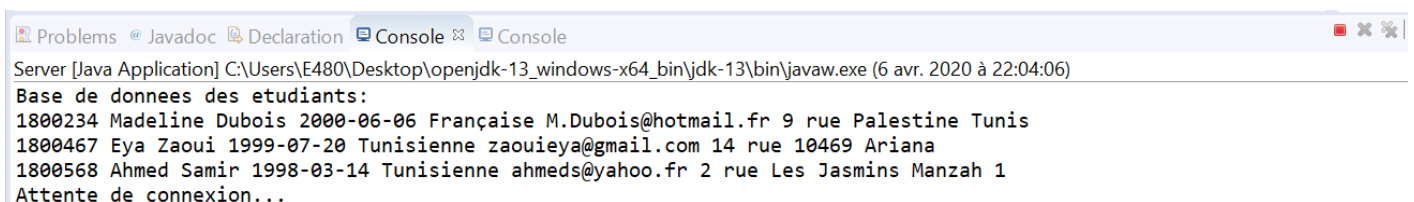
La base de données est écrite en SQL et développée sur **MySQL Server** version 5.5, gérée par phpMyAdmin 5.0.1.

Ce projet a besoin d'un MySQL Server activé et est mieux compilé comme un projet Java dans un IDE.

2. Présentation du système

Le déroulement du système se fait comme suit :

- On met MySQL Server en marche. On y importe la base de données « etudiant.sql ». On importe ensuite le projet « Etudiants_EyaZaoui » dans un IDE .
- On exécute Server.java pour mettre notre serveur en marche. Si tout va bien, on reçoit en console de l'IDE ce résultat :



```

Server [Java Application] C:\Users\E480\Desktop\openjdk-13_windows-x64_bin\jdk-13\bin\javaw.exe (6 avr. 2020 à 22:04:06)
Base de donnees des etudiants:
1800234 Madeline Dubois 2000-06-06 Française M.Dubois@hotmail.fr 9 rue Palestine Tunis
1800467 Eya Zaoui 1999-07-20 Tunisienne zaouieya@gmail.com 14 rue 10469 Ariana
1800568 Ahmed Samir 1998-03-14 Tunisienne ahmeds@yahoo.fr 2 rue Les Jasmins Manzah 1
Attente de connexion...

```

Figure 4- Server.java en marche

Si des problèmes de connexion surviennent, on aura des messages d'aide dans la console :



```
Problems Javadoc Declaration Console Console
<terminated> Server [Java Application] C:\Users\E480\Desktop\openjdk-13_windows-x64_bin\jdk-13\bin\javaw.exe (7 avr. 2020 à 00:59:02)
com.mysql.cj.jdbc.exceptions.CommunicationsException: Communications link failure

The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.
Verifier que vous avez mysql-connector-javafx dans le Build Path (voir READ_ME.txt)
Verifier que votre MySql Server est ACTIF.
Verifiez que vous avez importer la BD sur votre MySql server avec les URL, username et mot de passe appropriés.
```

Figure 5- Server: Problèmes de connexion

En fait, notre programme serveur tente en premier lieu de créer un Server Socket sur un port déclaré au début.

Si aucun problème ne survient -par exemple, port en cours d'utilisation par un autre programme-, le serveur tente d'accéder à la Base de Données(BD). Cet accès est défini dans la méthode membre `mySQLAccess()`. Le serveur déclare un objet de la classe `Connection` du paquet `java.sql`. On lui instancie une connexion par le `DriverManager`, qui prend en argument un URL décrivant quel type de `JavaDataBaseConnector` à utiliser (ici, c'est `jdbc:mysql`) et où trouver le serveur MySQL (ici, sur `localhost:3306`). Il prend aussi le nom de la BD (`etudiants`) et le nom d'utilisateur et mot de passe (`root` et chaîne vide comme mot de passe). Avec ces données, le `DriverManager` établit la connexion avec MySQL Server sur le port `3306` et accède à la BD. Notre objet `Connection` peut maintenant exécuter des requêtes SQL vers notre BD. On crée une requête pour récupérer toutes les entrées de la table `Etudiant`, pour visualiser le contenu de la BD.

Le serveur reste sur écoute du socket en attente d'une connexion d'un client.

- On exécute `ClientUI.java` pour lancer le programme client. D'abord, nous aurons cette interface graphique, développée avec les paquets `Swing` et `AWT` :

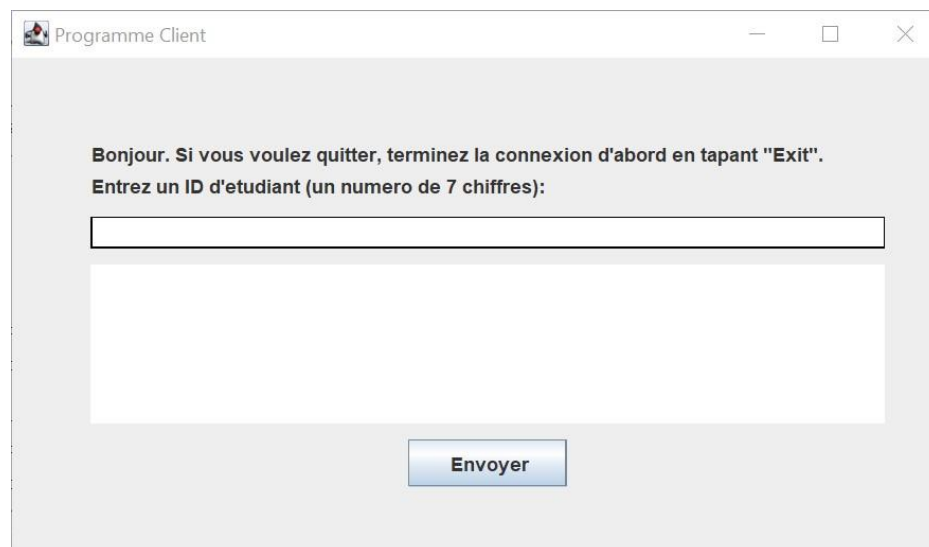


Figure 6- Démarrage de ClientUI.java

Ensuite, ce programme instancie un objet Client, qui établit une connexion avec notre serveur. Dans le constructeur du Client on trouve :

```
//get the localhost IP address
    InetAddress serverAddress = InetAddress.getLocalHost();

//establish socket connection to server
    socket = new Socket(serverAddress.getHostName(), port);
```

On trouve aussi l'instanciation des flux d'Entrée/Sortie (Input/Output), pour assurer l'échange de données dans le socket.

Notre serveur se rend compte de la connexion du client et instancie de sa part les flux I/O. On aura dans la console du serveur ce ligne s'ajouter :



Figure 7- Server.java : un client détecté

- Le serveur entre en écoute sur socket en boucle, jusqu'à la réception de « exit » :

```
//keeps listening until receiving 'exit' call or program terminates
message = brSocket.readLine();
while (!message.equalsIgnoreCase("exit"))
```

Cela assure la possibilité d'échanger plusieurs requêtes entre le client et le serveur dans une même session, puisque le serveur ne s'éteint que lorsque le client demande de terminer la connexion : serveur itératif en mode connectée. D'autre part, il faudrait vérifier que le client ne quitte son programme qu'après avoir entré « exit » pour terminer la session, pour que le serveur ne reste pas en marche indéfiniment. Pour cela, l'icône de fermeture du programme ClientUI n'est activé que si le client entre « exit ».

Le message « exit » entré engendre :

- La signalisation au serveur de quitter sa boucle.
- La fermeture des ressources (sockets et flux I/O) de chaque côté client-serveur.
- L'activation de l'icône de fermeture du programme de ClientUI.
- La désactivation du bouton « envoyer ».

La fin de connexion entre le client et le serveur est définie dans la méthode `endConnection()` du Client.

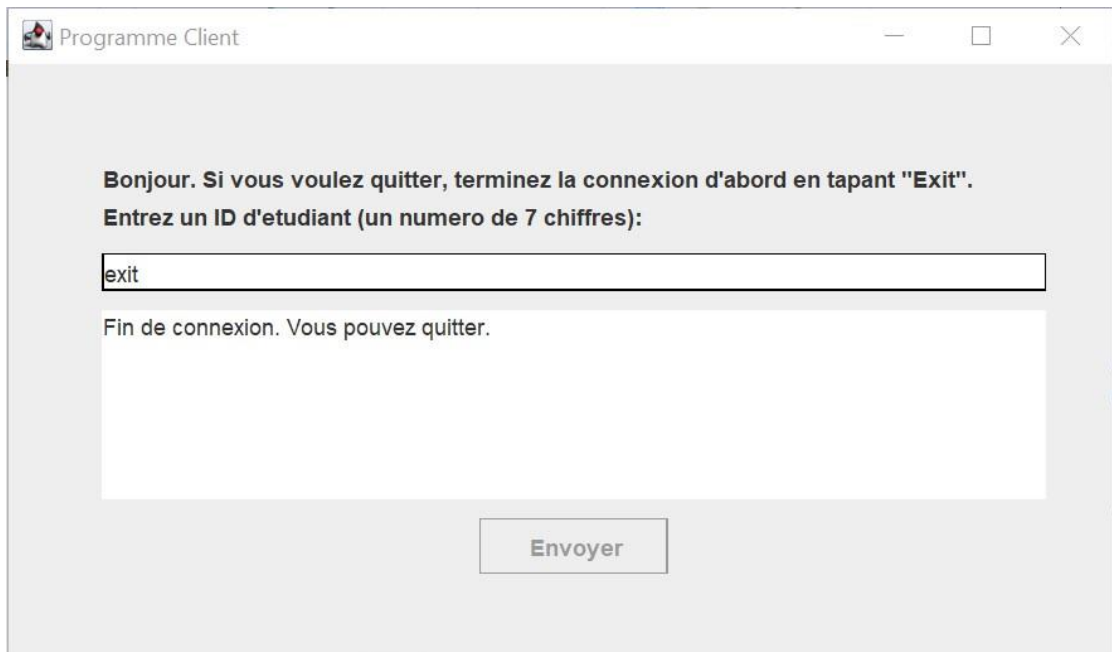


Figure 8- ClientUI: Fin de connexion

Evidemment, si l'utilisateur exécute ClientUI avant de mettre en marche le serveur, il sera signalé dans l'interface, le bouton « envoyer » sera désactivé, et l'icône de fermeture sera accessible.

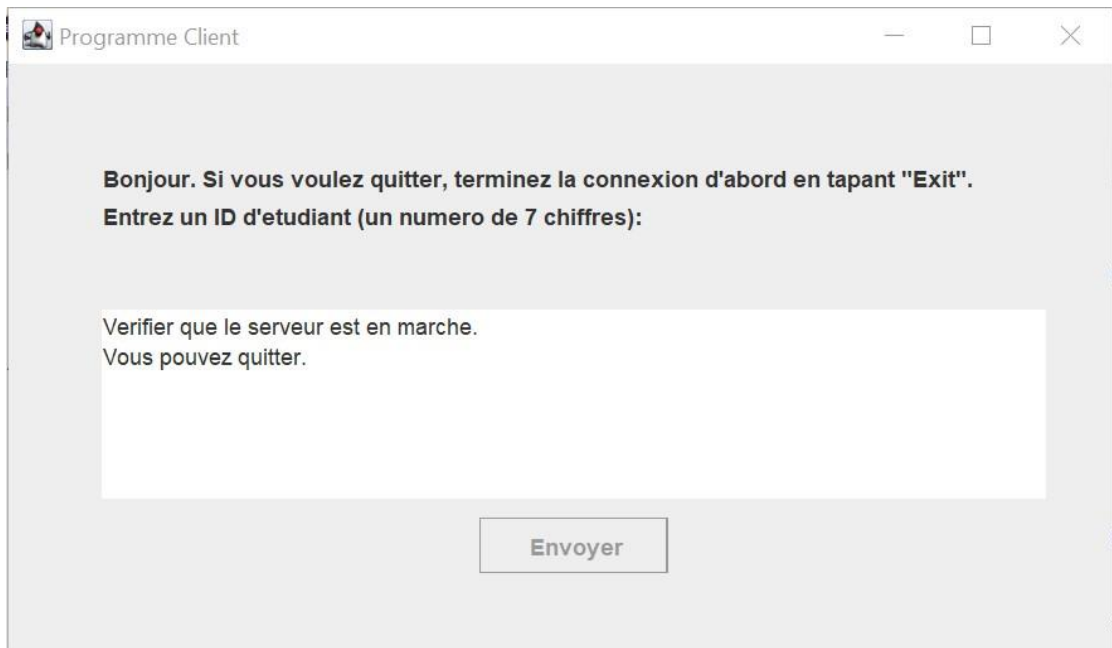


Figure 9- Client UI: connexion refusée, serveur arrêté

- La connexion entre le client et le serveur est établie. Il suffit d'entrer un ID d'étudiant valide pour voir le résultat de la BD :

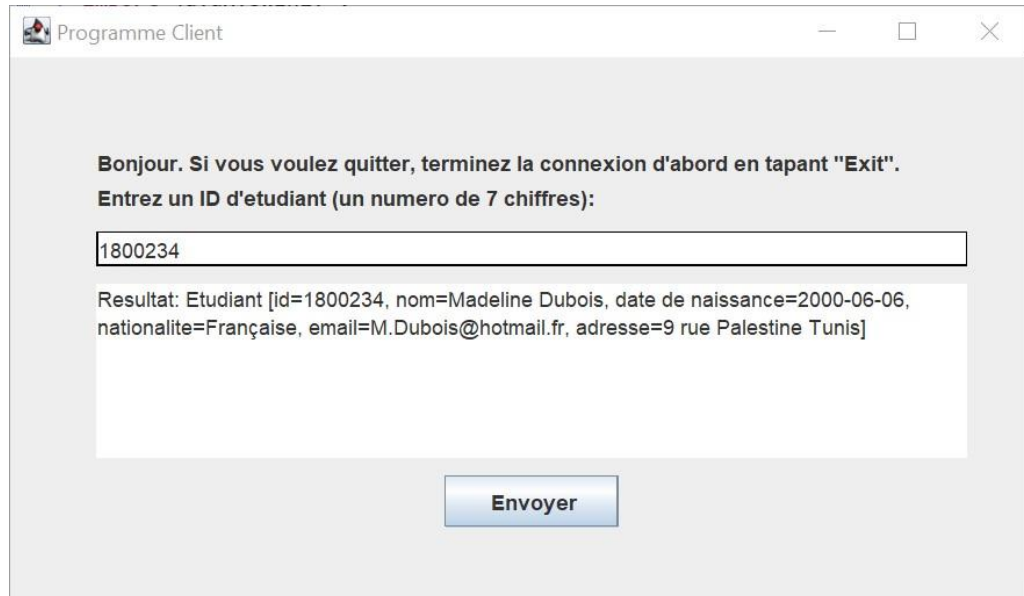


Figure 11- Résultat côté client

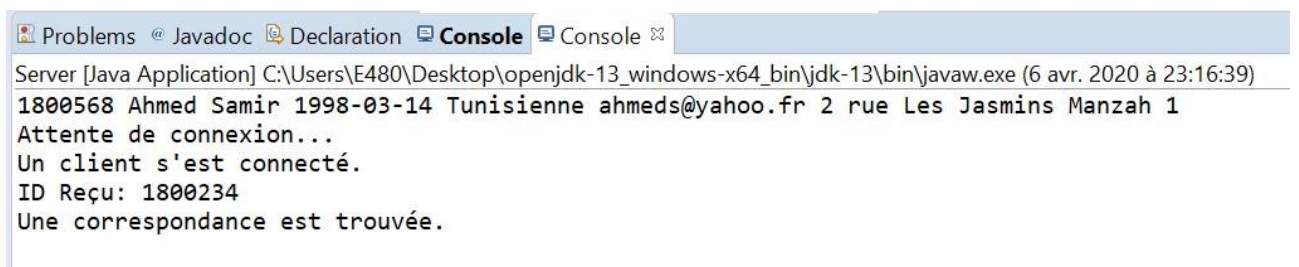


Figure 10- Résultat côté serveur

- La validité du format de l'entrée de données est traitée dans la méthode `actionPerformed(ActionEvent)` de la classe `ClientUI` :

```
//defining send Button action:

@Override
public void actionPerformed(ActionEvent e) {

    //Handle input format:

    int id=0;
    String request = tf.getText();
    String resultStr = "";

    if (!request.equalsIgnoreCase("exit")) {

        if (request == null || request.length() != 7) {

            txt.setText("Requete invalide.");

        } else {

            try {
                id = Integer.parseInt(request);
            } catch (NumberFormatException ex) {
                txt.setText("Requete invalide.");
                return;
            }

        }

        //If valid, write to socket
    }
}
```

On assure que la donnée entrée est : soit « exit » (Pour signaler au serveur de terminer la connexion), soit un nombre de 7 chiffres. Sinon, la donnée n'est pas envoyée dans le socket et l'utilisateur est signalé que sa requête est invalide.

Quand l'utilisateur entre un nombre de 7 chiffres, celui-ci est envoyé dans le socket vers le serveur. Le serveur recherche un `IDetudiant` correspondant à ce nombre dans la BD. S'il trouve une entité correspondante, il instancie avec elle un objet `Etudiant` et l'envoie dans le socket -au client- avec tous ses attributs et ses valeurs. Sinon, il envoie qu'aucun résultat n'est trouvé pour cet `IDetudiant` :

```

// look for student by ID number
System.out.println("ID Reçu: " + message);
sql = "select * from etudiant where id = "+message;
mserver.statement = mserver.con.prepareStatement(sql);
mserver.result = mserver.statement.executeQuery();

if(mserver.result.next()) {

    System.out.println("Une correspondance est trouvée.");
    et = new Etudiant(mserver.result.getInt(1),mserver.result.getString(2),
        mserver.result.getDate(3).toString(),mserver.result.getString(4),
        mserver.result.getString(5),mserver.result.getString(6));
    response = et.toString();

} else {
    response = "Pas de resultat.";
    System.out.println(response);
}

pwSocket.println(response);
pwSocket.flush();

```

Conclusion

Nous avons créé un programme client et un programme serveur en Java qui communique en mode TCP via les Sockets. Le serveur reste en écoute sur le port du socket pour une connexion client. Le client établit une connexion avec le serveur, envoie ses requêtes, reçoit des réponses du serveur puis ferme la connexion. A la réception d'une requête, le serveur accède à une base de données SQL pour récupérer les informations correspondantes à un ID étudiant fourni par le client.

Nous avons utilisé les bibliothèques java.net pour l'implémentation des sockets, java.sql pour la communication avec la BD et Swing et AWT pour l'interface graphique.

Bibliographie

[S. Krakowiak, 2003-2004] *Programmation client-serveur sockets – RPC*, Université Joseph Fourier, France. Lien : <http://lig-membres.imag.fr/krakowia/>.
Plusieurs tutoriels sur <https://www.javatpoint.com/java-tutorial>.

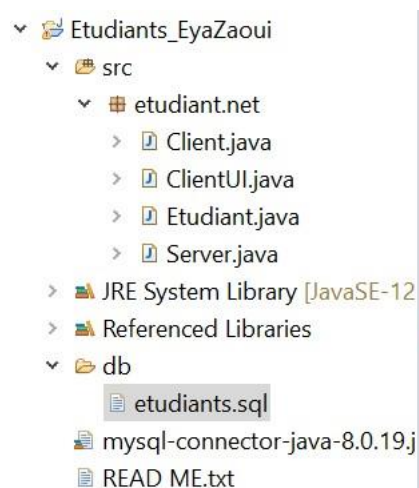
Annexe

❖ Le code source :

Pour voir le code source des différentes classes Java, il suffit d'importer le projet « Etudiants_EyaZaoui » dans un IDE. Veuillez utiliser l'option « importer un projet » de l'IDE pour le charger dans sa totalité, pour qu'il n'y ait pas des problèmes de compilation. Dans Eclipse, cela se fait comme suit :

File>Import...>General>File System , puis vous localiser « Etudiants_EyaZaoui ».

Pour voir le script SQL de la base de données « etudiants.sql », vous pouvez l'ouvrir depuis l'IDE. Vous le trouvez sous le dossier « db » dans le projet.



❖ Modes d'emploi :

- Importer la base de données dans MySql Server et mettre celui-ci EN MARCHE (sur localhost:3306).
- Exécuter Server.java (dans l'IDE).
- Exécuter ClientUI.java (dans l'IDE).