

**Institut Universitaire de Technologie,
Aix-Marseille Université**

**MÉMOIRE
Diplôme Universitaire de Technologie
Spécialité Réseaux et Télécommunications**

**Automatiser la création de trombinoscope
avec une application Android**

Paul MATTEI

Responsable académique : Ivan Madjarov

2020

Table des matières

1	Introduction	3
2	Présentation du projet	3
2.1	Contexte	3
2.2	Missions et objectifs	4
2.3	Environnement de travail	4
3	Travail réalisé	5
3.1	Application Java	5
3.1.1	Description	5
3.1.2	Interface utilisateur	6
3.1.3	Récupération de la saisie client	8
3.1.4	Communication avec le serveur	11
3.1.5	Traitement de la réponse	13
3.2	Serveur Web Node.js	16
3.2.1	Description	16
3.2.2	Protocole de communication	16
3.2.3	Utilisation des routes	17
3.2.4	Traitement des requêtes HTTP	17
3.2.5	Communication avec la base de données	20
3.2.6	Envoi de la réponse au client	21
3.3	Base de donnée MySQL	22
3.3.1	Description	22
3.3.2	Schéma relationnel et caractéristiques	22
3.3.3	Structure de réponse	23
4	Conclusion	24
5	Remerciements	26
6	Glossaire	28
7	Sitographie	30

1 Introduction

Dans le cadre de mon mémoire de fin d'année, j'ai été amené à travailler durant une période de sept semaines sur le développement et la conception d'une application mobile.

Dans un premier temps je commencerai par faire la présentation du projet.

Ensuite je détaillerai l'ensemble du travail que j'ai pu réaliser dans ses aspects généraux comme dans les plus techniques.

Et enfin je conclurai par un bilan du travail accompli ainsi que par les bénéfices qu'à su m'apporter le projet.

2 Présentation du projet

Ce projet a pour but de permettre à l'utilisateur un accès depuis son smartphone en lecture et en écriture correspondant au modèle CRUD* (Ajouter, Consulter, Modifier, Supprimer) à l'ensemble des données présentes sur le système.

De cette manière on peut facilement retrouver et ajouter des données que ce soit le nom, le prénom ou la photo d'un utilisateur par exemple.

2.1 Contexte

D'une manière générale, une application quel que soit sa nature (web, mobile, bureautique, console) qui garantit un accès à des données extérieures établit forcément un lien avec un serveur externe pour interagir avec le client par le biais d'un protocole réseau.

En l'occurrence il s'agit du protocole de communication HyperText Transfer Protocol* (HTTP).

Ce processus de communication client-serveur n'est jamais vraiment perceptible dans les applications puisqu'il s'agit d'une partie purement technique qui fonctionne le plus souvent en arrière-plan de manière asynchrone*. Et à vrai dire il est rare que l'on se préoccupe de savoir si telle information est gérée par son appareil ou si l'information provient partiellement ou entièrement d'un élément externe.

C'est pourtant tout un travail auquel il faut œuvrer sans relâche lorsqu'on travaille dans ce domaine puisque la plupart des applications d'aujourd'hui ne pourraient même plus fonctionner sans protocole de communication, sans communication avec un serveur.

Accessoirement c'est dans un contexte assez particulier que j'ai effectué mon projet puisque ce fut durant une période de confinement globale de la population.

J'ai donc été touché par ce phénomène mais cela ne m'a pas empêché de mener à bien mon projet puisque je pouvais travailler entièrement depuis chez moi.

2.2 Missions et objectifs

L'objectif de mon projet a été la conception et l'élaboration d'une application Android ainsi que d'un serveur web Node.js et d'une base de données MySQL.

Ce sont les trois éléments principaux de mon projet, chacun étant indispensable à sa réalisation.

Que ce soit pour la base de données, pour le serveur ou pour l'application client, ils ont tous les trois des spécificités préétablies initialement dans le cahier des charges propre au projet.

Ces spécificités sont les suivantes :

- La base de données est conçue sur le modèle relationnel qui permet une optimisation dans l'accès mais aussi dans la structure des tables qu'elle comporte.
- Le serveur reproduit une architecture REST* qui fonctionne avec CRUD.
L'utilisation de Express.js* est requise.
- L'application mobile doit prévoir un minimum de dix champs de saisis pour l'utilisateur.
L'application intègre un outil qui permet de choisir entre une image existante et une photo en instantanée lors de la saisi du champ « photo ».

2.3 Environnement de travail

D'une manière inhérente au projet, il a fallu que je prenne en main un environnement de travail qui incluait différents types de logiciel ou de Framework*.

L'installation et la préparation de l'environnement de travail s'est dérouler étape par étape.

Dans un premier temps mon environnement se limitait à Android Studio, c'est-à-dire à la partie client du projet. Puis il m'a fallu installer Node Package Manager* (NPM) pour travailler du côté du serveur avec Express.js.

Lorsque les premiers échanges les plus basiques était établies entre le client et le serveur, j'ai voulu mettre en place la communication avec la base de données. Au début j'utilisais une base de données hébergée et accessible gratuitement en ligne.

Dès lors je me suis rendu compte qu'il y avait un problème : le Round-trip delay time* entre mon ordinateur et la base de données est péniblement long (de l'ordre de la douzaine de seconde).

Une problématique survient alors, comment accéder à une base de données MySQL hébergé gratuitement depuis une API situé sur mon ordinateur.

Il existe que très peu de service de ce genre qui aurait pu satisfaire les conditions de mon projet.

J'aurai pu utiliser ma base de données étudiante fourni par la DOSI, mais cette dernière comme beaucoup d'autres ne sont pas accessibles via les modules *mysql* présent sur NPM et passé un certain délai une erreur de type timeout* survient.

De ce fait j'ai décidé d'opter pour la solution la plus simple mais aussi la plus performante.

Installer la base de données directement sur mon ordinateur, cela s'est mis en place avec l'utilisation du logiciel *MySQL-client* ainsi que *MySQL Workbench*.

L'utilisation de *MySQL Workbench* était auxiliaire sur le plan technique, bien que très utile ce logiciel permet de bénéficier d'une instance de la base de données pour permettre de travailler dessus avec une interface visuelle, ce qui est bien plus simple que d'utiliser le terminal Windows comme il est nativement proposé avec *MySQL-client*.

3 Travail réalisé

J'ai choisi de présenter le travail que j'ai réalisé en trois parties principales.

La présentation sera donc structurée en fonction des axes principaux qui sont : la partie client et serveur puis la base de données.

Dans chacun de ces axes sera la plupart du temps question d'une décomposition relative au modèle CRUD (ajouter/modifier, consulter, supprimer)

La partie client s'appuie essentiellement sur le développement de l'application Android depuis Android Studio en langage Java. Cette phase détermine donc l'intégralité de l'implémentation de l'interface utilisateur et surtout les moyens de communications avec le serveur.

Comme pour chaque application, l'utilisateur doit pouvoir prendre la main en utilisant ses fonctionnalités et naviguer dans ses différents menus.

Ici les fonctionnalités sont les suivantes : permettre à l'utilisateur d'accéder à l'ensemble des données situés dans une base de données externe, lui permettre d'ajouter un profil, de l'éditer et de le supprimer. C'est en fait l'architecture CRUD qui ressort comme étant les fonctionnalités principales de l'application.

La partie serveur s'occupe de traiter les requêtes clients. Le serveur effectue un traitement interne à l'aide des scripts qui lui sont implémentés puis il envoie une réponse adéquate qui sera réceptionné par le client. L'utilisation de Node.js permet un processus de communication réseau programmé en langage javascript.

Les fichiers du serveur sont stockés sur une clé USB, j'ai travaillé sur la portabilité de ce dernier pour le rendre effectif depuis n'importe quel ordinateur.

Initialement le serveur est démarré depuis ces fichiers et est en écoute du trafic local, puis à l'aide du protocole SSH le serveur devient accessible depuis l'adresse publique qui idéalement est celle-ci www.androscope.serveusercontent.com .

La partie base de données est consacrée à la gestion des données utilisateur.

Elle est déployée en interne sur un ordinateur portable mais elle est accessible à l'ensemble du réseau local.

3.1 Application JAVA

3.1.1 Description

L'application Java sera réalisée à l'aide de Android Studio, la version SDK* numéro 19 est requise.

L'application présente un menu principal et trois sous menus.

Depuis le menu principal on a un accès aux différentes fonctionnalités du modèle CRUD.

J'ai choisi de regrouper la partie modifier avec la partie ajouter c'est pour cela qu'il n'y a pas quatre sous parties mais bien trois.

3.1.2 Interface utilisateur

Android studio inclus une interface qui permet de créer dynamiquement différents menus pour notre application et d'y ajouter divers éléments comme des images, du texte ou des boutons qui offre la possibilité d'ajouter des fonctions et des attributs à l'application.

J'ai donc utilisé cette interface dynamique pour créer les quatre menus présents dans mon application et leurs différentes sections.

Lorsque l'on veut créer un menu (ou un layout* en terme technique Android Studio), quelconque soit-il, il faut dans un premier temps choisir comment seront organisés les éléments qu'il contient car dans Android studio il existe différents types de layout qui ne possède pas tous les mêmes caractéristiques. Chaque layout possède plus de cinquante attributs pour personnaliser l'affichage et la disposition du layout et des éléments qu'il contient.

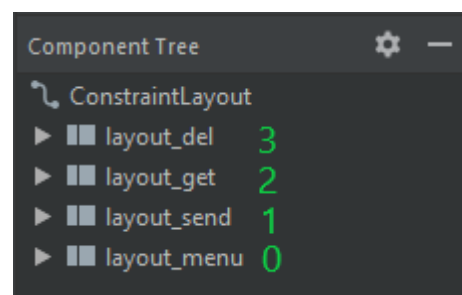
Dans mon cas j'ai utilisé des layout de type relatif (RelativeLayout) qui sont très utiles pour organiser les éléments comme on le souhaite à l'aide d'attribut qui leur sont propres comme :

- *layout_below* / *layout_above* : pour spécifier l'emplacement de l'élément en fonction de la position d'un autre, qu'il soit situé en dessous en haut ou sur les côtés.
- *layout_align* : qui définit un alignement vertical ou horizontal

Pour naviguer dans les différents layout de l'application j'ai inséré depuis le menu principal des boutons qui pointent respectivement sur les menus auxquelles ils correspondent (bouton consulter pour la méthode consulter etc.) Et un système d'indexation des menus, pour indexer les quatre interfaces. Concrètement cela implique que le menu principal a pour index « 0 » et les sous menus envoyer, consulter, supprimer ont les index « 1, 2, 3 »



Interfaces indexées (coté application)



Menus indexés (coté Android Studio)

Pour accéder aux quatre menus de l'application, j'ai implémenté quatre méthodes que j'ai ajoutées dans les attributs *OnClick* des boutons Ajouter/Modifier, Consulter, Supprimer et Retour.

Lorsqu'un clic est effectué sur un des boutons, alors, son traitement correspondant est appelé une fois que l'assignation à une variable globale « index » est faite.

De cette manière je peux situer continuellement sur quel layout l'utilisateur est présent, ce qui sera nécessaire pour l'étape d'après.

```
128 // region Menu Principal
129 public void OnClickAjouter(View v)
130 {
131     index = 1;
132     post = new Post();
133     SetLayout();
134 }
135
136 public void OnClickConsulter(View v)
137 {
138     index = 2;
139     get = new Get();
140     SetLayout();
141 }
142
143 public void OnClickSupprimer(View v)
144 {
145     index = 3;
146     delete = new Delete();
147     SetLayout();
148 }
149
150 // endregion
```

Méthodes référencées sur les 3 boutons du menu principal

L'ajout de boutons pour effectuer des retours depuis les sous menus a aussi été essentiel, cela peut paraître comme une évidence mais le fait de pouvoir effectuer un retour est indispensable.

J'ai donc ajouté et normalisé les boutons de retour, c'est-à-dire qu'ils ont tous le même aspect et surtout ils appellent tous la même méthode. Cette méthode s'occupe de récupérer l'index actuel pour effectuer les traitements nécessaires selon le bouton en question, puis elle affiche le menu principal comme convenu, étant donné qu'un retour ne s'effectue qu'à partir d'un premier et dernier sous menu.

D'autre part j'ai travaillé sur la disposition des champs de saisie de l'utilisateur pour que celui-ci puisse saisir ses données.

Le menu *envoyer* présente des items EditText, ImageView, Radio, Spinner...

Chacun de ces items correspond à un champ de saisie : l'EditText permet d'éditer un texte, le Spinner permet de choisir entre différents éléments d'une liste, et l'ImageView permettra d'afficher et de récupérer une image.

La disposition de ces items et de tous les éléments de manière générale peut se faire à l'aide de multiples attributs. Les ID, l'unité de mesure, le ScrollView* participent à l'élaboration d'une mise

place dynamique des éléments affichés à l'écran. Ce qui est important puisqu'il existe de nombreux terminaux qui possèdent tous des tailles d'écrans différentes et auxquelles on doit pouvoir s'adapter. C'est ce que j'ai essayé de mettre en place en utilisant une unité de mesure qui s'adapte en fonction de la densité de pixel (DP*).

J'ai aussi utilisé des ScrollView vertical pour permettre à l'utilisateur de glisser sur l'écran si le contenu dépasse.

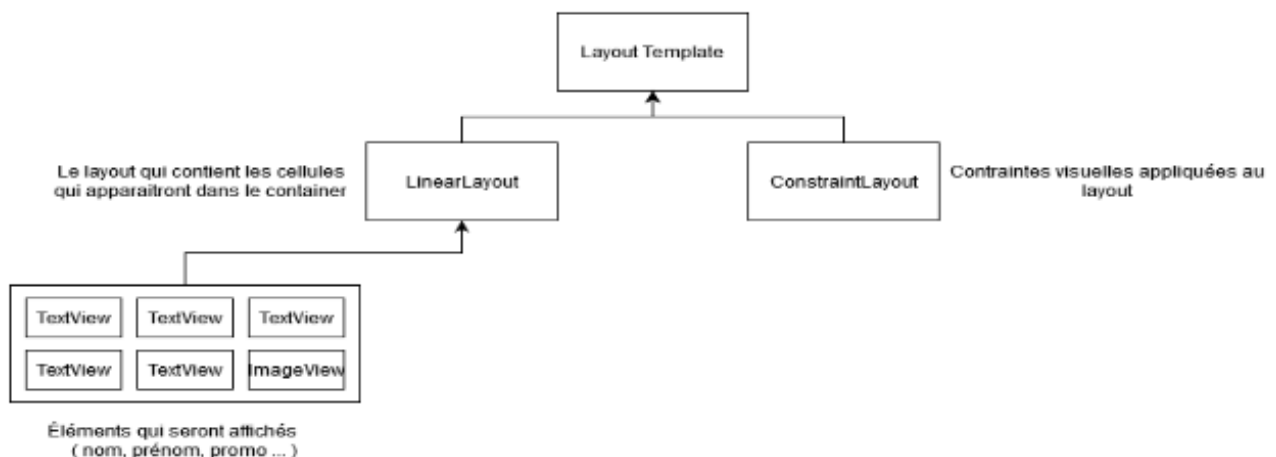
L'application est aussi responsive si l'usager décide de faire une rotation avec l'écran et de passer en vue paysage, ce qui est préférable en l'occurrence car il y a plus d'espace et les éléments visuel ont tendances à être un peu serrés sur un écran relativement petit.

D'autre part, la disposition du menu consulter doit être entièrement dynamique puisque le nombre de ViewGroup* dépendra du nombre d'utilisateur enregistrés dans la BDD.

Pour gérer cette difficulté j'ai créé un Layout nommé « Layout modèle » dont je reparlerai dans la partie traitement de la réponse. Ce layout n'est pas visible ni même présent dans l'interface mais réside dans les fichiers. Ce layout qui sert de modèle permet d'être instancier verticalement une multitude de fois les uns à la suite des autres, ils sont tous contenus dans un autre layout parent. Ce layout modèle contient les éléments nécessaires à l'affichage d'un profil (TextView et ImageView notamment, pour du texte et l'image).

Pour accéder à ces éléments on a besoin de procéder de manière récursive car ils sont inclus dans une sous-section de layout modèle.

En effet le layout instancié depuis les ressources du projet à l'aide d'une méthode nommée « inflate » se décompose de la manière suivante



Décomposition du layout modèle/template

Dernièrement concernant le menu de suppression de profil, il y a plus simplement un champ de saisie et un bouton *supprimer*.

3.1.3 Récupération de la saisie client

Lorsque le client va cliquer sur un bouton que ce soit pour enregistrer, modifier ou supprimer un profil, il y aura toujours un processus qui récupère les valeurs saisies par l'utilisateur et gère les éventuelles erreurs liées à la saisie ou à la connexion.

Ce traitement se fait en trois étapes :

1. Vérifications des champs de saisie
2. Vérification de la connexion locale
3. Vérification de la connexion au serveur

Chacune de ces étapes est un test, si un de ces test échoue alors la communication vers le serveur n'est même pas initiée et on renvoie un message d'erreur approprié au client.

Pour l'étape 1 : On essaie de récupérer tous les champs de saisie et d'assigner leur valeur à des variables privées s'ils sont non nuls.

En cas d'erreur lors de la récupération d'un des champs, ce qui peut arriver si la saisie n'est pas adaptée à son pattern respectif, alors on assigne le champ qui est en cause (l'adresse mail par exemple) dans une variable *error* et on continue de récupérer les autres champs de cette manière.

On vérifie ensuite que la variable *error* est bien vide, autrement on termine le processus en renvoyant à l'utilisateur que le tel ou tel champ de saisie présente une erreur à l'aide d'un Toast*.

Enfin on contrôle la connexion locale à internet et celle vers le serveur, si une des deux connexions ne peut pas être établie on retourne une erreur.

L'étape 2 et 3 sont des traitements qui s'exécutent sur un autre thread*, ce qui implique qu'il suffit ici dans cette méthode *OnClick* du menu Ajouter/Modifier de vérifier si la variable qui correspond au statut de la connexion local/serveur est satisfaite. Mais il existe bien un traitement de fond qui assigne aux variables correspondantes l'état de la connexion. Ce qui a pu me poser un problème lorsque l'état de l'activité* principale était suspendu ou lorsqu'il reprenait, lorsque l'utilisateur éteint son téléphone par exemple. Le problème résidait dans le fait que le thread secondaire continue son exécution même si l'application n'est pas au premier plan, et à chaque fois que l'activité était en « reprise » un nouveau thread était créé et il y pouvait donc y avoir une multitude de même requête essayant de déterminer l'état du serveur qui tournaient en arrière-plan.

Il a donc fallu que je mets en place un système pour éviter que ce thread secondaire soit créé plusieurs fois en même temps.

```
// region Public
public void OnClick()
{
    if(!userInput.PutValueAndContinue())
    {
        String[] error = userInput.error.split( regex: "\\+");
        String mssg = "Champ(s) invalide(s):";
        for(String item : error)
        {
            mssg += " " + item;
        }

        MainActivity.MakeToast(mssg, y: 150);
        return;
    }

    2 if(!TestConnexion.connectedLocally)
    {
        MainActivity.MakeToast( mssg: "Echec de la requête. \nConnexion à internet requise.", y: 150);
        return;
    }

    3 else if(!TestConnexion.connectedToServer) {
        MainActivity.MakeToast( mssg: "Communication avec le serveur impossible.", y: 150);
        return;
    }

    PostAndGetReponse();
}
// endregion
```

Traitement d'un clic submit dans le menu Ajouter

Lors du travail sur la récupération des données il y a trois types de champs de saisies qui se démarquent des autres dans la conception de l'application, les autres étant des champs de saisies où l'on récupère plus simplement la valeur depuis l'input* en question.

D'une part, il y a la récupération d'une image. Pour récupérer l'image depuis la mémoire du téléphone de l'utilisateur ou depuis un cliché en instantané j'ai décidé d'utiliser une librairie existante nommée *ImagePicker*. L'utilisation de cette librairie est relativement simple, elle permet en quelques lignes l'affichage d'un champ de saisie où l'utilisateur doit renseigner une image.

Une fois que l'image est saisie on fait aussi une vérification sur la taille en octets de l'image, il ne faut pas qu'elle soit trop lourde.

Ensuite, on va appliquer un processus de formatage* nécessaire pour réduire la taille de l'image. J'ai en partie eu recours à l'ajout d'une nouvelle classe appelée *Formatter* avec des méthodes statiques de conversion. Le formatage appliqué permet de réduire le poids d'une image de plus de 95%.

Le processus est le suivant :

1. On redimensionne l'image dans un format d'affichage standardisé. (150px*150px)
2. On convertit l'image dans un format PNG.
3. On convertit le format binaire en chaîne de caractère. Cette étape ne favorise pas vraiment la réduction de la taille puisqu'il l'augmente de manière négligeable mais permet de stocker l'image dans une variable *String*.

En suivant ce processus une image initiale de 3Mo de type *Bitmap** devient une variable *String* de 0,06Mo prêt à être envoyé au serveur. Ce système est utilisé pour que la communication avec le serveur soit plus rapide et que les requêtes HTTP soient moins lourdes.

D'autre part j'ai voulu mettre en place un système où chaque utilisateur présent dans la base de données est identifié par un identifiant de quatre caractères minimums.

Dès lors que l'utilisateur entre dans le menu de création de profil depuis l'application, un identifiant unique qui n'est pas déjà présent dans la base de données est distribué automatiquement. C'est-à-dire que le champ de saisie *ID* se voit automatiquement assigner une valeur qui sera valide.

L'utilisateur peut aussi changer de valeur et mettre celle qui lui convient.

Le troisième élément regroupe le *Spinner* et le *Radio* qui sont relativement simple à implémenter mais dénote tout de même d'une autre approche, plus compliquée, qu'avec un input texte basique. Ils sont utilisés respectivement pour la création d'une liste déroulante et pour des cases à cocher.

```
public boolean PutValueAndContinue()
{
    Reset();
    PutNom();
    PutPrenom();
    PutEmail();
    PutTelephone();
    PutPromo();
    PutStudent();
    PutActivite();
    PutTp();
    PutId();
    PutId();
    CheckEditInput();//verif des champs edit text, les autres inputs sont vérifié à la volé ( radio, spinner )
    PutImage();

    if(error.isEmpty())
        System.out.println(error + " Pas d'erreur de saisie input");
    else
        System.out.println(error.split( regex: "\\s+" ).length + "Input(s) invalide(s) " + error);

    return error.length() <= 0;
}
```

Traitement général de la récupération des données

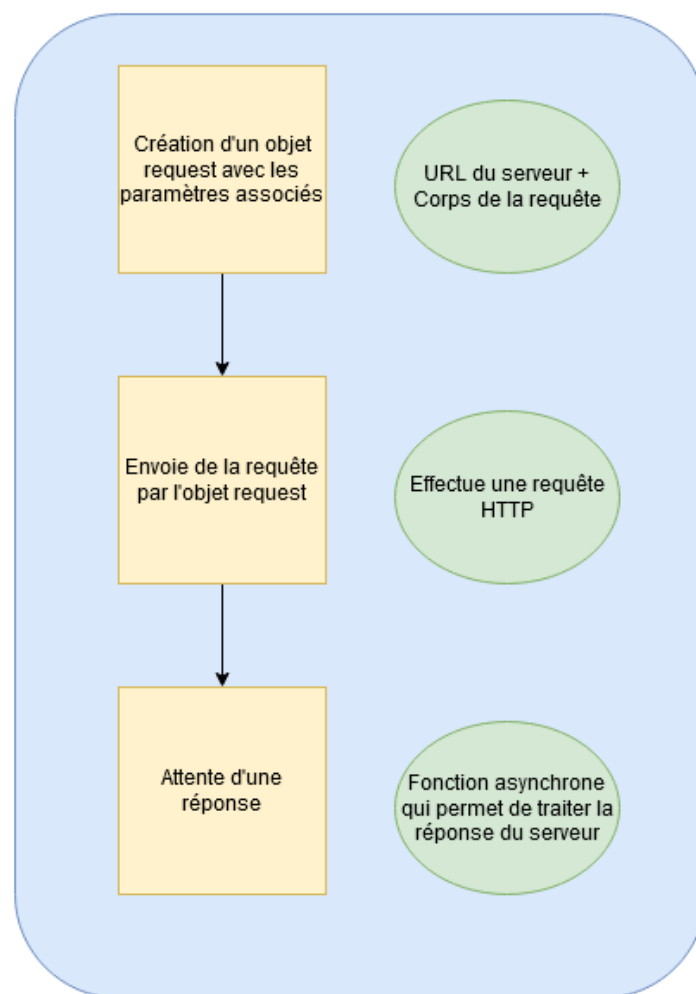
3.1.4 Communication avec le serveur

Une fois toutes les vérifications ont été faites, on peut passer à l'envoi de la requête vers le serveur. Pour effectuer des requêtes HTTP j'ai utilisé une librairie qui est dédié à cela et qui est connue dans ce milieu qui se nomme *Volley*.

Tout d'abord il faut savoir que dès lors qu'on aborde le milieu client-serveur, il est question de communication asynchrone. Cela implique que les réponses aux requêtes émises ne sont pas instantanées et qu'il faut être capable de savoir les traiter en différées.

Il existe différents types de retour à la suite d'une requête HTTP, les plus connus étant les réponses 200, 404, 500 etc. qui indique l'état de la gestion de la requête initiale. Ce dernier pouvant être par exemple conforme et traité sans problème (200) mais il peut aussi présenter des anomalies, des erreurs (500, 404...)

Avec Volley, la procédure pour envoyer une requête HTTP vers un serveur distant est assez simple et on distingue notamment 3 étapes, qui sont les suivantes :



Procédure HTTP avec Volley

Suivant le fonctionnement ci-dessus j'ai mis au point une méthode qui envoie une requête au serveur dans le but de vérifier si l'ID en question existe déjà dans la base de données.

J'ai eu besoin de mettre en place cette requête pour assigner automatiquement un identifiant valide à l'utilisateur. Je vais à présent expliquer en prenant l'exemple de cette requête l'implémentation du principe de fonctionnement de Volley et donc de la communication client-serveur directement depuis Android Studio.

```
final RequestQueue queue = Volley.newRequestQueue(userInput.activity);
final String url = "https://androscope.serveo.net/POST/ID/";

queue.start();
```

Étape n°1 – Envoie d’une requête http avec Volley

Ici on distingue la création d’un objet type RequestQueue propre à Volley ainsi que l’URL correspondant à la requête. Ensuite on appelle la méthode *start()* de l’objet pour indiquer que cette requête doit être la prochaine à être traitée.

```
HashMap m = new HashMap();
m.put("id", id);
JsonObjectRequest jsonObjRequest = new JsonObjectRequest(Request.Method.POST, url, new JSONObject(m))
```

Étape n°2 - Envoie d’une requête http avec Volley

La requête HTTP est envoyée à la suite de cette ligne ci-dessus. Le type, l’url et le corps de la requête doivent être renseignés. Ici il s’agit d’une requête POST ayant comme url {racine-serveur}/POST/ID/.

Le corps de la requête est un ID créé aléatoirement coté client sous la forme d’une variable JSON* pour faciliter l’acheminement du message.

```
response ->
{
    JSONObject reply, message = null;
    int count_index;
    try
    {
        reply = response.getJSONObject("reply");
        message = reply.getJSONObject("message");
        System.out.println(reply.optString( name: "code") + " " + message.optString( name: "count"));
    }
    catch (JSONException e)
    {
        e.printStackTrace();
    }

    count_index = Integer.parseInt(message.optString( name: "count"));
    if(count_index > 0)
    {
        SetId();
    }
    else
    {
        userInput.getEdt_Id().setText(id);
    }
}, error -> System.out.println("Erreur de communication vers le serveur:" + error));
jsonObjRequest.setRetryPolicy(new DefaultRetryPolicy(
    initialTimeoutMs: 0,
    maxNumRetries: 0,
    DefaultRetryPolicy.DEFAULT_BACKOFF_MULT));
queue.add(jsonObjRequest);
```

Étape n°3 - Envoie d’une requête http avec Volley

L'étape numéro 3 inclut deux parties qui sont le traitement lors d'une réponse par le serveur et le traitement lorsque la requête n'a pas reçu de réponse du serveur ce qui peut arriver si la connexion n'est pas établie ou en cas d'anomalie interne coté serveur par exemple.

De la même manière que pour la requête ci-dessus qui vérifie si l'ID est unique dans la base de données, j'ai pu élaborer d'autres requêtes notamment pour la partie ajouter, modifier, consulter et supprimer. Je vais donc essayer d'explicitier les méthodes que j'ai conçues pour ces quatre autres requêtes sans trop rentrer dans les détails du fonctionnement car le principe de fonctionnement reste la même décomposition en 3 étapes.

Pour la partie Ajouter et Modifier, qui pour rappel est une seule et même partie dans l'application, j'ai procédé d'une manière un peu particulière comparé aux autres parties.

Lorsqu'on veut ajouter un profil il faut que toutes les informations soit présentent dans la requête, c'est-à-dire qu'il doit y a avoir au minimum 10 champs présent dans le corps du message autrement ce n'est pas possible d'ajouter un profil et le programme retourne une erreur.

Jusqu'à là rien d'étonnant, si ce n'est que lorsqu'on veut modifier un profil il faut pouvoir modifier certaines informations uniquement et donc avoir la possibilité d'envoyer une requête avec un nombre de champs limité. Pour être clair si l'on veut modifier uniquement sa poursuite d'études, il ne faut pas que l'on soit obligé de remplir tous les autres champs pour que la requête puisse aboutir et ça pour des raisons de simplicité évidente.

Pour concilier le fait qu'une requête doit être pleine (Ajouter) et qu'à la fois elle doit pouvoir être limité (Modifier), j'ai utilisé la solution qui s'appuie sur un traitement coté serveur, notamment sur le fait de savoir si l'ID renseigné dans la requête existe déjà, le cas échéant le système fait face à un utilisateur déjà enregistré qui veut modifier ses données. Autrement le système reconnaît que l'ID est introuvable et que l'utilisateur souhaite ajouter un nouveau profil.

Je reviendrai sur ce fonctionnement par la suite dans la partie serveur.

Quant à la partie Consulter, il a fallu d'une manière plus simple pour l'envoi de la requête, spécifier l'adresse en question qui dans ce cas est {racine-serveur}/GET/DATA et puis le type de la requête, qui est ici une requête GET. En effet nous n'avons pas vraiment besoin d'inclure un corps dans le message comme c'est le cas avec un requête POST.

C'est donc uniquement avec l'URL que le serveur est capable de retourner les informations à la suite d'un traitement interne.

Enfin la suppression d'un profil s'effectue par le biais d'une requête HEAD ou POST sur l'url {racine-serveur}/DELETE/{id-concerné}. Ici non plus l'envoi de la demande n'est pas très compliqué puisqu'il suffit de renseigner l'ID utilisateur dans le corps ou l'url de la requête HTTP pour lequel on veut supprimer ses données, c'est ensuite au serveur de traiter la demande et de renvoyer une réponse.

3.1.5 Traitement de la réponse

A ce stade le client a tenté d'initier la communication avec le serveur et il attend sa réponse. C'est à présent à la réception et au traitement de la réponse d'être développés.

Durant l'avancement de mon projet j'ai eu comme initiative de normaliser les réponses envoyées par le serveur sous un même format ce qui m'a permis par la suite de bénéficier d'un traitement simplifié coté client.

Lors de la réception d'une réponse on va dans un premier temps lire le statut de la requête afin de se situer d'une manière générale sur le traitement qu'a reçu de cette dernière.

Ensuite on pourra selon le statut de la requête effectuer les opérations nécessaires.
Pour pouvoir récupérer le statut de la requête, on doit d'abord la décomposer.
Puisque celle-ci est normalisé, la décomposition est plus simple et est toujours la même.
Globalement on a deux parties principales qui sont le *code* et le *message* et des sous parties.
Je n'insiste pas sur la composition de la requête dès à présent puisque c'est au niveau du serveur que la réponse est formulée.
Une fois décomposer on peut alors récupérer et assigner les valeurs des champs qu'elle contient pour les réutiliser dans l'application.

Que ce soit dans la partie Ajouter/Modifier ou dans la partie Supprimer le principe est le même, c'est-à-dire qu'on va afficher un Toast de retour. Ce Toast présentera le même message que celui contenu dans le corps de la requête, ce message est directement ajusté selon divers facteurs (nombre de table affectés, id existant, statut de la requête etc.) depuis le serveur.
De cette manière l'utilisateur est informé de la progression de sa demande même si la requête initiale n'a pas pu aboutir.
Typiquement sur le menu Ajouter l'utilisateur recevra un Toast qui lui indiquera combien de table et combien d'erreur il y'a eu pendant sa requête. Sur le menu Supprimer, l'utilisateur pourra savoir si son profil a bien été supprimé ou si la requête a échoué.

Au niveau de la consultation des données, le traitement de la réponse s'est avéré nettement plus compliqué par la taille des requêtes car j'ai décidé de faire en sorte de recevoir une seule requête en guise de réponse qui contient toutes les informations utilisateur de la base de données.

J'ai dans un premier temps créé une nouvelle classe nommée « User » qui possède tous les attributs d'un utilisateur (nom, prénom, email etc.) et qui permettra de créer une liste d'utilisateur.

Cette liste contiendra à la fin du processus l'entièreté des informations utilisateurs, ce qui les rendra très simple d'accès.

Pour que cette liste soit complète j'ai décidé de parcourir à l'aide d'une boucle *for* tous les utilisateurs présents en tant qu'objet dans la réponse reçu. Pour chaque utilisateur on instancie un objet *User* directement dans la liste à l'aide de la méthode *Liste.add()* et on définit ses attributs à la volée afin qu'on ait plus besoin de les modifier par la suite.

(J'ai regroupé pour des raisons de simplicité les champs email et téléphone dans une variable *contact* ainsi que les champs TP, TD et la poursuite d'études dans une variable *etudiant*.)

```
ArrayList<User> users = new ArrayList<>();
int max = usersJ.length();
for (int i = 0; i < max; i++)
{
    JSONObject jsonobject = usersJ.getJSONObject(i);
    String nom = jsonobject.optString( name: "nom");
    String prenom = jsonobject.optString( name: "prenom");
    String photo = jsonobject.getString( name: "photo");
    String contact = jsonobject.getString( name: "contact");
    String promo = jsonobject.getString( name: "promo");
    String etudiant = jsonobject.getString( name: "etudiant");
    String ancien = jsonobject.getString( name: "ancien");
    users.add(new User(nom, prenom, photo, contact, promo, etudiant, ancien));
}
```

Traitement d'une réponse depuis le menu consulter

A présent qu'on possède la liste *users* qui contient tous les utilisateurs et leurs spécifiés, on peut passer à son affichage pour que l'utilisateur puisse les consulter. Pour réaliser cette affichage le programme procède en 3 étapes (Figure 1) :

- 1) Création des layout modèle selon le nombre d'utilisateur
- 2) Insertion de toute les Views* de chaque layout modèle dans une liste (Figure 2)
- 3) Assignation des valeurs de l'utilisateur à ce modèle

Le code qui suit peut sembler compliqué à lire car je l'ai élaboré dans son intégralité dans un contexte particulier qui est notamment relatif au layout modèle et à sa structure que j'ai présenté précédemment. D'une manière conceptuelle et simplifiée le programme s'occupe donc d'instancier un nombre de layout correspondant au nombre d'utilisateur et il affecte à ces layout les attributs des utilisateurs (nom, prénom, promo...)

```
public void Instantiate()
{
    int ressources = R.layout.layout_template;

    for(int i = 0; i < size; i++)
    {
        activity.getLayoutInflater().inflate(ressources, layoutContainer); // 1) création des layout
        InsertRows((ViewGroup) layoutContainer.getChildAt(i)); // 2) création de la liste qui contient toute les views
    }
    SetRowsValue(); // 3) Assignment des valeurs aux views
}
```

Figure 1 : Procédure d'affichage des données utilisateurs

```
private void InsertRows(ViewGroup parent)
{
    ViewGroup subroot = (ViewGroup) parent.getChildAt( index: 0);
    int max = subroot.getChildCount();// nombre maximum d'element enfant
    for(int i = 0; i < max; i++)
    {
        View v;
        if(i == 0)//on accède au nom & prénom qui sont dans le même viewgroup
        {
            ViewGroup vc = ((ViewGroup)subroot.getChildAt(i));
            v = vc.getChildAt( index: 0);
            views.add(v);
            v = vc.getChildAt( index: 1);
        }
        else//on accède à tous les autres éléments (contat, promo, image ...)
            v = subroot.getChildAt(i);

        views.add(v);//on ajoute tous les élément de chaque layout model dans cette liste
    }
}
```

Figure 2 : Création d'une liste contenant toutes les Views de chaque modèle layout

3.2 Serveur Web Node.js

3.2.1 Description

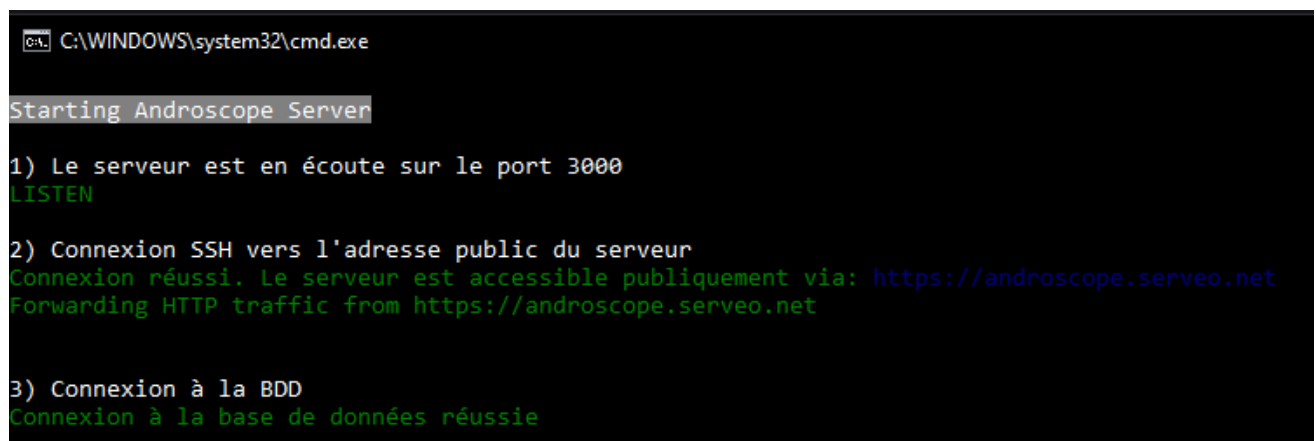
Le serveur fonctionne avec Node.js qui permet d'utiliser le langage JavaScript pour traiter les requêtes qu'il reçoit. Node.js est très rapide notamment grâce à son moteur V8* et du fait qu'il fonctionne de manière asynchrone. Node.js est monothread* et traite les requêtes une par une, il se comporte différemment uniquement si une requête devient un peu longue contrairement à un fonctionne simultanée où le serveur est tout le temps capable de lancer plusieurs processus pour les traiter en parallèle.

Dans ce projet le serveur utilise Express.js qui est un framework qui offre les outils de bases pour concevoir l'application Node.js, il y a notamment les routes qui correspondent aux différentes URL auxquelles le serveur doit répondre.

C'est à l'aide d'une commande NPM que l'on démarre le serveur, j'ai ajouté cette commande dans un fichier batch pour automatiser son démarrage, de cette manière un simple double clic suffit pour lancer le serveur.

D'autre part j'ai ajouté un système de debug* (Figure 1) qui est très efficace pour situer l'état du serveur à son démarrage mais aussi lors du interne de chaque requête.

Lorsque le serveur démarre, un interpréteur de commande Windows indique étape par étape ce qu'il se passe et comment cela se passe t'il. J'ai aussi ajouté un module dédié au couleur dans un terminal Windows pour rendre la visibilité beaucoup plus simple.



```
C:\WINDOWS\system32\cmd.exe
Starting Androscope Server

1) Le serveur est en écoute sur le port 3000
LISTEN

2) Connexion SSH vers l'adresse public du serveur
Connexion réussie. Le serveur est accessible publiquement via: https://androscope.serveo.net
Forwarding HTTP traffic from https://androscope.serveo.net

3) Connexion à la BDD
Connexion à la base de données réussie
```

Figure 1 : Démarrage type du serveur

3.2.2 Protocole de communication

Le protocole de communication mise en place sur le serveur est le protocole SSH qui est un protocole de communication réseau sécurisé.

L'utilisation de ce protocole s'est avérée nécessaire puisque Node.js et Express.js ne permette pas de rendre le serveur accessible publiquement mais uniquement depuis le réseau local sur lequel il est déployé.

Le serveur est dans un premier temps en écoute local sur un port spécifique, puis une connexion SSH sur l'adresse du public du serveur est effectué pour rediriger les paquets réseaux vers le serveur (port forwarding). Dans mon cas j'utilise un service gratuit qui s'appelle « serveo.net » et qui permet d'établir la redirection de paquet à l'aide d'une commande SSH. Ce service gratuit fonctionne très bien mais il est arrivé à plusieurs reprise que celui-ci soit indisponible durant quelques jours.

C'est pourquoi j'ai aussi utilisé un second service gratuit lorsque le premier n'était pas disponible mais celui-ci n'offre pas un sous-domaine fixe. Ce qui implique qu'à chaque redémarrage du serveur il faut aussi changer les variables *url* qui correspondent à l'URL du serveur dans l'application Android, ce qui n'est pas vraiment souhaité.

3.2.3 Utilisation des routes

C'est avec Express.js que le serveur peut implémenter des routes.

Une fois l'installation du package terminée on peut directement commencer à développer les routes du serveur. Le serveur dispose au total 7 routes qui permettent à la suite d'une connexion sur celles-ci d'effectuer un traitement interne spécifique. Il y a d'une part les routes présentent pour répondre aux requêtes du model CRUD. Mais aussi d'autre route que j'ai décidé d'implémenter, notamment celle pour répondre aux requêtes client que j'ai décrite plus tôt (savoir si l'ID est existant) mais aussi une autre route qui permet de renvoyer le statut du serveur et de la base de données.

L'écriture et la syntaxe de ces routes (Figure 1) nous permet de spécifier une URL correspondante ainsi qu'un type de requête (GET, POST...).

```
app.get('/', function (req, res)
{
    incoming_request("GET", "/");
    res.send('Bienvenue sur le serveur Androscope');
})
```

Figure 1 : Formation d'une route Express.js

On constate que sur la formation de la route (Figure 1) il y a d'une part le module Express.js qui s'occupe de créer la route et d'autre part le traitement interne qui est développé entre les crochets. La fonction utilisée dans la route prend comme argument la requête et l'objet de réponse qui permettra d'envoyer notre réponse au client.

Ici la réponse au client est toute simple puisqu'il s'agit d'une connexion sur la racine du serveur, on envoie simplement un message de bienvenue.

3.2.4 Traitement des requêtes HTTP

Le traitement de la requête est normalisé sur le serveur, j'ai décidé de reproduire le même schéma d'exécution pour n'importe quelle route. Ce qui a comme avantage de situer l'acheminement de n'importe quelle requête facilement.

```
35 app.post('/POST/DATA/', async function (req, res)
36 {
37     incoming_request("POST", "/POST/DATA/", req.body); // 1) LOG
38
39     var reply = await connexion_bdd.set(req.body); // 2) reponse json
40
41     reponse_from_server(res, reply, index_request); // 3) envoie de la reponse
42 })
43
44 app.get('/GET/DATA/', async function (req, res)
45 {
46     incoming_request("GET", "/GET/DATA/", "Données table MySQL"); // 1) LOG
47
48     var reply = await connexion_bdd.get(); // 2) reponse json
49
50     reponse_from_server(res, reply, index_request); // 3) envoie de la reponse
51 })
52
53
54 app.post('/DELETE/', async function (req, res)
55 {
56     incoming_request("DELETE", "/DELETE/", req.body); // 1) LOG
57
58     var reply = await connexion_bdd.delete(req.body); // 2) reponse json
59
60     reponse_from_server(res, reply, index_request); // 3) envoie de la reponse
61 })
```

Traitement normalisé des requêtes

Le processus lors de la réception d'une requête se fait en donc en 3 étapes comme le montre l'image ci-dessus :

- 1) Un debug afficher sur l'invite de commande
- 2) Le traitement de la requête et la formation de la réponse
- 3) Envoie de la réponse au client

Dans cette partie il sera surtout question du traitement de la requête et de la formation de la réponse. Cette partie sera donc dédiée à l'aspect programmation du traitement. Cet aspect sera introduit par son concept plus général qui permet une meilleure compréhension du programme. Le traitement est différent pour chaque type requête, je vais donc présenter le traitement appliqué aux requêtes du model CRUD.

Lorsqu'on souhaite ajouter ou modifier un profil depuis l'application, voici le traitement effectuer au niveau du serveur. (Figure 1)

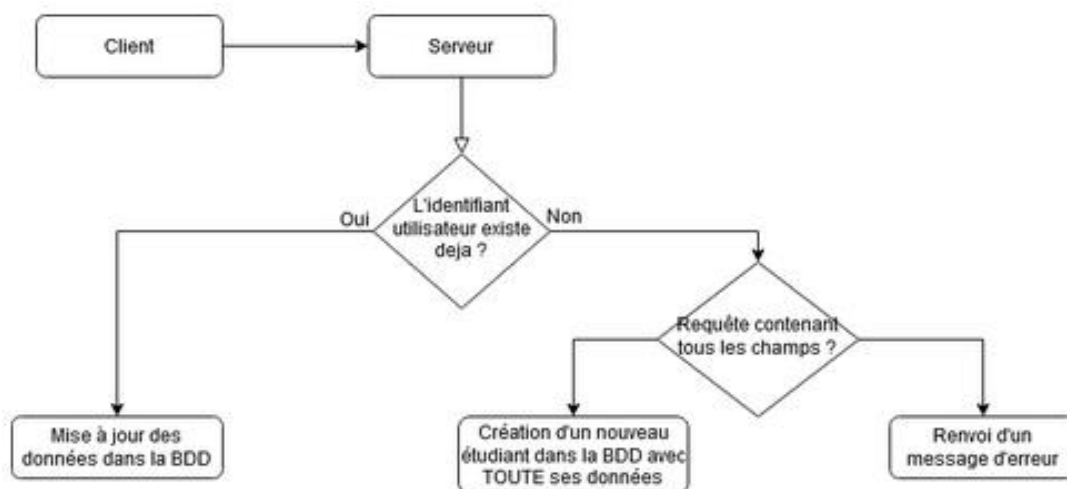


Figure 1 : Traitement de la requête ajouter/modifier

En réalité la fonction qui s'occupe de créer la requête MySQL pour communiquer avec la base de données est la même pour un utilisateur existant que pour un nouvel utilisateur.

A l'instar de l'application Android qui regroupe la partie Ajouter et Modifier, la fonction interne du serveur qui s'en occupe regroupe aussi ces deux éléments.

Cette fonction prend donc en paramètre le corps de la requête et un boolean définit au préalable qui indique s'il s'agit d'un nouvel utilisateur. Elle permet de retourner la requête MySQL adéquat en une seule variable de type String. Le fonctionnement est le suivant :

La fonction s'occupe d'abord de définir le début de la requête (INSERT / UPDATE) (Figure 2) et ceci pour toutes les tables concernées.

Ensuite elle ajoute les valeurs correspondantes à chaque table suivant la syntaxe relative à MySQL. (Figure 3). Ces valeurs sont présentes dans le corps de la requête.

La réalisation de cette fonction ainsi que son fonctionnement ne sont pas des moindres puisqu'elle permet à elle seule de définir toutes les requêtes MySQL imbriquées à la suite des autres en une seule variable, ce qui implique accessoirement que le code peut être compliqué à lire.

```

if(user_exist)// UPDATE DEBUT DE REQUETE
{
    if(i == 0 && tabToModify.includes("etudiant"))
        sql += "UPDATE "+table_etudiant+" SET ";

    else if(i == tab_etudiant && tabToModify.includes("groupe"))
        sql += "UPDATE "+table_groupe+" SET ";

    else if(i == tab_groupe && tabToModify.includes("photo"))
        sql += "UPDATE "+table_photo+" SET ";

    else if(i == tab_photo && tabToModify.includes("poursuite"))
        sql += "UPDATE "+table_poursuite+" SET ";
}
else//INSERT INTO DEBUT DE REQUETE
{
    if(i==0 && tabToModify.includes("etudiant"))
        sql += "INSERT INTO "+table_etudiant+" VALUES ('"+json.id+"', ";

    else if(i == tab_etudiant && tabToModify.includes("groupe"))
        sql += "INSERT INTO "+table_groupe+" VALUES ('"+json.id+"', ";

    else if(i == tab_groupe && tabToModify.includes("photo"))
        sql += "INSERT INTO "+table_photo+" VALUES ('"+json.id+"', ";

    else if(i == tab_photo && tabToModify.includes("poursuite"))
        sql += "INSERT INTO "+table_poursuite+" VALUES ('"+json.id+"', ";
}

```

Figure 2 : Traitement d'une requête ajouter/modifier

```

if(user_exist)//UPDATE
{
    index ++;

    if(json[item].length > 0)
        sql += item+"='"+json[item]+'";
    else
    {
        sql += item+"=null";
    }
    if(index<nb_toModify_etu || index<nb_toModify_etu+nb_toModify_groupe && index > nb_toModify_etu||
    index<nb_toModify_etu+nb_toModify_groupe+nb_toModify_photo && index>nb_toModify_etu+nb_toModify_groupe|| index <
    nb_toModify_etu+nb_toModify_groupe+nb_toModify_photo+nb_toModify_poursuite && index
    >nb_toModify_etu+nb_toModify_groupe+nb_toModify_photo)//encore des éléments à mettre dans la requête sql
    {
        sql += " ,";
    }
    else
    {
        var id = i < tab_etudiant ? "id":(i<tab_groupe ? "id_groupe" : i<tab_photo ? "id_photo": "id_poursuite");
        sql += " WHERE "+id+"='"+json[id]+'";
    }
}
else// INSERT INTO
{
    index ++;

    if(json[item] != undefined && json[item] != null && json[item].length > 0)
        sql += " '"+json[item]+'";
    else
    {
        sql += "null";
    }
    if(index<nb_toModify_etu || index<nb_toModify_etu+nb_toModify_groupe && index > nb_toModify_etu||
    index<nb_toModify_etu+nb_toModify_groupe+nb_toModify_photo && index>nb_toModify_etu+nb_toModify_groupe|| index <
    nb_toModify_etu+nb_toModify_groupe+nb_toModify_photo+nb_toModify_poursuite && index
    >nb_toModify_etu+nb_toModify_groupe+nb_toModify_photo)//encore des éléments à mettre dans la requête sql
    {
        sql += " ,";
    }
    else
    {
        sql += " );";
    }
}
}

```

Figure 3 : Traitement d'une requête ajouter/modifier

Concernant la partie consulter il faut que le serveur puisse récupérer chaque donnée de chaque table présentent dans la BDD.

Pour récupérer ces données j'ai utilisé une fonction qui de manière asynchrone va récupérer les données des tables une par une, cette fonction retournera en une seule variable toutes les données.

```
454 async function get_all_tables()
455 {
456
457     var rep = {"code":"","data":{"etudiant":[],"poursuite":[],"groupe":[],"photo":[]}}
458     var etu = await new Promise(resultat =>
459     { ... });
474
475     var poursuite = await new Promise(resultat =>
476     { ... });
491
492     var groupe = await new Promise(resultat =>
493     { ... });
508
509     var photo = await new Promise(resultat =>
510     { ... });
525     return rep;
526
527 }
```

Figure 4 : Traitement d'une requête consulter

Quant à la partie supprimer, ce fut bien plus simple puisqu'il suffisait dans un premier temps de vérifier si l'ID utilisateur présent dans le corps du message était enregistré dans la BDD.

Ensuite le programme définit simplement les 4 requêtes MySQL (Suppression de 4 tables).

```
var sql_query =
    "DELETE FROM " + table_groupe + " WHERE id_groupe='"+id+"'";
    "DELETE FROM " + table_photo + " WHERE id_photo='"+id+"'";
    "DELETE FROM " + table_poursuite + " WHERE id_poursuite='"+id+"'";
    "DELETE FROM " + table_etudiant + " WHERE id='"+id+'";
```

Figure 5 : Traitement d'une requête supprimer

À la suite des différents processus qui sont propres à chaque requête présentée ci-dessus, on dénote qu'il y a plusieurs éléments qui reviennent à chaque fois.

D'une part l'utilisation de fonction asynchrone qui est indispensable pour pouvoir traiter les requêtes les unes après les autres. Avec JavaScript il faut utiliser le mot clé *await* qui attend une promesse et bloque le traitement jusqu'à ce que cette promesse soit retournée.

D'autre part il existe certaines méthodes dont on peut se servir pour accéder au contenu de la requête, notamment *hasOwnProperty* qui renvoie *true* si la requête contient un élément clé.

C'est comme cela que l'on devra décortiquer la requête pour savoir ce qu'elle contient.

3.2.5 Communication avec la base de données

Pour communiquer avec la base de données depuis Node.js il faut d'abord installer le package nommée « *mysql* ». Ensuite on peut initialiser la connexion avec la base de données en utilisant la méthode *createConnection* qui prend comme paramètre des champs nécessaire comme le champ *host*, *user* et *password* et d'autres champs optionnels comme le port ou le *multipleStatements*.*.

Enfin on peut démarrer un échange avec la base de données avec la méthode nommée « *query* » qui prend comme argument la requête MySQL au format String.

3.2.6 Envoie de la réponse au client

Le format de la réponse construite depuis le serveur est normalisé pour faciliter son traitement à l'aide de méthode commune et aussi pour faciliter sa réception coté client. Il y a d'un côté le schéma de la réponse au format JSON (Figure 1) et de l'autre côté son implémentation et son traitement sur le serveur (Figure 2).

On constate alors que l'implémentation est quasiment la même que la structure du schéma au détail près que le champ « info » n'est pas présent dans l'implémentation car dans la pratique je n'en ai pas eu besoin.

```
1 {↵
2   "code":      // Statut HTTP (200, 404 ...)↵
3   "message":   // Contenu de la requête↵
4   [↵
5     "info":    // Information sur le contenu↵
6     "user":    // Représente chaque utilisateur↵
7     [↵      // et contient ses informations↵
8       "nom":↵
9       "prenom":↵
10      "contact":↵
11      "promo":↵
12      "etudiant":↵
13      "ancien":↵
14      "photo":↵
15    ]↵
16  ]↵
17 }
```

Figure 1 : Schéma de standard de réponse

```
<Request#2>User made a GET request on /GET/DATA ( 29-5-2020 16:21:16 )
<parameter> "Donnée table MySQL"
{
  code: '200 OK+200 OK+200 OK+200 OK+',
  message: {
    users: [
      [Object], [Object],
      [Object], [Object],
      [Object], [Object],
      [Object], [Object],
      [Object], [Object],
      [Object], [Object]
    ]
  }
}
```

Figure 2 : Implémentation de la réponse (Debug)

Une fois que le traitement interne est réalisé, la réponse est alors envoyée au client et s'effectue de manière très brève à l'aide de Express.js.

Pour envoyer la réponse au client, le système utilise notamment la méthode *send* de l'objet qui correspond au client qui a initialisé la communication, elle prend comme paramètre le statut HTTP de la réponse ainsi que le corps de son message.

Il peut arriver que l'on spécifie un statut d'erreur manuellement à la suite d'une requête qui par exemple ne correspond à aucun ID utilisateur dans la base de données et qui ne contient pas tous les champs, dans ce cas, on précise aussi dans le champ message de la réponse le problème rencontré.

3.3 Base de données MySQL

3.3.1 Description

La nécessité d'une base de données dans un tel projet repose sur le fait que les utilisateurs doivent avoir un contrôle sur des données hébergées. Que ce soit leur nom, prénom, photo ... ces données doivent être consultables et accessibles parmi l'ensemble des utilisateurs. C'est exactement le rôle d'une base de données, en l'occurrence MySQL.

Une base de données MySQL a comme avantage une haute compatibilité, en l'occurrence un package NPM permet la communication avec celle-ci de manière très exhaustive. MySQL est aussi gratuit et très performant.

3.3.2 Schéma relationnel et caractéristiques

MySQL est un système de gestion de données relationnelles c'est-à-dire que les données stockées sont divisées en plusieurs parties (qu'on appelle les tables) plutôt que de les regrouper dans un ensemble commun.

Dans ce projet il est question de quatre tables, les utilisateurs ont donc des données stockées dans quatre tables différentes. Pour que les données de chaque table soient reliées avec leur utilisateur respectif on utilise des clés étrangères qui pointe vers la clé primaire de la table étudiant. De cette manière la gestion des données est organisée et ordonnée.

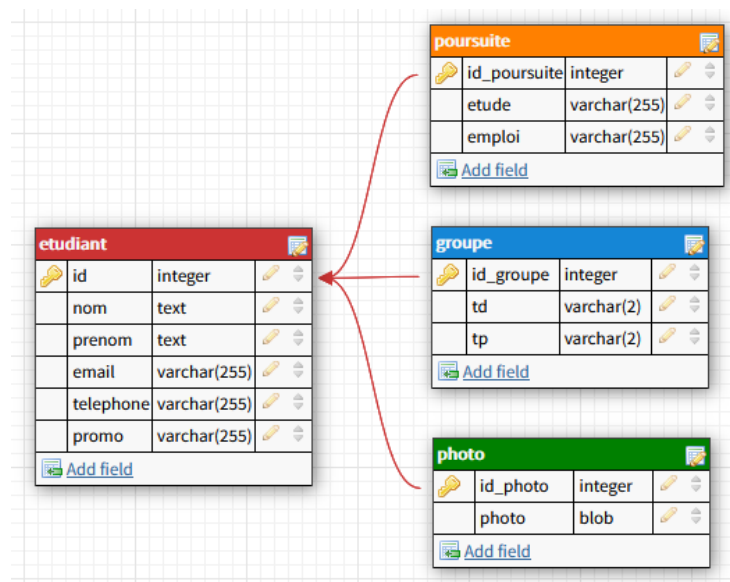


Schéma relationnel base de données

3.3.3 Structure de réponse

Lorsqu'on envoie une requête MySQL depuis Node.js, la base de données retourne toujours un objet de type RowDataPacket en guise de réponse.

Lors d'une requête SELECT cet objet retourné contient chacun des éléments sélectionnés dans une liste. On peut y accéder de la même façon qu'on accède à une liste en javascript, ce qui nous permet d'une manière assez simple de réutiliser les informations contenues dans le RowDataPacket.

Pour les autres requêtes (INSERT, DELETE, UPDATE) l'objet retourné par MySQL contiendra uniquement des informations comme le nombre de tables affectés ou le nombre d'erreur puisque'on ne demande pas une réponse spécifique.

4 Conclusion

A la fin des sept semaines durant lesquelles j'ai travaillé sur ce projet, l'objectif principal d'automatiser le trombinoscope via une application Android ainsi que les contraintes liées au projet on étaient respectés :

- L'application Android permet à l'utilisateur de gérer et d'accéder aux données présentent dans la base de données MySQL en suivant le modèle CRUD (Figure 1)
- La base de données est déployée et accessible via le serveur. (Figure 2)
- Le serveur répond aux requêtes clients en s'appuyant sur l'architecture REST. (Figure 3)

D'autre part je pense que ces semaines de travail ont su m'apporter des connaissances substantielles dans le domaine de la gestion de projet mais aussi en programmation notamment en Java et Javascript.

J'ai travaillé sur des aspects intéressants comme la communication réseaux entre un client et un serveur ou l'implémentation de route via Express.js et j'ai aussi redécouvert des notions de programmation comme l'exécution de fonction asynchrone ou l'utilisation de thread.

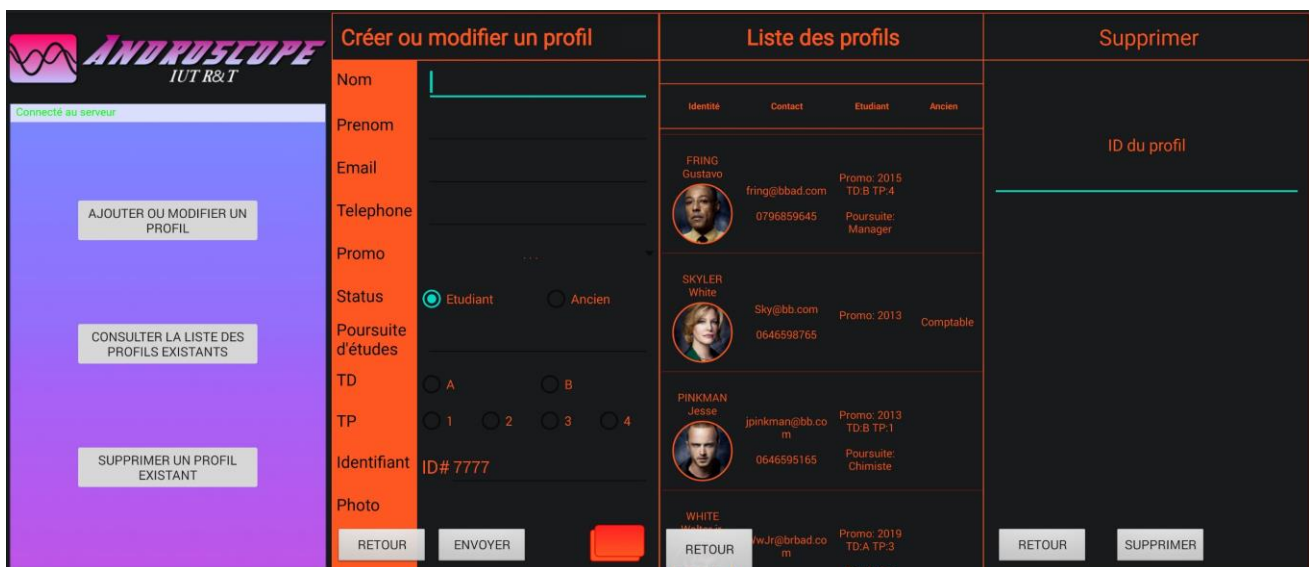


Figure 1 : Application Android et ses menus

nom	prenom	email	telephone	promo
Schrader	Hank	HANKSchrader@email.com	0656985847	2018
Goodman	Saul	Gman@bbad.com	0646595165	2013
Fring	Gustavo	fring@bbad.com	0796859645	2015
Skyler	White	Sky@bb.com	0646598765	2013
Pinkman	Jesse	jpinkman@bb.com	0646595165	2013
White	Walter Jr.	WwJr@brbad.com	0425865498	2019
Mattei	Paul	p.mattei@mail.com	0622340956	2018
White	Walter	walter.W@breakingbad.com	0612345678	2013

```
connect: async function connexion_bdd()
{
  console.log("3) Connexion à la BDD".bold);

  con_bdd = mysql.createConnection(
  {
    host      : host,
    database  : name,
    user      : user,
    password  : pass,
    port      : port,
    connectTimeout: 5000,
    multipleStatements: true
  });

  var rep = await connect_bdd();
  return rep;
},
```

Figure 2 : Vue et connexion sur la base de données

```
app.get('/', function (req, res)
{
  incoming_request("GET", "/");
  res.send('Bienvenue sur le serveur Androscope');
})

app.post('/POST/DATA/', async function (req, res)
{
  incoming_request("POST", "/POST/DATA/", req.body); // 1) LOG
  var reply = await connexion_bdd.set(req.body); // 2) reponse json
  reponse_from_server(res, reply, index_request); // 3) envoi de la reponse
})

app.get('/GET/DATA/', async function (req, res)
{
  incoming_request("GET", "/GET/DATA/", "Données table MySQL"); // 1) LOG
  var reply = await connexion_bdd.get(); // 2) reponse json
  reponse_from_server(res, reply, index_request); // 3) envoi de la reponse
})

app.post('/DELETE/', async function (req, res)
{
  incoming_request("DELETE", "/DELETE/", req.body); // 1) LOG
  var reply = await connexion_bdd.delete(req.body); // 2) reponse json
  reponse_from_server(res, reply, index_request); // 3) envoi de la reponse
})

app.get('/GET/STATUS/', function (req, res)
{
  incoming_request("GET", "/STATUS/", "Status serveur");
  reponse_from_server(res, services_state, index_request);
})

app.post('/GET/ID/', async function (req, res)
{
  incoming_request("POST", "/POST/ID/", req.body); // LOG
  var reply = await connexion_bdd.checkId(req.body);
  reponse_from_server(res, reply, index_request);
})
```

Figure 3 : Traitement général des requêtes HTTP sur le serveur

5 Remerciements

Je remercie mon professeur Ivan Madjarov pour avoir proposé un tel projet mais aussi pour les connaissances qu'il m'a apporté en programmation.

6 Glossaire

CRUD, Create Read Update Delete, désigne les quatre opérations élémentaires dans la gestion des données.

HTTP, Hypertext Transfer Protocol, protocole de communication client-serveur.

Asynchrone, Une fonction asynchrone est une fonction qui s'exécute de façon décalée.

REST, Representational State Transfer, architecture de serveur où il est notamment question de communication à partir d'une URL suivant différentes méthodes.

NPM, Node Package Manager, plateforme de partage de module.

Express.js, module présent dans NPM qui fournit un ensemble de fonctionnalité pour application Web.

SDK, Software Development Kit, ensemble d'outils utilisé par les développeurs pour le développement d'un logiciel.

Framework, une architecture prête à l'emploi qui simplifie le travail de développeur.

RTDT, Round-Trip Delay Time, le temps que met un signal à parcourir l'ensemble d'un circuit.

Timeout, situation d'erreur où le programme cesse de fonctionner à cause d'une demande qui prend trop de temps.

Activité (Android), composante principale d'une application Android.

Input, champ de saisie pour l'utilisateur.

Layout, représentation visuelle de l'application Android.

ScrollView, objet qui permet de descendre ou de monter la partie visuelle d'une page.

DP, Density Pixel, unité de mesure relative à la densité de pixel de l'écran.

ViewGroup, désigne un ensemble de View.

View, chaque élément présent dans l'affichage hérite de la classe View et en est une instance.

Toast, la classe Toast d'Android Studio est utilisé pour afficher un message à l'écran.

Thread, un processus qui permet d'exécuter des instructions.

Monothread, en opposition au multithread, un système monothread s'exécute sur un seul thread.

Formatage, opération qui permet de configurer des éléments.

Bitmap, objet qui représente une image en bit sur Android Studio.

JSON, JavaScript Object Notation, format de données organisée.

Moteur V8, compilateur de code javascript écrit en C++ développé par Google.

Debug, permet d'identifier les erreurs liées à une exécution.

MultipleStatements, attribut dérivé du module MySQL de NPM qui permet d'imbriquer plusieurs requêtes MySQL à la fois.

7 Sitographie

GitHub, pour la documentation et l'utilisation des différents Framework suivant :

mysql, disponible sur

<https://github.com/mysql>

image-picker, disponible sur

<https://github.com/esafirm/android-image-picker>

picasso, disponible sur

<https://github.com/square/picasso>

color.js, disponible sur

<https://github.com/Marak/colors.js>

localtunnel, disponible sur

<https://github.com/localtunnel>

StackOverflow et **Quora**, pour une aide technique sur la programmation.

Utilisation d'une variable JSON depuis Javascript, disponible sur

<https://stackoverflow.com/questions/4314008/how-do-i-build-json-dynamically-in-javascript>,

<https://stackoverflow.com/questions/18977144/how-to-parse-json-array-not-json-object-in-android>

Exposer son serveur Node.js publiquement, disponible sur

<https://stackoverflow.com/questions/46049434/turn-local-nodejs-into-a-online-server>,

<https://stackoverflow.com/questions/14293370/publish-node-js-server-on-the-internet>

Communication client-serveur (Express.js & Android Studio), disponible sur

<https://www.quora.com/How-do-I-send-data-to-a-Node-js-server-from-an-Android-app>