Département de génie informatique et de génie logiciel

INF3995

Projet de conception d'un système informatique

Rapport final de projet

Chaîne de blocs pour dossiers d'étudiants

Équipe No. 01

Charles Marois
Francis Granger
Rose Hirigoyen
Bernard Meunier
Gabriel Pollo-Guilbert

Décembre 2019

Table des matières

1. Objectifs du projet

Le but du projet était de créer une plateforme utile au Registrariat contenant l'ensemble des notes des élèves selon les cours qu'ils ont suivis à Polytechnique. Cette plateforme sera partagée avec les employeurs potentiels pour étudiants et diplômés de Polytechnique Montréal afin qu'ils puissent avoir accès à ces informations. L'utilisation d'un registre distribué rend l'information inviolable et fiable pour tous.

Le système conçu est composé de plusieurs éléments. Premièrement, l'utilisation d'un registre distribué nécessite l'utilisation d'une chaîne de blocs pour garder en mémoire chaque modification, ainsi que trois mineurs utilisés pour produire et agrandir cette chaîne. Ces mineurs communiquent entre eux, mais aussi avec un serveur central. Celui-ci s'occupe de faire travailler les mineurs et sert d'interface entre la chaîne de blocs et les clients du système.

L'information contenue dans le système est accessible par une application Android de différentes façons. Deux types de comptes peuvent y avoir accès, soit des comptes d'édition et des comptes de consultation. Le type *édition* est conçu pour le personnel de Polytechnique, qui devra entrer les informations sur les dossiers que l'on retrouve sur la chaîne de blocs. Il peut aussi consulter l'information qui se trouve dans la chaîne. Les comptes de consultation sont eux utilisés par les employeurs potentiels et les étudiants, et ils ne peuvent que consulter l'information.

Un client administrateur sur PC est requis pour faire la gestion des comptes. Il peut créer et supprimer les deux types de comptes. De plus, le client PC a accès aux logs du serveur qui consistent en des messages d'erreur, de modification de la chaîne de blocs et autres.

Ce rapport fait donc état de notre présentation au client et du travail effectué au cours des derniers mois. Nous y décrirons le système développé et son fonctionnement, les résultats de nos tests, les défis et réussites rencontrés et terminerons par discuter des améliorations à apporter à notre projet et des apprentissages effectués.

2. Description du système

2.1 Le serveur (Q4.5)

Le serveur est l'interface principale entre les utilisateurs de l'application mobile et la chaîne de blocs. Cette interface a été implémentée comme mentionné dans l'appel d'offre selon une API REST suivant le cadriciel *Pistache* [1]. Le dossier « serveur/rest » comprend son code source et celui-ci est compilé selon le fichier *CMakeLists.txt* du même

dossier. Lors de l'initialisation du serveur[1], nous initialisons également la base de données personnelle du serveur[2], les gestionnaires pour l'adresse IP et pour la communication HTTPS par OpenSSL, le travailleur ZMQ qui communiquera avec les mineurs[3], le journal[4] et le routeur commun agissant comme intergiciel aux points finaux des requêtes[5]. Chacune de ces composantes seront décrites en détail pour mieux mettre en contexte le fonctionnement du serveur.

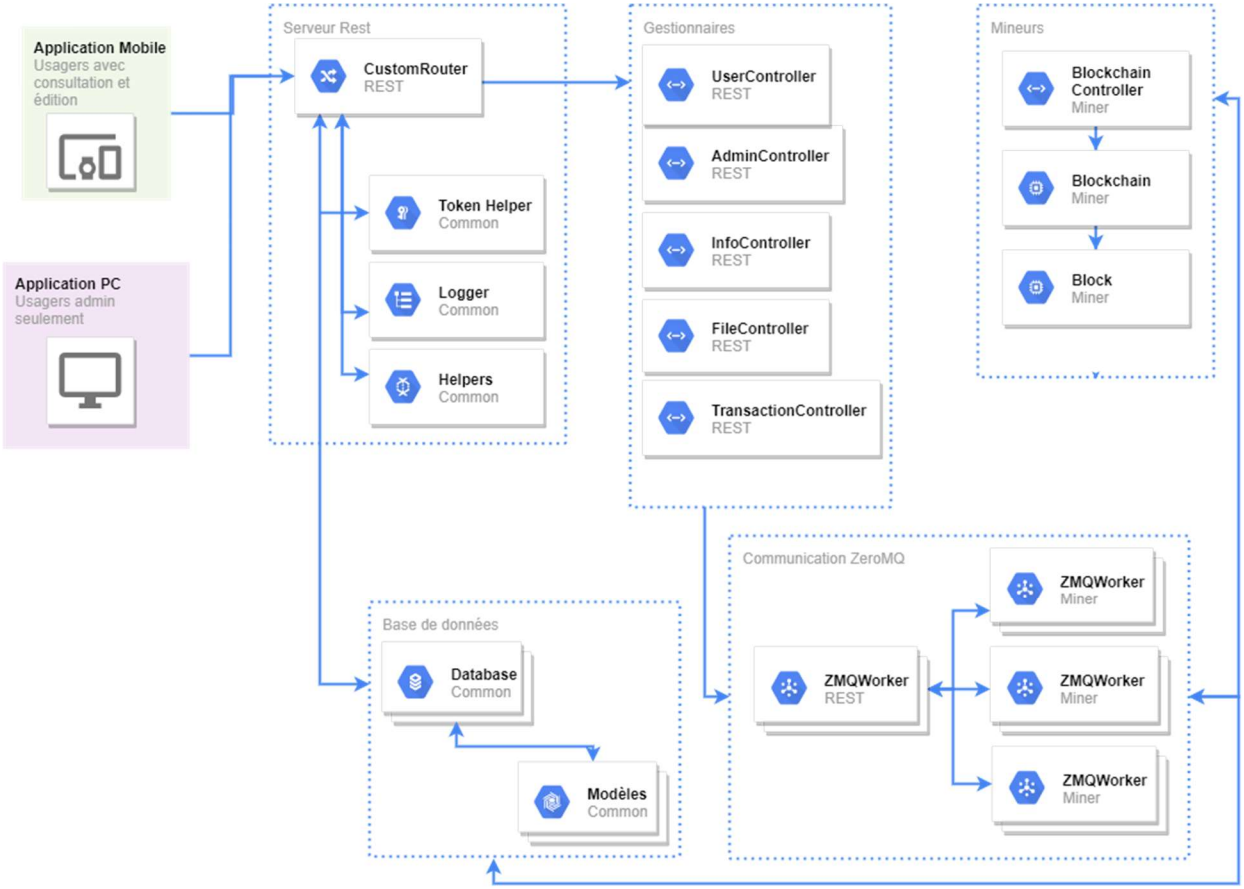


Figure 2.1.1 : Architecture du serveur

Nous avons opté pour une base de données SQLite [2] pour sa simplicité d'utilisation et de contrôle. La classe *Database* possède une méthode pour chaque requête à la base de données. Pour chaque requête à la base de données, nous faisons d'abord appel à la classe *Query* pour enregistrer celle-ci sous forme d'objet avec ses variables. Nous appelons ensuite la classe *Statement* qui contient la méthode *step()* qui retourne un booléen selon la validité de la requête. Selon cette réponse, nous pouvons

[1] Dont la localisation est : server/rest/rest.cpp
[2] Dont la localisation est : server/common/src/database.cpp
[3] Dont la localisation est : server/rest/src/zmq.cpp
[4] Dont la localisation est : server/common/src/logger.cpp
[5] Dont la localisation est : server/rest/src/custom_router.cpp

extraire l'information désirée avec *getColumnText()* et la renvoyer aux utilisateurs. Il faut noter que la base de données est statique et qu'elle suit le patron singleton. La méthode *init()* permet l'initialisation de l'instance unique et la méthode *get()* permet de retourner celle-ci. Enfin, l'attribut *functions_* est une mappe qui associe une méthode à son traitement par un des points finaux du routeur. La méthode *executeRequest()* utilise cette mappe pour répondre aux requêtes SQL déclenchés par le routeur. Ce traitement sera détaillé plus en détail lors de la description du routeur.

Autrement, plusieurs gestionnaires[6] nous permettent de compléter ou de bonifier les fonctionnalités du serveur. Le gestionnaire *firebaseHelper* permet de retrouver notre adresse IP sur un compte infonuagique personnalisé. Si elle est inexistante, il met automatiquement à jour la base de données dans le nuage. Ceci évite aux développeurs d'avoir à encoder manuellement l'adresse IP utilisée et leur permet de tester le serveur de façon concurrente. Le gestionnaire *scriptsHelper* automatise la création du certificat d'autorité en faisant appel au script « serveur/cert/createCert.sh » qui effectue une commande OpenSSL dans la console validant la connexion HTTPS vers le serveur. Ceci permet aux données de ne pas transiter en clair entre les applications et le serveur. Le gestionnaire *tokenHelper* permet de générer un jeton d'authentification selon le nom et le mot de passe cryptée d'un utilisateur, de le décoder et de vérifier son expiration. Ce gestionnaire est basé sur la librairie externe *cpp-jwt* [3], c'est-à-dire *JSON Web Tokens*. Les gestionnaires *formatHelper* et *messageHelper* sont utiles au formatage de nos données et à leur transformation d'un type à un autre, surtout lors de la sérialisation ou désérialisation de données JSON faite par la librairie externe *nlohmann* [4]. À noter aussi que nous avons utilisé le générateur de hash PicoSHA2 [5] pour crypter les données sensibles vers la base de données.

Pour ce qui du journal du système, nous avons décidé d'y aller avec un patron singleton ici également. Ce singleton[7] enregistre les événements de la sortie standard du serveur ou des mineurs dans leur base de données respective. Il présente trois types de méthodes correspondant aux événements demandés par le client : INFO, ATTENTION, ERREUR. Pour traiter ces événements qui sont enregistrés sous forme d'énumération, nous faisons appel à la librairie externe *magic_enum* [6].

Le routeur est la partie centrale du serveur. Celui-ci est mis en marche par le contrôleur principal, qui définit différents paramètres comme le nombre de fils d'exécution, la taille maximale des requêtes, le certificat d'autorité et la clé privé utilisée pour l'utilisation d'une connexion sécurisée. Le contrôleur principal a pour attributs tous les contrôleurs secondaires[8] et le routeur. Le routeur, dans notre cas, agit comme intergiciel et vérifie l'authentification des requêtes et l'existence des usagers dans la base

[6] Dont leur localisation est : server/common/include/common/
[7] Dont la localisation est : server/common/src/logger.cpp
[8] Dont leur localisation est : server/rest/src/

de données. Il appelle ensuite le contrôleur correspondant selon l'interface REST déterminée par le client. Chaque contrôleur agit comme point final et appelle une méthode spécifique selon l'adresse de la requête. Toutes les requêtes sont transformées en structure définie par nos modèles[9] et envoyées au travailleur ZMQ. Le contrôleur *transactionController* est en charge de décoder en base64 le fichier PDF reçu par la requête tout en envoyant le reste des informations au travailleur ZMQ pour enregistrer celle-ci dans la chaîne de bloc, alors que le contrôleur *fileController* est en charge d'encoder en base64 [7] le fichier PDF présent dans la base de données du serveur.

Enfin, le travailleur ZMQ utilise la librairie externe *ZeroMQ* [8] comme mentionné précédemment et implémente une relation publication-souscription. Essentiellement, le serveur instancie un singleton qui agit comme connecteur réseau (socket) et dont le rôle est de publier les requêtes SQL qu'il reçoit. Les trois mineurs, quant à eux, ont aussi des connecteurs réseaux qui s'assurent de souscrire au travailleur du serveur afin de recevoir toutes les requêtes SQL. Le travailleur peut déclencher des requêtes GET, qui sont utiles pour questionner les mineurs sur l'état de la chaîne de blocs, ou des requêtes UPDATE, qui indiquent aux mineurs qu'une nouvelle transaction est à ajouter à la chaîne de blocs. En plus de la relation publication-souscription, le travailleur reçoit des notifications des mineurs. Le travailleur répond à celles-ci par sa méthode *handlePullFromMiner()*. Deux cas de figures peuvent se présenter: soit un mineur vient de souscrire au travailleur du serveur, donc il souhaite s'identifier, soit il renvoie une réponse aux requêtes émises par le serveur et l'identifiant du dernier bloc de la chaîne. À noter qu'un fil d'exécution est utilisé pour chaque relation et que les structures utilisées dans cette classe sont présentes dans le fichier « miner_models.hpp»[10].

Ainsi, la majorité de notre système correspond à ce à quoi nous nous étions engagés lors de l'appel d'offres. Le client avait mentionné que notre solution était peu détaillée à ce moment, alors nous espérons que la description ci-haut et qui suivra sera suffisamment exhaustive pour la bonne compréhension des fonctionnalités du système et les désirs du client.

2.2 Les mineurs

Le rôle principal du programme de mineur est de gérer la chaîne de blocs. Celle-ci est définie à partir de deux classes: *Block* et *BlockChain*. La première définit l'information et les opérations de base sur un bloc comme leurs chemins sur le disque, son haché ou encore la méthode pour le miner. La seconde classe vient encapsuler les

[9] Dont leur localisation est : server/common/include/common/models.hpp
[10] Dont la localisation est : server/common/include/common/miner_models.hpp

blocs individuels pour créer une chaîne. Elle offre les informations sur la chaîne au complet ainsi que les méthodes pour ajouter des nouveaux blocs.

Par-dessus la classe *BlockChain*, il y a la classe *BlockChainController*. Celle-ci établit un lien entre les fils d'exécution ZMQ et la chaîne de blocs. Ce contrôleur gère aussi différentes logiques du système distribué des mineurs. Par exemple, il est responsable du minage et d'arrêter le minage dans le cas où un autre mineur a trouvé un nonce valide. Il est aussi responsable de synchroniser les mineurs entre eux lorsqu'un mineur redevient en service.

Finalement, la classe ZMQWorker gère toutes les différentes communications ZMQ. Comme mentionné plus haut, nous utilisons plusieurs sockets ZMQ afin de permettre au serveur de discuter avec les mineurs et aux mineurs de discuter entre eux.

Les principales librairies utilisées dans le mineur sont bien sûr *libzmq* et son interface C++ *cppzmq*. Les hachés SHA256 sont calculés à l'aide d'une librairie *picosha2*, une librairie avec seulement des fichiers en-têtes facile à inclure dans un projet. Nous utilisons SHA256, parce que cette méthode de hachage est prouvée dans le Bitcoin et il est peu probable qu'il soit nécessaire d'utiliser une autre méthode. Le choix de l'algorithme de hachage ayant peu d'impact sur le fonctionnement du système final, il a été décidé de garder son fonctionnement simple.

Afin de ne pas lire la chaîne de bloc à chaque requête, on met en cache les informations les plus récentes de la chaîne de blocs dans une base de données SQLite. En fait, il est facile de reconstruire la base de données à l'aide de la chaîne de blocs, car l'information enregistrée dans les blocs est l'entièreté des commandes SQL modifiant l'état du système. Une commande n'effectuant qu'un accès sans modifier la base de données n'ajoute pas de blocs.

2.3 Tablette (Q4.6)

L'application mobile se veut simple et intuitive, et nous avons choisi le rouge de Polytechnique comme couleur principale. Nous avons essayé le plus possible de suivre les directives du *material design* (Google) en ce qui concerne les couleurs et le style général. Pour l'interaction, nous nous sommes inspirés de l'application mobile *To Do* de Microsoft. En effet, nous avons repris l'idée du panneau latéral et de l'ajout en chaîne d'éléments dans une liste. Nous avons également choisi de prioriser une application robuste et des fonctionnalités qui répondent bien plutôt qu'une interface complexe et instable. Nous voulions nous assurer que tout ce qui était développé était utile et bien testé.

L'application est écrite en Kotlin et n'utilise que deux activités principales afin de minimiser les changements de contexte. Nous priorisons plutôt l'utilisation de fragments,

et nous alternons entre ces derniers pour présenter l'information. Les fragments nous permettent d'échanger facilement de l'information entre les scènes.

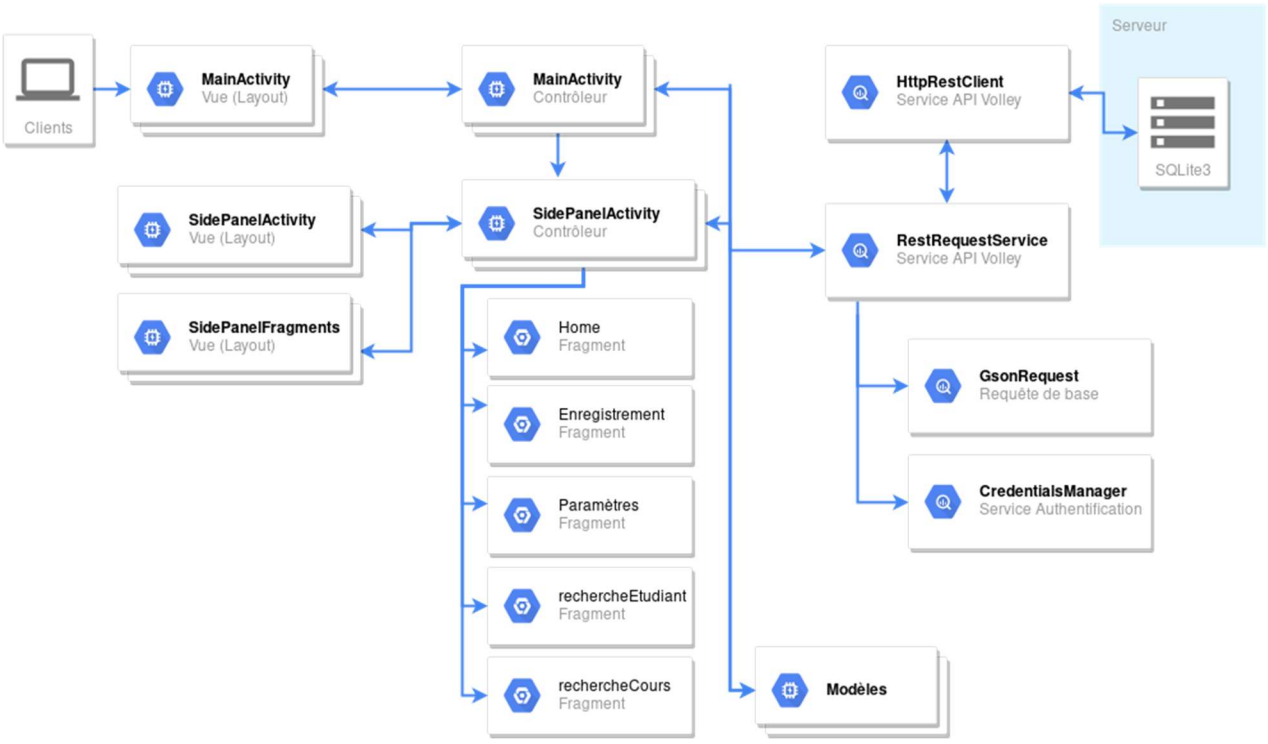


Figure 2.1.3 : Architecture de l'application Android

Notre application est donc séparée en 3 principales composantes, soit les activités, les fragments et les services.

Tout d'abord, nous avons l'activité de connexion, qui authentifie les utilisateurs, et l'activité du panneau latéral, qui est la pièce maîtresse de notre application, puisque c'est elle qui gère tous les fragments.

Ensuite, nous avons l'activité du panneau latréal, qui gère tous les fragments relatifs aux fonctionnalités de notre application. Nous avons en effet cinq fragments principaux, auxquels se rattachent de plus petits fragments ou des adapteurs pour les vues contenant des listes. D'ailleurs, nous avons utilisé des *RecyclerViews* pour faire les listes, qui sont en fait des modèles avancés des listes normales et qui permettent de seulement gérer ce qui est affiché à l'écran, ce qui augmente la performance de l'application.

Nos cinq fragments principaux sont donc le menu principal, les fragments pour voir la liste des classes et des élèves, auxquels se rattachent des fragments plus simples pour obtenir des informations sur un élément en particulier, le fragment utilisé pour enregistrer les classes d'élèves, qui est le plus complexe de tous puisqu'il regroupe un plus grand nombre d'interactions, et le fragment pour changer le mot de passe du compte.

Finalement, les services sont composés de tous les outils supplémentaires dont nous avons eu besoin au cours du développement. Par exemple, nous avons un *CredentialsManager*, qui nous permet d'obtenir des jetons d'authentification, ainsi qu'un *RestRequestService*, qui contient la forme de toutes les requêtes exécutées par l'application. Nous avons également défini les modèles, qui représentent les données échangées par l'application, ainsi que des outils qui nous permettent par exemple de mieux gérer toutes les erreurs lancées par l'interaction avec le serveur.

En ce qui concerne les librairies, nous utilisons Volley afin de gérer les requêtes Http. Nous avons également utilisé la librairie *AndroidPdfViewer* de barteksc afin de gérer l'affichage des PDFs dans l'application. Finalement, nous avons utilisé koin, une librairie pour nous aider à gérer nos dépendances.

2.4 Application sur PC

L'application sur PC se veut simple et rapide d'utilisation comme toute bonne application administrative d'un système. L'accent a été mis principalement sur la robustesse de l'application, la facilité à accéder et visualiser l'information et les fonctionnalités plutôt que sur son apparence.

Nous avons utilisé une architecture JavaFX pour construire notre application. Cette architecture utilise Java comme langage et contient tous les éléments nécessaires pour faire une application simple comme la nôtre. Le modèle utilisé pour la structure est le modèle MVC et nous avons utilisé des fichiers FXML pour faire les différentes vues. L'application est composée de plusieurs éléments comme on peut le voir sur le schéma ci-dessous.

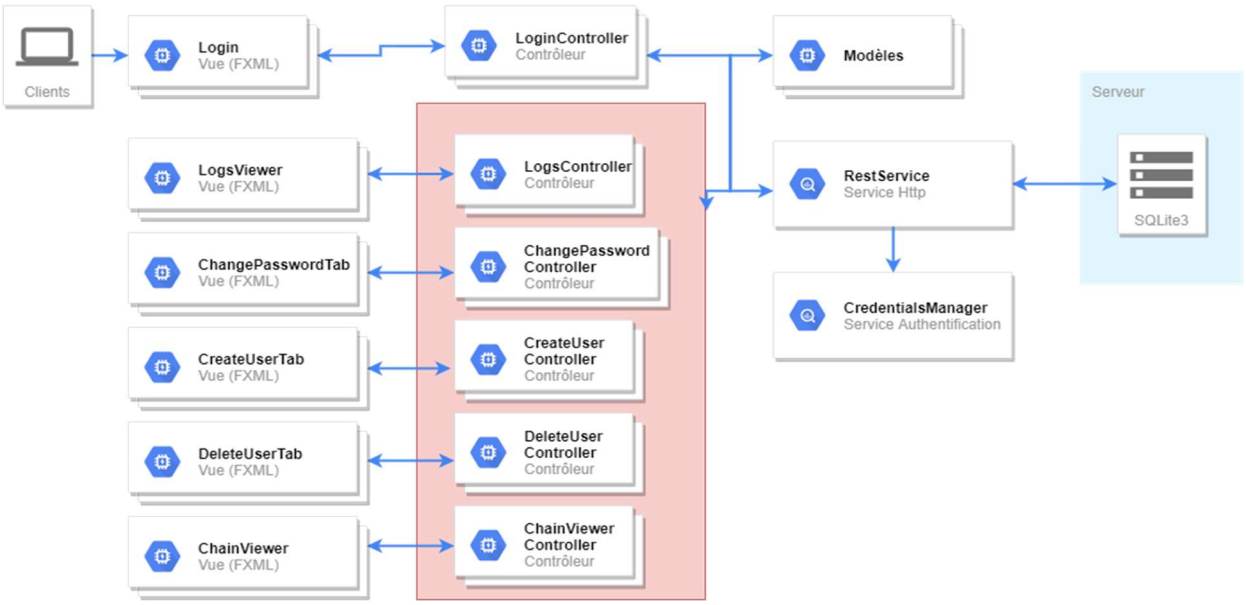


Figure 2.1.4 : Architecture de l'application PC

L'utilisateur voit en premier la vue de *Login*. Celle-ci est liée au contrôleur nommé *LoginController*. Lorsque l'utilisateur tente de se connecter, une requête est faite au serveur à l'aide d'un service HTTPS. Une fois connecté, un panneau d'affichage est créé et on y ajoute quatre onglets contenants chacun une différente vue. Premièrement, on y retrouve la vue de visualisation des logs nommée *LogsViewer* et qui est contrôlée par *LogsController.* Cette vue contient principalement un tableau affichant les logs du système. Ensuite on retrouve un onglet contenant les champs nécessaires pour créer un utilisateur. Sa vue s'appelle *CreateUserTab* et est contrôlée par *CreateUserController*. Troisièmement, on y retrouve un onglet pour supprimer des utilisateurs. Cette vue qui s'appelle *DeleteUserTab* et son contrôleur *DeleteUserController* contenant une table où chacune de ses rangées contient un nom d'utilisateur, les permissions reliées à cet utilisateur ainsi qu'un bouton supprimer pour le supprimer. Le quatrième onglet est pour changer le mot de passe de l'utilisateur connecté et qui affiche la vue *ChangePasswordTab* contrôlée par *ChangePasswordController.* Finalement, nous avons un onglet pour afficher la chaine de blocs ainsi que et donne chaque bloc une couleur selon leur statut de vérification. Cette vue s'appelle ChainViewer et est contrôlée par ChainViewerController. *Pour toutes ces vues, tout*es les communications au serveur se font avec le service HTTPS de l'application et avec l'aide de plusieurs modèles différents.

Toute communication en direction du serveur se fait de manière cryptée à l'aide de la librairie OpenSSL permettant un cryptage SSL. De plus, un jeton d'authentification est sauvegardé du côté client et est joint aux différentes requêtes vers le serveur pour que ce dernier puisse vérifier l'authenticité du client connecté.

2.5 Fonctionnement général (Q5.4)

Toutes les parties subséquentes nécessitent que vous ayez notre dépôt GIT installé localement avec la commande suivante :

```
git clone https://githost.gi.polymtl.ca/git/inf3995-01
cd inf3995-01
```

Pour partir le serveur REST:

Nous sommes dépendants de la librairie pistache pour notre interface REST, celle-ci semble être seulement disponible sous linux. Donc nous recommandons un système comme Ubuntu ou WSL (nous tenons à mentionner que la moitié de notre équipe à réussi à développer sous Windows avec ceci sans problème!).

- Dépendances principales :
  1. Cmake 3.15+ pour créer les fichiers de compilation
  2. Ninja 1+ pour accélérer la compilation
  3. Clang 8+ pour compiler le code (fonctionne avec modification pour GCC 9+)
- Il faut rouler les commandes suivantes pour compiler le code et ses dépendances. Si vous avez des utilitaires manquants, il faudra les installer via le code source ou via apt (sur ubuntu).

```
1. cd server
2. ./build.sh # ceci exécute cmake et ninja
```

- Il est aussi possible de compiler le code avec VS Code :

- Il faut lancer les commandes suivantes pour lancer le serveur :

```
1. cd build/rest
2. ./rest
```

- Il vous sera demandé d'entrer un mot de passe pour sécuriser votre adresse IP avec notre certificat.

```
1. Enter pass phrase for rootCA.key: equipe01
```

- Vous devriez voir à l'écran :

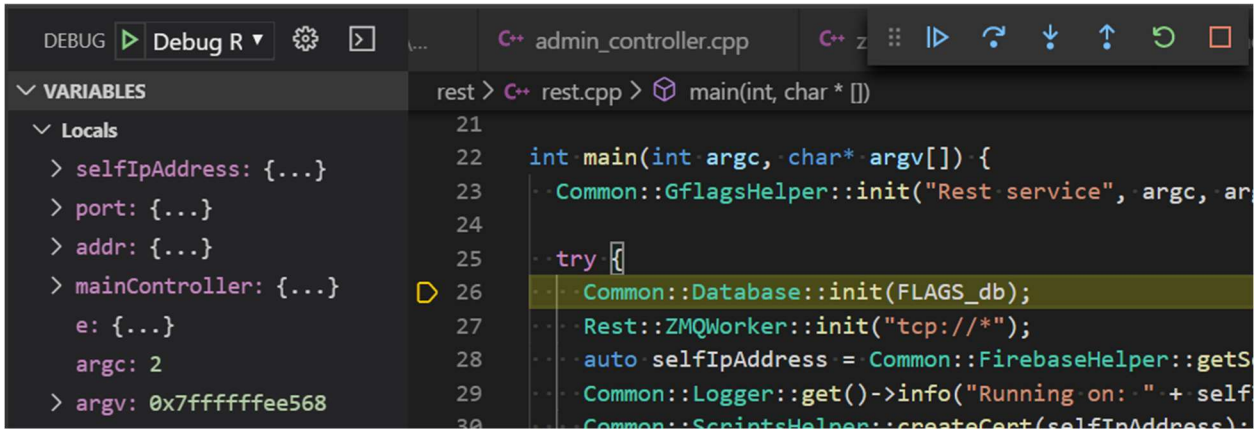
```
Enter pass phrase for rootCA.key:
Ip succesfully added

3: INFO: Mon Dec  2 23:44:46 2019: 0: ZMQ/miners: thread started


4: INFO: Mon Dec  2 23:44:46 2019: 0: ZMQ/proxy: thread started

Succesfuly able to update firebase database for user server
```

- Il est aussi possible de lancer le programme en déverminage avec la commande F5 dans VS code :



- Le serveur comporte plusieurs arguments que vous pouvez passer via la ligne de commande :

```
DEFINE_string(user, "server", "Developper using the service");
DEFINE_string(cert, "server.crt", "Path to server cert");
DEFINE_string(key, "server.key", "Path to server key");
DEFINE_string(db, "rest.db", "Path to sqlite db file");
DEFINE_int32(port, 8080, "REST http port");
DEFINE_int32(threads, 4, "Number of threads");
DEFINE_string(transactions, "transactions/", "Path to transactions");
DEFINE_int32(buffer_size, 4000000, "Maximum number of bytes in buffer");
DEFINE_int32(timeout, 100, "Maximum number of seconds before timeout");
```

Pour partir les mineurs:

Les mineurs sont dans le même projet Cmake que le serveur, alors les dépendances sont presque identiques.

- Il faut lancer les commandes suivantes pour lancer un mineur:

```
1. cd build/server
2. ./miner
```

- Si vous voulez lancer un deuxième mineur simplement le lancer dans un autre terminal avec les arguments suivants:

```
1. ./miner --db blockchain2.db --blockchain blockchain2
```

- Vous devriez voir à l'écran :

```
7: INFO: Tue Dec  3 00:54:57 2019: 1: This is miner #1, connected to server
```

- Les mineurs comportent plusieurs arguments que vous pouvez passer via la ligne de commande:

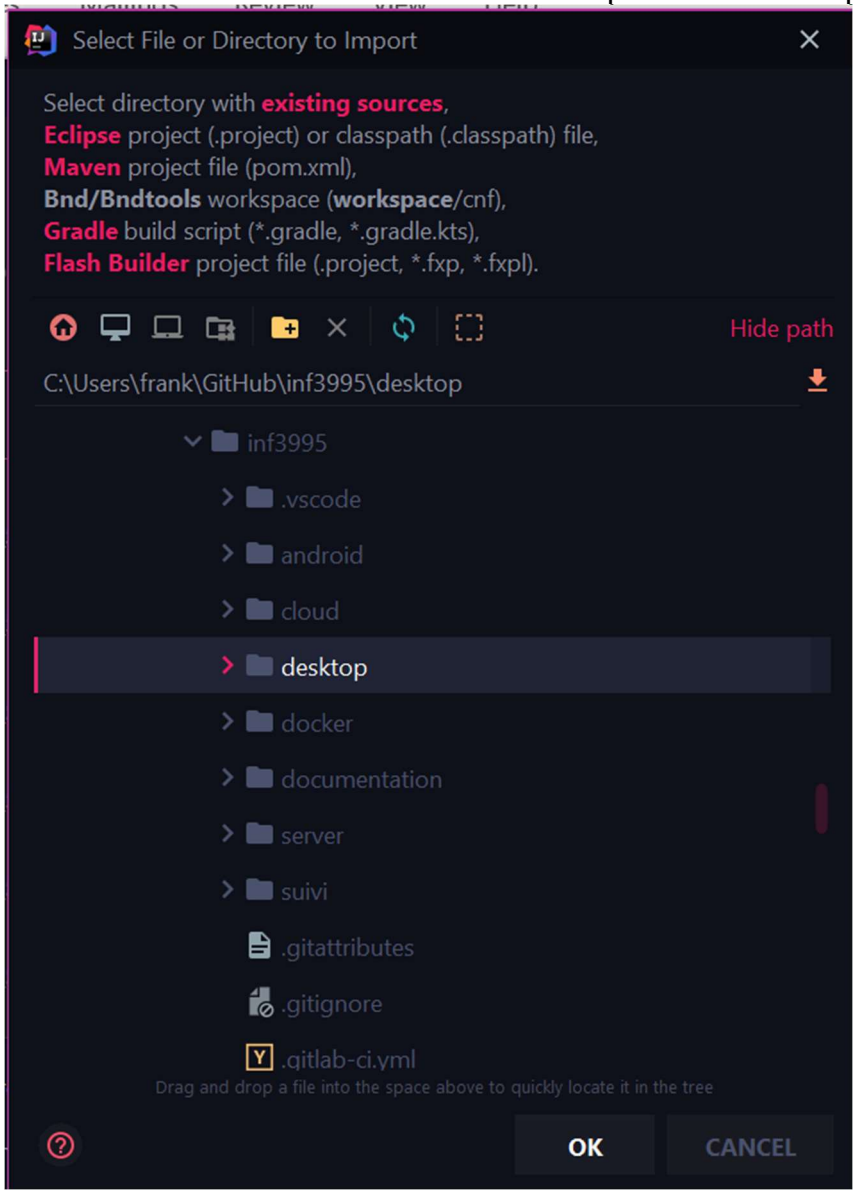
```
DEFINE_string(addr, "", "REST service address");
DEFINE_string(user, "server", "Developper using the service");
DEFINE_string(db, "blockchain.db", "Path to sqlite db file");
DEFINE_string(blockchain, "blockchain/", "Path to blockchain folder");
DEFINE_int32(difficulty, 3, "Hashing difficulty");
```

Pour partir l'application PC:

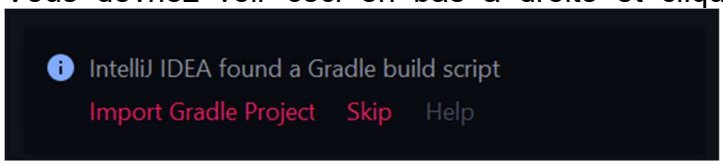
Nous vous recommandons d'utiliser le logiciel Intellij pour compiler et lancer l'application JavaFX.

- À l'ouverture d'Intellij, cliquez sur Import Projet :

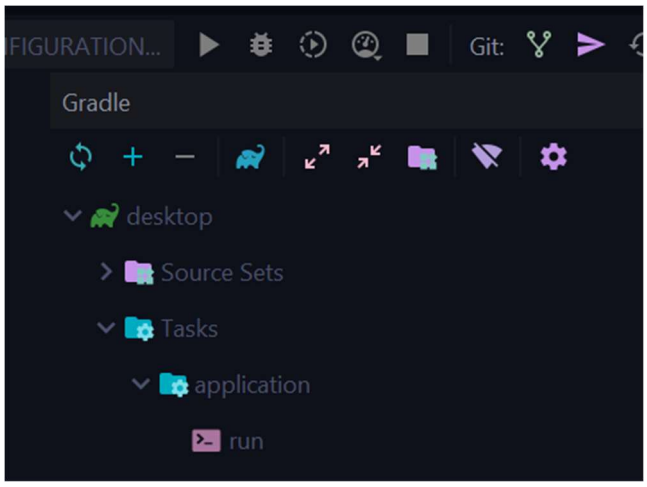
- Sélectionnez le chemin vers le répertoire desktop du dépôt GIT :



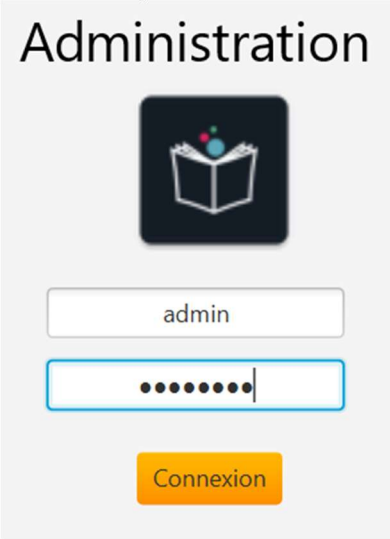
- Acceptez toutes les valeurs par défauts et laissez Intellij initialiser le projet
- Vous devriez voir ceci en bas à droite et cliquer sur Import Gradle Project



- Une fenêtre devrait s'ouvrir sur le côté ou vous pouvez double cliquer sur run :



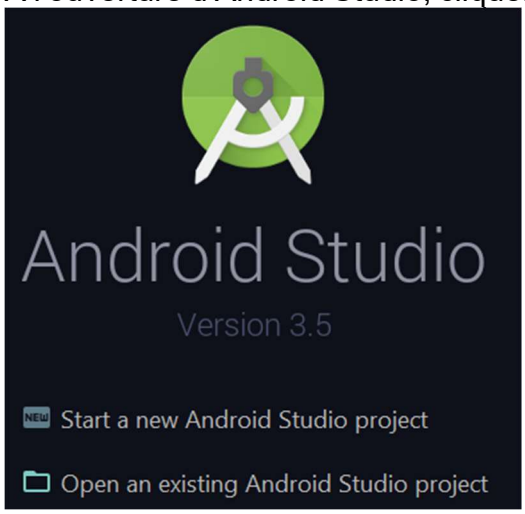
- Vous pouvez vous connecter avec admin et equipe01



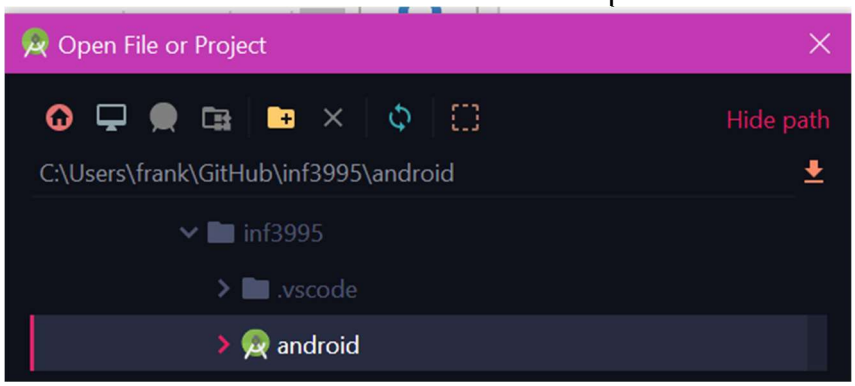
Pour partir l'application Android:

Nous vous recommandons d'utiliser Android Studio.

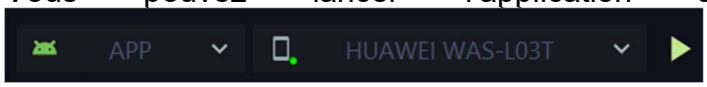
- À l'ouverture d'Android Studio, cliquer sur Open an existing Android Studio project:



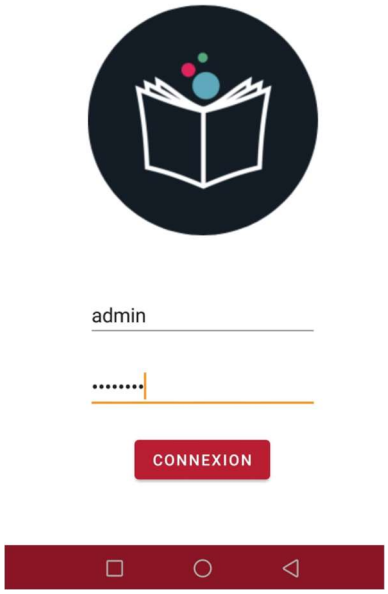
- Sélectionner le chemin vers le répertoire android du dépôt GIT :



- Acceptez toutes les valeurs par défauts et laissez Android Studio initialiser le projet
- Vous pouvez lancer l'application avec le bouton vert :



- Vous pouvez vous connecter avec admin et equipe01 :



3. Résultats des tests de fonctionnement du système complet (Q2.4)

Afin de tester l'ensemble du système, une série d'étapes étaient nécessaires. Une version ARMv7 du serveur et du mineur devaient être compilés afin d'être mises sur les cartes. Chaque programme s'exécutant sur les cartes devait ensuite être démarré. Dans certains cas, il était nécessaire de réinitialiser les données à zéro afin de démarrer les tests d'intégrations sur un système vierge.

La partie la plus longue à effectuer est la compilation des logiciels. Notre système de construction compile et installe toutes les dépendances du système afin de créer deux binaires statiques finaux. Cela fait en sorte qu'il est très facile de construire l'application sans devoir installer des librairies externes. Par contre, le temps de compilation est augmenté considérablement. Ce délai était encore plus considérable lorsqu'il était temps de compiler sur les cartes où le processeur ARM n'était pas très puissant.

La solution pour augmenter la rapidité de ce processus fut de *cross-compile* les binaires ARMv7 à partir d'une machine x86 considérablement plus puissante. Grâce à ce processus, on a pu accélérer le temps de compilation d'environ 45 minutes à moins d'une minute. Pour accélérer encore plus le processus de tests, nous avons aussi configuré un répertoire de paquets ArchLinux avec un paquet contenant les deux binaires et les fichiers de configuration. Grâce à cela, il était trivial de déployer les nouvelles versions des logiciels sur les cartes. À la fin, on était en mesure de redéployer une nouvelle version

sur les cartes en environ 1 minute, ce qui permettait de tester et d'apporter des modifications très rapidement au système.

En général, toutes les fonctionnalités qui étaient demandées par le client sont fonctionnelles. Cela dit, l'expérience utilisateur pourrait être amélioré. Par exemple, certaines requêtes provenant de l'application mobile peuvent avoir une latence considérable, ce qui rend son utilisation moins fluide et agréable à l'utilisateur. Ceci est dû au fait que la réponse à un requête est seulement envoyée lorsqu'au moins un mineur a fini de miner le bloc.

Ce problème pourrait être résolu en répondant immédiatement à la requête avec le résultat et démarrer le minage en arrière-plan. Cette solution n'a pas été envisagée lors des premiers concepts du système en raison d'une mauvaise compréhension des requis. Plus précisément, la chaîne de blocs dans le système est seulement utilisée comme manière d'archiver de l'information et son historique tout en empêchant la modification non-autorisée. L'erreur dans le prototype fut de considérer l'aspect décentralisé de la chaîne de blocs, ce qui n'était pas dans les requis et a grandement complexifié le système.

Un autre problème important, lui aussi due à l'architecture décentralisée, est qu'il y a une grande quantité de situations de compétition à travers le système. Par exemple, si deux mineurs minent un bloc en même temps et arrivent à deux nonces différents, le comportement n'est pas déterminé. Il est possible que tout fonctionne bien, comme il est possible que les chaînes de blocs divergent. Une solution en conservant l'aspect décentralisé aurait été d'ajouter des algorithmes de consensus, comme l'algorithme de Paxos ou de Raft. Les réseaux décentralisés comme le Bitcoin utilisent de tels algorithmes.

La meilleure solution dans le cadre de ce projet aurait été de transitionner vers un système centralisé autour du serveur. La majorité des situations de compétitions pourraient ensuite être évitée en utilisant un verrou partagé sur l'ensemble du système.

4. Déroulement du projet (Q2.5)

Ce qui a bien été :

L'aspect dont l'équipe est le plus fière est certainement la grande qualité de code du produit. Tant au niveau de notre développement que le rendu du projet. Nous avons respecté tous les plus hauts standards de l'industrie dans le C++, le Kotlin et le Java. En plus de ceci, nous avions une plateforme bien établie de développement continu et de déploiement continu. Ceci Cus a pris beaucoup de temps en début de session, mais nous a certainement sauvé d'innombrables heures en fin de parcours. Dans un autre ordre d'idées, nous avons eu une bonne chimie d'équipe et une excellente communication tout au long du projet. Tout le monde a mis la main à la pâte et tout le monde a pu se rendre

utile. Il est important d'avoir une bonne communication pour changer rapidement de contexte, compléter les tâches de certaines personnes ou simplement pour faire de la programmation en pair. Somme toute, toutes les composantes mentionnées dans l'appel d'offre sont dans le produit final, le projet s'est très bien passé et nous en sommes fiers.

Ce qui doit être amélioré :

D'abord, notre gestion du temps laissait à désirer. Bien que nous ayons été capable de livrer la marchandise à temps pour les deux remises, finir à sept heures du matin n'est jamais une excellente idée. Le pire dans tout cela, c'est que nous n'avons pas appris de nos erreurs entre la première et la deuxième remise, nous nous sommes un peu assis sur nos lauriers, et finalement, c'est arrivé de nouveau. Ensuite, nous aurions dû utiliser une autre librairie que ZMQ pour la communication entre mineurs et serveur. Celle-ci a été assez fastidieuse et redondante.

5. Travaux futurs et recommandations (Q3.5)

Bien que notre système remplisse la grande majorité des fonctionnalités demandées par le client, nous avons noté quelques lacunes lors de la démonstration finale. Ces lacunes devront être corrigées pour respecter l'entente conclue.

Tout d'abord, nous avons remarqué que lorsqu'un usager est supprimé de la base de données par un administrateur, son jeton d'authentification n'est pas invalidé automatiquement. L'usager peut donc faire des requêtes non autorisées tant et aussi longtemps qu'il ne se déconnecte pas de l'application ou lorsque son jeton expirera après l'heure d'émission. Ceci est une lacune relativement simple à corriger, puisqu'il suffit de corriger la réponse de la requête de suppression de compte effectuée par un administrateur. Ensuite, nous avons opté pour un chiffrement du mot de passe du côté serveur plutôt que du côté client. Ceci a pour effet qu'un individu mal intentionné peut accéder aux identifiants d'un utilisateur s'il intercepte sa requête de connexion et est en soi un problème de sécurité de base. Il suffit d'utiliser le même algorithme de chiffrement, dans notre cas SHA256, mais du côté client. Il existe très probablement une librairie externe pouvant faire ce travail et le tour serait joué. De toute façon, nous encryptions déjà le mot de passe avant de le sauvegarder dans notre base de données et nous créions un salage spécifique pour chaque utilisateur. Cette nouvelle implémentation viendrait bonifier la sécurité de notre système. Également, nous souhaiterions aussi faire une dernière analyse statique du code source, communément appelée *lint,* pour garantir une qualité logicielle adéquate et rigoureuse au client.

En ce qui concerne l'application Android, nous avons noté deux recommandations du client. Celui-ci aimerait premièrement que nous validions la bonne forme syntaxique du matricule entré par le registrariat. Nous validons déjà l'information relative aux noms de cours et des notes attribuées aux étudiants, alors cela ne devrait pas demander beaucoup de ressources supplémentaires. Deuxièmement, le client souhait aussi que

nous améliorions la pile de retour de nos fragments. Ainsi, au lieu de retourner toujours à la page d'accueil lorsqu'on complète l'ajout d'un cours ou lorsqu'un usager sélectionne le bouton « retour arrière », le client voudrait que l'usager retourne tout simplement à la page précédente. Cela demanderait un peu de travail considérant que nous utilisons en ce moment deux piles de retour distinctes selon le fragment en cours d'utilisation, mais nous avons préalablement réfléchi au problème et avons une solution potentielle à proposer au client si nécessaire.

Pour les programmes sur les cartes, il serait important de centraliser les opérations des mineurs autour du serveur. L'aspect décentralisé de ceux-ci augmentent considérablement la complexité du système et ajoute une panoplie de situations de compétition pouvant rendre le système dans des états non-déterminés. Aussi, l'utilisation de ZMQ comme moyen de communication entre le serveur et les mineurs devrait être remplacer par quelque chose de plus flexible et facile d'utilisation. Une alternative aurait été d'utiliser gRPC, un cadriciel de communication développé et utilisé par Google dans leurs systèmes infonuagiques. L'avantage principale de ce cadriciel est qu'il est facile de voir les différents types messages à travers le réseau et il génère tout le code réseau pour différents langages facilement. Bref, gRPC est beaucoup plus flexible que ZMQ et est prouvé dans les systèmes distribués à très large échelle.

Enfin, l'application de bureau est complète et ne nécessite pas de travail supplémentaire. En effet, le client était satisfait des fonctionnalités implémentées et celles-ci ont toutes passé les tests effectués.

6. Apprentissage continu (Q12)

[ROSE HIRIGOYEN] Selon moi, ma principale lacune au niveau technique a été mon manque d'expérience en développement d'applications Android. En effet, bien que je sois capable de me débrouiller, le fait que je n'aie pas vraiment d'idée globale de l'interaction entre les fragments a créé quelques problèmes au niveau de l'application finale, par exemple avec la pile de fragments qu'on utilise pour revenir en arrière. Pour y remédier, j'ai fait quelques maquettes de l'application afin de savoir un peu plus dans quelle direction aller lors du développement, mais cela aurait pu être amélioré par plus de recherches préalables sur le sujet, et en regardant plus d'exemples d'applications déjà conçues. Au niveau personnel, ma principale lacune a été mon manque d'organisation. En effet, je n'ai pas su bien prioriser le projet par-dessus mes responsabilités en tant que co-directrice du STEP et parmi tous mes autres travaux scolaires. J'ai réussi à rattraper une bonne partie du retard lors du deuxième livrable, et je me suis assurée de bien m'organiser avec mes coéquipiers pour le troisième. J'ai cependant trouvé quelques nouvelles méthodes d'organisation, comme maintenir une liste de choses à faire et prioriser les périodes en équipe pour bien me concentrer sur le projet, et je pourrai les appliquer lors du prochain projet.

[CHARLES MAROIS] Ma principale lacune à mon sens était mon manque d'expérience en programmation C++, et donc, je crois que je ne maîtrisais pas suffisamment les bonnes pratiques de ce langage. Pour remédier à cette situation, nous nous assurions de contrevérifier mon travail par quelqu'un avec davantage d'expérience en C++. Je me suis aussi assuré de lire davantage de documentation sur les bonnes pratiques du C++ et l'utilisation de classes dites gestionnaires. Ayant moins d'expérience en création d'interfaces REST, j'ai aussi dû mettre à jour sur les bonnes pratiques d'implémentation comme l'ajout d'un intergiciel (middleware) permettant de filtrer les requêtes et d'être plus vigilant quant aux états des objets. Effectivement, j'avais moins d'expérience avec l'utilisation de mutex pour protéger certaines variables et états, j'ai donc appris énormément à ce sujet lors du développement de notre serveur et des mineurs. J'ai eu beaucoup de soutien de mes collègues ce que j'ai bien apprécié. Je crois que nous avions une bonne communication et ça nous a permis d'atteindre nos buts et un projet fonctionnel. En ce qui attrait aux améliorations que j'aurais pu apporter, je crois que je dois prendre davantage de temps pour réfléchir au design de mes systèmes, de l'état de mes objets et du flot de l'information. J'ai tendance à coder directement selon mes intuitions, mais cela est parfois contreproductif, car je me retrouvais à redévelopper certains sections plusieurs fois, car j'ai mal considéré certains aspects essentiels des requis. Bref, ce projet a été un terrain d'apprentissage fort utile.

[BERNARD MEUNIER] Mon plus grand défi durant la réalisation du projet était mon absence d'expérience dans la création d'application *front-end*. Je n'avais jamais utilisé d'architecture MVC auparavant et aussi très peu travaillé avec des interfaces utilisateur. J'ai vu le projet comme une opportunité pour pouvoir enfin me pratiquer à en faire et j'ai donc choisi de travailler beaucoup sur le développement du client su PC. Pour pouvoir régler ce problème, j'ai commencé par faire beaucoup de recherches pour pouvoir implémenter un modèle MVC à l'application selon les bonnes pratiques de travail. Ensuite, j'ignorais comment m'y prendre pour structurer et compartimenter les différentes vues à l'intérieur de l'application. J'ai donc eu besoin d'encore une fois faire beaucoup de recherche sur internet à propos des meilleures pratiques, mais cette fois-ci j'ai aussi demandé de l'aide de mes coéquipiers pour avoir leur opinion puisque la plupart avaient de l'expérience sur le sujet. Finalement, je voulais faire une application qui avait une belle apparence et un design intéressant. C'était l'élément le plus difficile, puisque cette fois-ci c'est beaucoup plus compliqué rechercher comment faire en ligne. Pour en venir à bout, j'ai dû faire énormément d'essais-erreur avec plusieurs styles pour voir comment était le résultat et s'il me satisfaisait. J'ai aussi consulté mes collègues pour avoir leur opinion et conseil sur le sujet. Je suis content d'avoir mis les efforts pour apprendre à faire le développement d'une application client. Je me sens plus confiant à en refaire dans le futur. Bien sûr, l'application PC de notre projet reste loin d'être parfaite, il y a beaucoup de changement que je ferais, surtout sur les éléments que j'ai faits au début de la session maintenant que j'ai pu accumuler plus d'expérience de développement. Je voudrais

mieux partitionner mes tâches dans plus de vues, par exemple, c'est le contrôleur de Login qui gère la création de fenêtres du reste de l'application et cela ne semble pas être à sa place. Côté apparence, je voudrais moins utiliser le design de JavaFX et sa palette de couleur, mais prendre le temps de modifier l'apparence des éléments et faire quelque chose d'unique avec son propre style.

[FRANCIS GRANGER] : L'aspect sur lequel je devais le plus travailler pendant le projet était ma gestion d'équipe. Bien que nous nous sommes fait vanter les mérites d'un réseau décentralisé dans nos cours de HPR, il est primordial d'avoir un membre de l'équipe qui reste au courant du développement et de l'avancement de chaque membre. Ceci n'est pas vraiment un rôle que l'équipe m'a donné, c'est quelque chose qui s'est plus concrétisé plus le projet avançait. J'ai donc essayé le plus souvent de prendre part et d'aider mes collègues le plus possible, tant d'un point de vue technique que d'un point de vue de description des tâches. J'aurais pu m'améliorer sur ce sujet en étant moins direct et peut être plus constructif dans mes commentaires. Nous étions dans un environnement assez dynamique où nous n'avions pas tellement le temps de discuter pour trouver la meilleure solution. Donc clairement pousser les gens encore plus tôt à travailler serait quelque chose à faire!

[GABRIEL GUILBERT] : Ma principale lacune dans ce projet fut la maitrise du C++. Ayant beaucoup d'expérience dans le C, j'ai sous-estimé la différence entre les deux langages. Il s'avère qu'écrire du C++ de bonne qualité est beaucoup plus long que prévu. Même si ce langage est vu à priori dans un autre cours, il est beaucoup plus complexe. J'ai dû lire beaucoup plus sur le sujet et l'analyseur statique utilisé dans notre projet m'a beaucoup aidé à définir les mauvaises pratiques et les comprendre. En termes de savoir-faire, mon principal problème est que je suis une personne totalement désorganisée dans l'ordonnancement de mes tâches. J'ai essayé dès le début du projet de mieux allouer du temps libre dans mon calendrier afin de pouvoir travailler sur le projet. Cette technique a fonctionné pendant quelques semaines. Le moment où j'avais peu de motivation, je préférais procrastiner. Ce manque d'organisation explique en partie pourquoi il a été nécessaire de faire des nuits blanches avant les remises. Le manque de motivation apparait souvent quand je tombe sur un embûche qui demande plus de travail que je ne l'aurais pensé. Je crois qu'une bonne solution pour empêcher cela serait de demander plus d'aide de la part de mes coéquipiers, car j'ai rarement l'habitude de le faire.

7. Conclusion (Q3.6)

Au vu du travail réalisé, nous sommes plutôt satisfaits de ce qui a été accompli. Lors de l'appel d'offre, nous avions accordé une grande importance à la qualité technique de notre travail. Il était également important pour nous de bien tester chaque aspect de notre solution et de s'assurer qu'elle était robuste et intuitive. Selon nous, nous avons bien atteint ces objectifs avec notre pipeline de déploiement efficace, notre code bien révisé et nos applications intuitives et résistantes aux imprévus. On en conclut donc que notre

démarche de conception était appropriée au projet, et que l'effort commun de l'équipe avec la bonne organisation des tâches ont permis à ce projet d'être réalisé avec succès. Notre idée de base par rapport au système n'a pas beaucoup changé, et à part des améliorations relatives à la qualité de code, nos fonctionnalités n'ont pas nécessité de modifications majeures. Bien définir le projet dès la rédaction du rapport nous a donc été très bénéfique. Selon nous, nous avons également bien été capables d'interagir avec le client et de s'ajuster à ses demandes à la suite du second livrable, ce qui était également l'un des buts du projet. Finalement, puisque nous avions une vision claire et une bonne organisation, nous avons pu livrer un produit structuré et des micro-services s'intégrant bien les uns aux autres. Nous avons même eu le temps d'ajouter notre touche personnelle, ce dont nous sommes fiers.

8. Références

[1] Stefani, M. (2019). oktal/pistache. [En ligne] GitHub.
Disponible: https://github.com/oktal/pistache.

[2] Sqlite.org. (2019). *SQLite Documentation*. [En ligne]
Disponible: https://www.sqlite.org/docs.html.

[3] Marulidharan, A. (2019). Arun11299/cpp-jwt. [En ligne] Github.
Disponible: https://github.com/arun11299/cpp-jwt.

[4] Lohmann, N. (2019). Nlohmann/json. [En ligne] Github.
Disponible: https://github.com/nlohmann/json

[5] Okada, S. (2019). Okdshin/PicoSHA2. [En ligne] Github.
Disponible : https://github.com/okdshin/PicoSHA2

[6] Goncharov, D. (2019). Neargye/magic_enum. [En ligne] Github.
Disponible: https://github.com/Neargye/magic_enum.

[7] Yang, H. (2019). Base64 Encoding Algorithm. [En ligne] Github.
Disponible:http://www.herongyang.com/Encoding/Base64-Encoding-Algorithm.html

[8] ZeroMQ project. (2019). Zeromq/libzmq [En ligne] Github.
Disponible : https://github.com/zeromq/libzmq