Département de génie informatique et de génie logiciel

INF3995

Projet de conception d'un système informatique

Rapport final de projet

Prédiction de données sur l'usage du BIXI

Équipe No. 6

Anastasiya Basanets

Samuel Charbonneau

Jean-Olivier Dalphond

Olivier Dion

Mariam Sarwat

8 Décembre 2020

Table des matières

1. Objectifs du projet

L'entreprise *BIXI* souhaite mettre sur pied un système informatique complet de présentation des données d'utilisation. Le projet en question doit fournir une solution pour permettre de consulter les données *BIXI* et pour prévoir, de façon efficace, l'utilisation éventuelle des différentes stations *BIXI*. Il doit inclure différentes composantes informatiques pertinentes et interactives pour les utilisateurs, mais également pour les administrateurs. Il était demandé de diviser en différents sous-systèmes le *back-end* (côté serveur et engins de données) afin d'accélérer les processus et d'améliorer la disponibilité.

Dans la solution proposée par l'équipe, une application *Android* qui permet à un utilisateur d'entrer des informations personnelles, soit remplir un sondage, permettant de construire une base de données sur les usagers qui consultent l'application. Ensuite, on doit y trouver une section de recherche de station, qui puisse être affichée sur une carte une fois trouvée et sélectionnée. De plus, il doit être possible d'y consulter les données historiques d'utilisateurs du système *BIXI*. Finalement, l'application doit inclure une section qui permet aux usagers de visualiser l'utilisation des stations dans le futur. Notons que cette dernière section devra mettre à profit de l'intelligence artificielle afin de prédire les données.

Un logiciel pour ordinateur doit être créé pour permettre à un administrateur de consulter les allées et venues des utilisateurs de l'application mobile et de surveiller la disponibilité de l'ensemble du système. L'administrateur doit être en mesure d'y lire le journal d'information des différentes requêtes effectuées sur le serveur et les engins de données.

Il est important que l'ensemble du système informatique soit sécurisé et disponible en tout temps. En cas de panne du côté des services, les deux applications de type client ne doivent pas être affectées. En d'autres mots, les deux applications doivent être en mesure de gérer les différentes erreurs qui peuvent survenir, ceci incluant bien sûr les pannes de service. De plus, les engins de données doivent être en mesure de gérer leur propre perte.

2. Description du système

2.1 Le serveur (Q4.5)

Le serveur web a été conçu spécifiquement pour fournir une interface *REST API*. Il est capable de s'adapter à différentes situations de production et de développement,

tout en restant découplé en différents modules pour faciliter son développement et sa maintenance. Le serveur possède un environnement de développement flexible permettant de le configurer à la compilation. Enfin, les dépendances sont peu nombreuses, et ce pour garder le contrôle sur la base du code, mais aussi pour s'assurer que le serveur va passer l'épreuve du temps.

Premièrement, le serveur est hautement flexible par rapport à l'environnement dans lequel il va être exécuté. En effet, il est possible de choisir sur quel port les connexions entrantes doivent être écoutées et sur quelle interface écouter. Cela est pratique quand on veut tester localement en utilisant l'interface *loopback* et en utilisant un port qui ne requiert pas de droit administrateur. Aussi, il est possible de faire le chiffrement par *TLS* en choisissant une clef et un certificat. En d'autres mots, il est possible de faire le chiffrement par le serveur, sans utiliser de proxy. De plus, il est possible de choisir le répertoire où travaille le serveur. C'est en effet dans ce répertoire que les informations des sondages et d'authentification des utilisateurs sont contrôlées.

Deuxièmement, le serveur est divisé en différents modules pour faciliter son développement ainsi que sa maintenance. Le premier module est celui des différents utilitaires (utils/). Chaque utilitaire offre une interface à des fonctionnalités qui peuvent être utilisées par différents modules qui en ont besoin comme bloc de fondation. Par exemple, on y trouve des structures de données telles que des listes (include/utils/list.h) qui proviennent du noyau de *Linux*. On y retrouve aussi des tampons dynamiques génériques (include/utils/buf.h) et un arbre binaire (utils/btree.c). On y retrouve aussi un analyseur grammatical ainsi qu'une interface pour manipuler des données *JSON* (utils/json-parser.y, utils/json.c). De plus, on y trouve un scanneur lexical et une interface pour manipuler des données *HTML* (utils/http-scanner.c, utils/http.c). Enfin, il y a un analyseur syntaxique pour les options passées en invite de commande. Le second module est le noyau (core/). Il utilise les utilitaires du précédent module pour faire la logique du serveur. En d'autres mots, il sert de colle entre les différents modules. C'est ce module qui gère les authentifications (core/auth.c), les sondages (core/poll.c) et qui fournit une interface *REST API* aux autres modules (core/rest.c). Le troisième module est celui des routes (routes/). Ce module utilise les différents utilitaires ainsi que l'interface *REST API* offerte par le module noyau pour implémenter des routes accessibles par les clients. Le dernier module est celui des tests (tests/). Ce dernier est chargé en mémoire, lorsque l'on souhaite vérifier l'intégrité du serveur, et installer des routes au serveur qui sont utilisées par des scripts exécutés en parallèle.

Troisièmement, le serveur possède un système de configuration qui permet de configurer, à la compilation, le comportement du serveur. Il s'agit d'un fichier texte en *Lua*.

Pour terminer, les dépendances sont minimales. En effet, pour l'exécution, seule la librairie *OpenSSL* est requise. Cependant, pour la compilation, il est nécessaire d'avoir recours à *GCC* (ou Clang), *GNU Make*, *GNU Bison* et *GNU Flex*. Ces outils existent depuis des dizaines d'années et garantissent la robustesse de leurs codes générés.

2.2 Les engins de données

Comme spécifié dans les requis du projet, les engins de données ont tous été conçus avec le langage *Python*. Afin de simplifier le code, chaque engin de données est indépendant ce qui favorise, entre autres, la réutilisation des différents modules qui seront utilisés par chaque engin. La figure suivante présente l'architecture des engins de données :

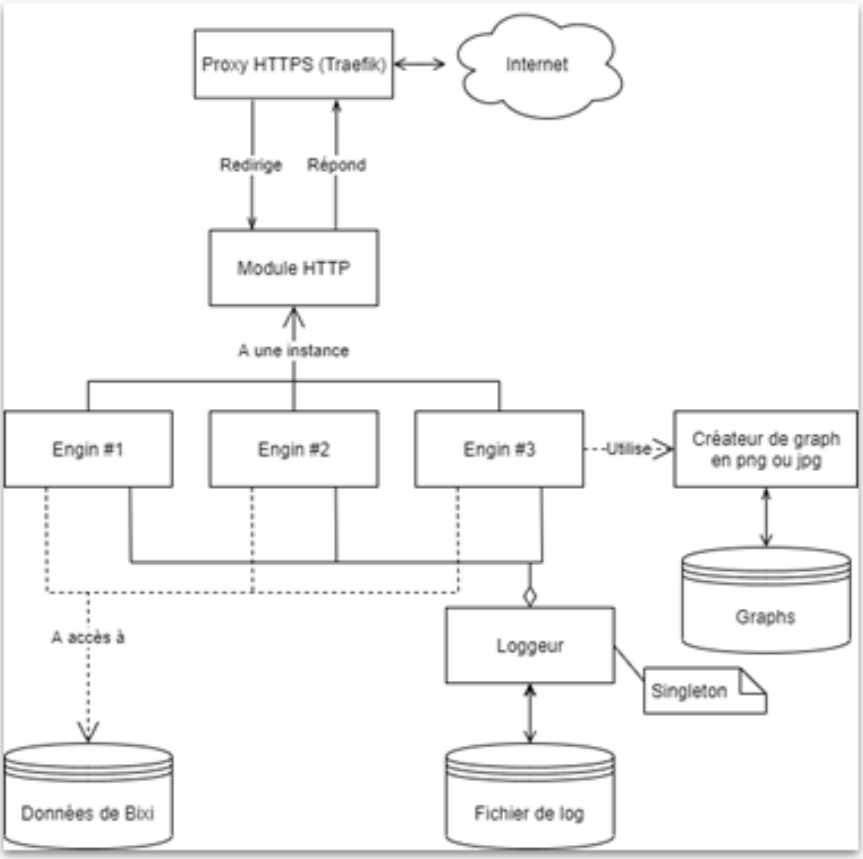

Figure 1. Architecture engin de données © 2020 Samuel Charbonneau.

Un premier module s'occupe de la gestion des requêtes *HTTP*. Les requêtes *HTTPS* des clients sont reçues par le proxy inverse qui lui va rediriger via une requête *HTTP* qui sera reçue par le module *HTTP*. Ce module va comparer la requête reçue avec

les routes prédéfinies pour chacun des engins. Ces engins définissent le code à exécuter pour chacune des routes dont il gère. L'association d'une route à une exécution de code se fait via un décorateur de fonction. Dans celui-ci, on retrouve donc la structure de la route qui permettra d'exécuter la bonne fonction dans l'engin selon les requêtes reçues par le module *HTTP*.

De plus, nous avons utilisé le module de journalisation de *Python* afin de journaliser les différentes étapes dans les requêtes. Ainsi, on peut consulter dans ce journal la date, l'heure et la nature de la requête reçue par l'engin ainsi que des informations à des moments clés lors de l'exécution du code. Chacun des engins a besoin de récupérer de l'information sur les stations *BIXI*. Toutes ces informations sont stockées dans une base de données à laquelle nous accédons par l'entremise de la librairie *pandas*.

Le premier engin de données s'occupe de récupérer les informations des stations *BIXI*. Les requêtes servent donc de recherche et d'acquisition d'information sur les stations. Il n'y a aucune information qui est reliée au nombre d'utilisation des stations, mais seulement leurs caractéristiques (code, nom, latitude et longitude). Par la suite, le deuxième engin sert les requêtes de récupération d'utilisation historique des stations *BIXI*. Cet engin récupère donc les données d'usage dans la base de données et ne fait que les préparer avant de les retourner au client. Finalement, le dernier engin s'occupe de donner les informations de prédiction sur l'utilisation des différentes stations. Il fait appel à l'intelligence artificielle afin d'arriver à ses fins. Cet engin interagit avec un module qui génère des graphiques pour une représentation visuelle intéressante des données de prédiction. Afin de gérer séparément les engins, nous utilisons docker.

Chaque engin est encapsulé dans un conteneur Docker. La synchronisation des conteneurs s'effectue avec *Docker Compose*. Finalement, la communication entre ces conteneurs du serveur et les engins s'effectue via la fonction *Linux poll()*.

En somme, les librairies utilisées par tous les engins sont donc *pandas*, *HTTP*, *logging*, *datetime* et *json*. De plus, l'engin 3 utilise d'autres librairies comme *matplotlib* et *numpy* pour exécuter les opérations de prédiction et créer des graphiques. Pour utiliser l'algorithme des forêts aléatoires, la librairie *Scikit-Learn* est utilisée. Lorsque l'usager décide d'obtenir une prédiction selon une station précise ou pour toutes, le résultat est affiché sur le graphique, mais aussi sauvegardé dans une mémoire *cache*. Ainsi, lorsque l'usager demande une présentation des erreurs, la dernière prédiction est montrée sur le graphique, en plus des données réelles, pour observer la comparaison. Bien

évidemment, si la *cache* est vide, une erreur survient, car il doit avoir une prédiction complétée au préalable.

2.3 Tablette (Q4.6)

Pour l'application Android, nous avons décidé d'utiliser l'architecture *MVVM*, très en vogue actuellement. Celle-ci permet une plus grande flexibilité quant à la manipulation de la vue, sans affecter la logique applicative et la manipulation des données. Ainsi, nous avons un *Model*, un *ViewModel* et une *View* pour chacun des quatre onglets de l'application: Paramètres, Sondage, Recherche (avec vue *Google Maps*) et Statistiques. La figure suivante présente une vue d'ensemble sommaire de l'architecture de l'application *Android* du projet. Notons qu'en ce qui concerne la vue, seuls les onglets qui seraient destinés à rester dans le produit remis aux usagers ont été retenus pour la figure.

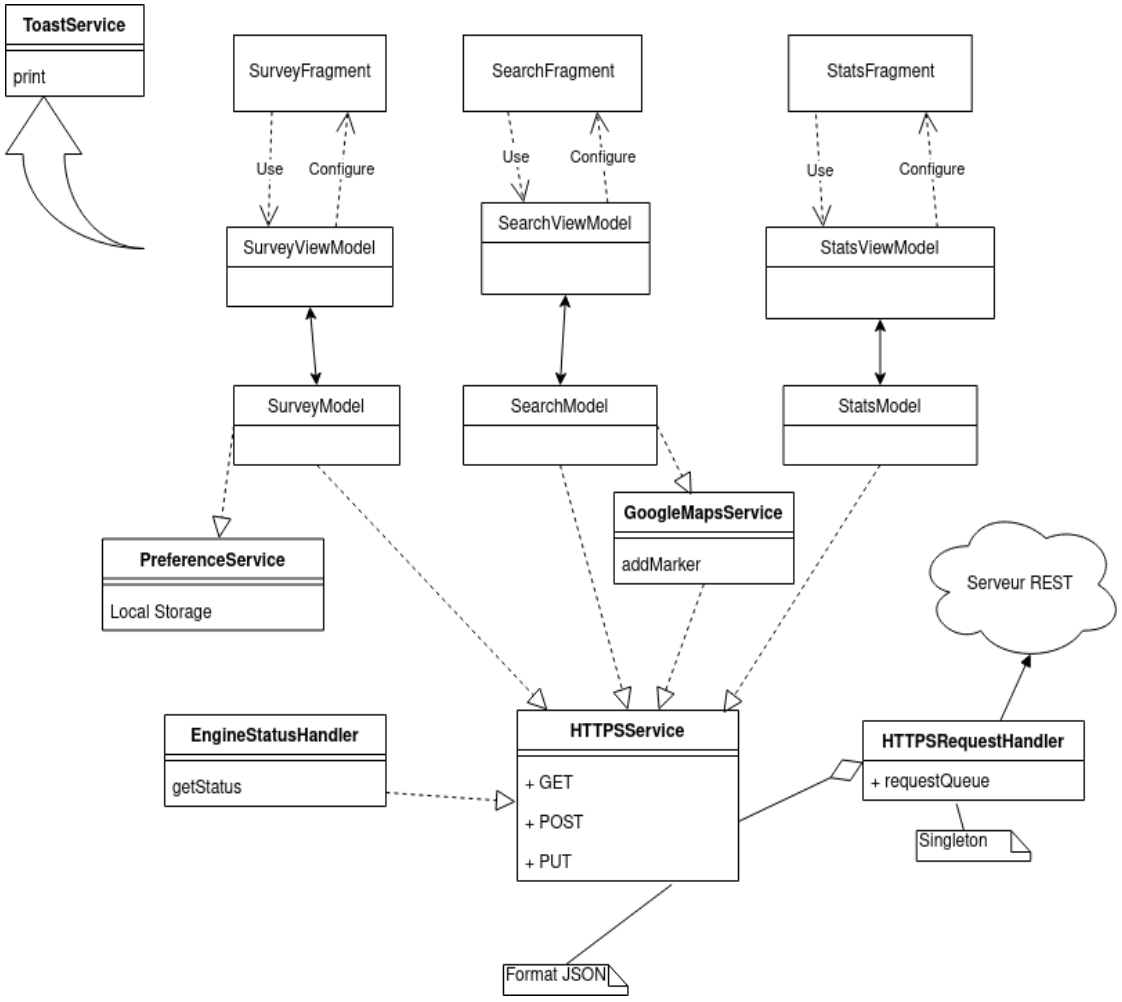

Figure 2. Architecture sommaire de l'application Android PolyBixi. © 2020 Jean-Olivier Dalphond, Samuel Charbonneau.

Notre choix de librairie et de standard de design s'est arrêté sur *Material*, puisque celle-ci s'harmonise parfaitement avec le reste du système d'exploitation et offre une structure facile à comprendre et à implémenter. C'est également une librairie très performante et très élégante. Pour la communication avec le serveur, nous avons décidé de conserver la même librairie que pour le prototype livré en début de session, soit *Volley*. L'intégration du certificat de sécurité a posé un certain défi, mais une fois que c'était complété, nous avons pu nous concentrer sur les fonctionnalités et la présentation des données provenant des engins et du serveur. Nous utilisons *JetPack*, faisant partie de la suite *AndroidX*, pour la navigation entre les différents fragments, qui est une librairie bien documentée et créée par *Google* spécifiquement pour *Android*. L'implémentation s'est d'ailleurs avérée très facile.

L'usager est d'abord accueilli dans l'onglet Paramètres, qui lui permet de spécifier l'adresse *IPv4* du serveur *REST* du système. Par souci de convivialité, nous avons pré-rempli ce champ pour l'usager, avec l'adresse de notre serveur *Linode* dédié. En cliquant sur Sauvegarder, on est redirigé vers le sondage.

L'onglet Sondage est divisé en deux vues, accessibles l'une par l'autre. Lors d'un premier accès à l'application, ou encore s'il indique qu'il veut remplir le sondage de nouveau, l'usager est invité à remplir les champs du sondage et à soumettre les données au serveur. Une fois que le serveur a accepté les données, l'application affiche le logo de *BIXI* et un message de remerciement, puis offre à l'usager deux options: aller à la recherche avec la carte de la ville, ou encore remplir de nouveau le sondage comme mentionné.

Dans l'onglet Recherche, nous avons utilisé *Google Maps* comme outil pour afficher la carte de Montréal et permettre l'interaction avec celle-ci. Nous avons ajouté des marqueurs avec *l'API* de *Google Maps* pour chaque station *BIXI*, dont les informations arrivent des engins de données. Nous voulions utiliser *OpenStreetMaps* en début de projet, mais la convivialité de *Google Maps* par rapport à l'environnement Android nous a convaincus de faire le changement. Pour utiliser une expression populaire, c'est « plug and play », malgré certains bogues que nous avons rencontrés. Pour gérer la liste des stations obtenue par la recherche de l'usager, et ajuster la vue selon le nombre de résultats, nous avons utilisé *RecyclerView* qui fait partie de la suite d'outils *AndroidX*. Que ce soit via la recherche textuelle ou sur la carte directement, on peut choisir une station précise et, lorsqu'on clique sur celle-ci via le marqueur ou le texte correspondant dans la boîte de recherche, la carte se précise et effectue un zoom vers l'emplacement précis de la station. On y voit apparaître un encadré avec son nom et son

numéro. L'usager peut, s'il désire obtenir des statistiques sur l'historique d'utilisation par heure de la station, appuyer sur cet encadré. Nous trouvons que cette façon d'accéder à l'information pertinente est optimale en termes d'expérience usager.

L'onglet Statistiques est divisé en deux parties, dont la disposition serait normalement revue pour une version officielle pour l'usager (on retirerait les choix qui sont en lien avec l'affichage des erreurs de l'engin de prédiction). La partie du haut est constituée d'un formulaire en trois parties, soit le type de requête de données (historique, prédiction ou erreur), la période de requête (par mois, heure, jour de l'année, jour de la semaine, température) et le numéro de la station, ou toutes les stations. Bien évidemment, l'usager ne recevra aucun résultat s'il rentre une mauvaise station. Il y a également un bouton pour soumettre la requête au serveur. Lorsque des résultats sont disponibles, le graphique est construit au-dessous du formulaire. Nous avons choisi de laisser le plus d'espace possible à celui-ci, puisque c'est l'utilité principale de cet onglet. La librairie utilisée pour construire et afficher les graphiques est *AAChart*. Nous avons constaté les limites de *MPAndroidChart* assez rapidement, puis avons entrepris de trouver une librairie gratuite qui convenait parfaitement à nos besoins. Cela n'a pas été de tout repos, mais nous avons finalement opté pour *AAChart*. Cette librairie a le défaut de ne pas permettre de spécifier un titre pour l'axe des X, mais coche toutes les autres cases. Pour pallier ce défaut, nous avons précisé le descriptif de chaque valeur sur l'axe des X, et nous avons inclus dans le titre du graphique l'unité qui y est représentée. Après consultation, nous considérons que c'est suffisamment clair. Pour les requêtes en historique, nous présentons les données sous forme de colonnes, et pour la prédiction, sous forme de ligne. La présentation des erreurs de prédiction se fait tout simplement en combinant les deux formats.

À travers tous les onglets, la présentation visuelle respecte le code de couleurs de *BIXI*, soit le rouge et le bleu (pour les vélos électriques). Nous avons également voulu laisser la liberté à l'usager d'utiliser la navigation en suivant les étapes depuis les paramètres jusqu'aux statistiques, ou encore d'accéder à l'onglet de son choix via le menu du bas. Un des objectifs que nous avons tenté de garder en tête dans la conception de l'application était de garder une présentation sobre et minimiser le nombre d'éléments.

Du point de vue de la logique de l'application, nous avons créé des services pour faciliter la gestion des données et l'interaction avec la tablette *Android*. Nous avons un service pour garder en mémoire la soumission d'un sondage et ne pas solliciter l'usager à la réouverture de l'onglet. Nous avons également un service qui regroupe toutes les fonctions de *Google Maps* qui sont utiles pour notre projet. Un autre service est

responsable des notifications *Toast* (natif à *Android*), puis nous avons deux services qui, de concert, permettent de centraliser toutes les interactions sécurisées avec le serveur et les engins de données. Pour alléger le code, nous avons regroupé la plupart des valeurs constantes, messages et classes de données dans le fichier *Utils*.

De façon générale, nous avons tenté de créer une application avec une interface légère et intuitive. Comme mentionné, il est possible de naviguer à travers les fonctionnalités de l'application en suivant les étapes logiques (navigation) ou encore via les onglets présentés dans le menu du bas. Nous estimons qu'il s'agit d'une bonne façon de fournir l'accès à toutes les fonctionnalités sans emprisonner l'usager dans une marche à suivre très stricte et une linéarité sans intérêt. En ce qui concerne la performance de l'application, nous avons tenté d'alléger le plus possible chaque onglet, surtout en considérant le poids qui vient, d'emblée, avec *Google Maps* et la génération des graphiques par la tablette elle-même. Cependant, ces choix offrent une expérience accrue pour l'usager, car il peut interagir avec la carte en zoomant, et avec les graphiques en sélectionnant des données précises. Dans l'onglet recherche, nous avons choisi d'afficher au fur et à mesure les résultats de la recherche faite par l'usager, ce qui tire profit de la performance de notre serveur. On en retire une plus grande immersion dans l'application, puisqu'on constate la disponibilité quasi immédiate des données des stations.

2.4 Application sur PC

L'application *PC* a été codée en utilisant le langage *Python* (version 3.8) ainsi que la librairie *PyQt5* qui est utilisée principalement pour l'interface usager de l'application. Cette dernière suit un modèle *MVC* assez simple. Le système possède un seul modèle. L'application contient 5 pages différentes, soit la page de connexion, la page d'accueil (le menu), la page pour changer le mot de passe du compte administrateur, la page pour visualiser les messages de journalisation des engins de données en temps réel et la page pour afficher les résultats du sondage. Nous avons donc décidé de faire en sorte que chaque page de l'application ait son propre contrôleur et sa propre vue. La figure suivante présente une vue d'ensemble sommaire de l'architecture de l'application *PC* du projet :

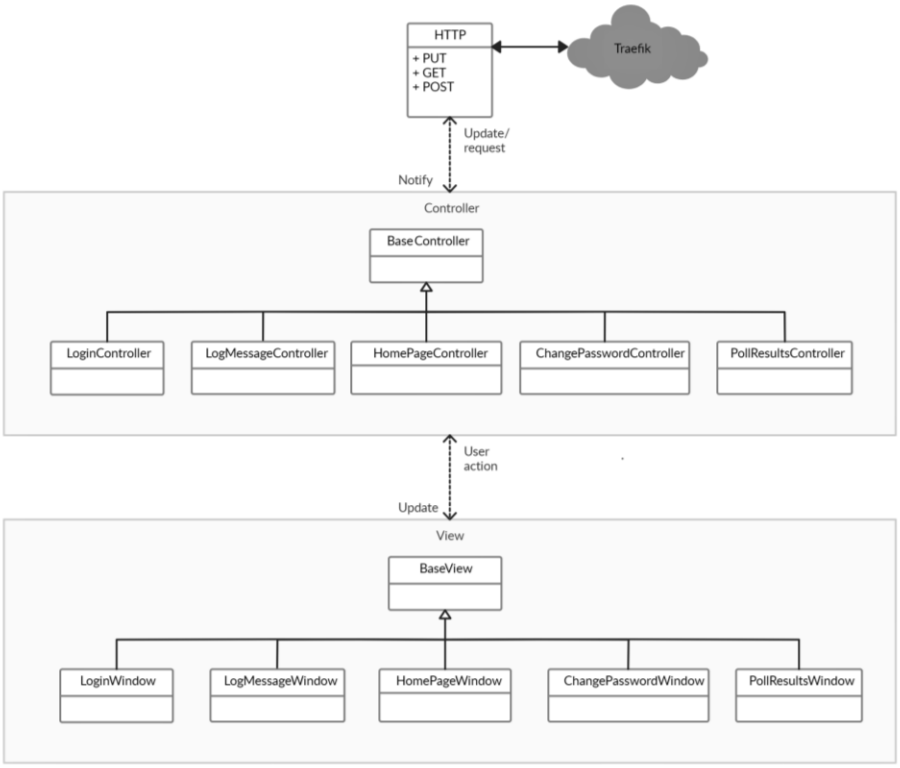

Figure 3. Architecture sommaire de l'application PC. © 2020 Mariam Sarwat.

Tout d'abord, le modèle est réalisé par la classe *HTTP* (contenue dans server_request_model.py), qui gère les requêtes. Celle-ci contient les méthodes *GET*, *PUT* et *POST* qui seront utilisées par les différents contrôleurs. Comme indiqué dans les spécifications du projet, chaque requête utilise le protocole de base *HTTP* et emploie un certificat pour s'identifier. Les requêtes sont effectuées à l'aide de la librairie *Requests* contenue dans la version 3.8 de *Python*.

Par la suite, tel que mentionné plus haut, nous avons choisi d'avoir un contrôleur par fenêtre de l'application pour un total de cinq contrôleurs. Afin de regrouper les méthodes communes entre les classes, nous avons décidé d'implémenter une classe de base s'appelant *BaseController*. Celle-ci inclut principalement les méthodes qui permettent une redirection vers une autre fenêtre de l'application. Le contrôleur de base est hérité par chaque contrôleur ce qui permet d'éviter la répétition de code. De plus, les contrôleurs partagent une seule instance de *HTTP*, soit le modèle, contenu dans le fichier *Auth*. Dans les points qui suivent, le contrôleur de chaque page est décrit sommairement :

- *LoginController* : responsable de vérifier le mot de passe ainsi qu'a validé l'adresse *IPv4* spécifiée par l'usager. Lors d'une connexion réussie, l'usager est redirigé vers

la page d'accueil de l'application. Dans le cas qu'une connexion à l'adresse *IPv4* spécifiée échoue, un message d'erreur avertit l'utilisateur.

- *HomePageController* : s'occupe uniquement de la redirection vers les différentes fenêtres.

- *ChangePasswordController* : utiliser pour changer le mot de passe du compte administrateur du côté du serveur web. La réponse reçue de la part du serveur web détermine le type d'erreur que retourne le contrôleur à la vue.

- PollResultsController : responsable de chercher les données du sondage auprès du serveur web. Si la réponse retournée par le serveur est valide, la méthode *json*() de la librairie *response* permet d'extraire les résultats du sondage. Ces résultats sont par la suite concaténés pour suivre le format que nous avons choisi et sont insérés dans une liste qui sera retournée à la vue.

- *LogMessagesController* : responsable à récupérer les messages de journalisation auprès des engins de données. Tout comme dans *PollResultsController*, on extrait les messages en texte ainsi que les graphiques seulement si la réponse retournée par les engins est valide. Dans le cas d'une erreur, le contrôleur retourne à la vue un message d'erreur à afficher. Nous avons décidé d'envoyer les messages par blocs selon le type de données. En d'autres mots, le contrôleur reçoit un *JSON* contenant seulement des messages en texte, des messages de type graphiques ou rien lorsqu'on atteint la fin du fichier. On demande donc aux engins de données des messages jusqu'à tant qu'on reçoive une indication qu'on est rendu à la fin.

De plus, comme il a été mentionné auparavant, nous avons choisi d'avoir une vue par fenêtre de l'application pour un total de cinq vues. Encore une fois, tout comme pour les contrôleurs, nous avons décidé d'implémenter une classe de base, soit *BaseView*. Celle-ci inclut principalement les méthodes qui permettent de spécifier les paramètres relatifs à une fenêtre. La vue de base est alors héritée par chaque vue ce qui permet d'éviter la répétition de code encore une fois. Dans les points qui suivent, la vue de chaque page est décrite sommairement :

- *LoginWindow* : contient la mise en page pour la page de connexion de l'application. Le champ de nom utilisateur est statique et ne peut donc pas être changé. Le champ d'adresse *IPv4* contient un validateur qui accepte seulement des adresses valides.

- *HomePageWindow* : contient la mise en page pour la page d'accueil de l'application. La page contient trois boutons qui redirigent l'utilisateur vers les différentes fonctionnalités offertes.

- *ChangePasswordWindow* : contient la mise en page pour la page changer le mot de passe. Afin de pouvoir changer le mot de passe du compte avec succès, il faut que les deux champs contiennent exactement la même chaîne de caractères. Nous avons aussi décidé que l'utilisateur ne peut pas choisir un mot de passe équivalent à une chaîne vide. Dans le cas d'une erreur ou d'un succès, la page affiche un message.

- *PollResultsWindow* : contient la mise en page pour la page de consultation des résultats du sondage. Cette page utilise des *threads* afin de pouvoir chercher les résultats du sondage tout en gardant l'interface usager réactive. La page contient un bouton de mise à jour qui appelle la fonction « get_poll_results » de son contrôleur. Les résultats retournés par le contrôleur sont par la suite affichés sur la page en utilisant un *QFormLayout*.

- *LogMessagesWindow* : contient la mise en page pour la page consultation des messages de journalisation des engins de données. La page contient un onglet par engins de données. Encore une fois, nous avons employé des *threads*. Dans ce cas, il y a un *thread* par engin de données. Au départ, nous affichons un *GIF* pour signaler à l'utilisateur que nous sommes en train de charger les messages. Après avoir chargé tous les messages initiaux contenus dans les journaux des engins, nous démarrons l'instance de *RepeatedTimer* qui cherche les nouveaux messages à chaque trois secondes. Cet intervalle peut être changé.

  **RepeatedTimer* : Appel une fonction à un intervalle spécifié par l'utilisateur.

Finalement nous avons quelques fichiers supplémentaires primordiaux au fonctionnement de l'application. Tout d'abord, le dossier *Common* contient les énumérations nécessaires pour les différents messages d'erreurs ainsi que les codes d'états possibles du serveur. De plus, le « main » est contenue dans le fichier *bixi-pc* qui se retrouve à la racine du dossier *PC*. Ce dernier crée l'instance app et lance la page de départ de l'application, soit la page de connexion.

2.5 Fonctionnement général (Q5.4)

Nous avons choisi de déployer le serveur et les engins de données sur un serveur fixe *Linode*. Il s'agit d'un service comparable à *Amazon EC2*, par lequel on peut louer du temps de calcul d'une instance d'un serveur distant, et qui est peu coûteux. En configurant le *Linode*, nous pouvons choisir la distribution *Linux* de base qui le compose, et nous y connecter en *ssh* ou via un terminal virtuel disponible sur le site web. Une fois la configuration initiale terminée, nous avons pu installer des paquets comme *Docker*, *Git* et *Python* afin de pouvoir exécuter notre code et construire le système. Un fichier Dockerfile est nécessaire pour chaque engin de données, pour le serveur et pour *Traefik*.

Afin d'orchestrer le tout de façon simple et efficace, les différents services du système *PolyBixi* sont lancés à l'aide de *docker-compose*. Le serveur web, pour sa part, écoute sur le port 80 dans son réseau. Pour les engins, un script dit de *entrypoint* est exécuté avec le numéro de l'engin. Ce script assure aussi que les engins sont lancés en mode *release*. Chaque engin écoute sur le port 80 dans leur réseau. Enfin, les ports du serveur et des engins sont automatiquement redirigés vers des ports éphémères par *Docker*. De cette façon, Traefik apprend les numéros de port des différents services par *Docker*. Tout cela est possible grâce au *namespace* des réseaux qu'offre *Docker*.

Le point d'entrée principale du système s'agit de l'application *Android*, qui incorpore la majeure partie des composantes fonctionnelles. En effet, on peut y remplir un sondage, on y présente la localisation et les informations de chaque station *BIXI* et on peut y obtenir des statistiques d'utilisation historique et des prédictions. C'est donc via l'application *Android* que l'usager peut interagir avec le serveur et les engins de données. Nous avons inclus le fichier *APK* dans la racine de notre entrepôt *Git*, à défaut d'avoir soumis notre application à *Google* pour qu'elle puisse être téléchargée sur le *Play Store*.

Lorsqu'on ouvre l'application la première fois, l'onglet Paramètres s'ouvre automatiquement, et on peut y configurer l'adresse *IPv4* du serveur. L'adresse du serveur *Linode* que l'on a utilisée pendant le projet est inscrite par défaut, de façon à accélérer l'accès aux données pertinentes. En confirmant l'adresse, l'onglet Sondage est ensuite affiché et l'usager est invité à soumettre ses informations de communication et à indiquer son intérêt pour l'infolettre (fictive, soit). Une fois le sondage rempli, le logo *BIXI* est affiché et on remercie l'usager, puis il peut choisir de remplir un nouveau sondage ou naviguer vers l'onglet de recherche. Dans ce dernier, une carte interactive de Montréal est présentée et, peu après, un marqueur est à l'emplacement de chaque station *BIXI*. On peut y effectuer une recherche textuelle dans le seul champ du haut, ou alors choisir une

station en manipulant *Google Maps* qui occupe le reste de l'écran. Après avoir choisi une station par l'une des deux méthodes, l'usager est redirigé vers l'onglet Statistiques, qui ouvre les données historiques par heure de la station choisie. Dans cet onglet, on peut donc effectuer une requête en historique, mais également obtenir une prédiction, et afficher les erreurs de l'engin de prédiction. L'usager peut obtenir ces informations pour une station en particulier s'il en connaît le code, ou encore pour toutes les stations. Les types de prédictions ou de données historiques présentées dans la section des engins de données du présent rapport sont bien sûr disponibles via les boutons radio au haut de l'onglet.

L'autre façon d'interagir est via l'application *PC*, qui présente les informations pertinentes aux membres de l'administration. On peut y trouver les entrées du sondage, l'état de chaque engin de données, et bien sûr, modifier le mot de passe de l'administrateur. Tout d'abord, l'application utilise une version de *Python* 3.8 ainsi que la librairie *PyQt5*. Il faut alors installer les deux avant de pouvoir rouler l'application. Lorsque les dépendances sont installées sur la machine, nous pouvons lancer l'application. Pour ce faire, il faut se diriger vers le dossier *PC* dans le projet et ouvrir un terminal où l'administrateur peut exécuter la commande « make run ». Après quelques secondes, la fenêtre de connexion s'affiche. En entrant le bon mot de passe ainsi qu'une adresse *IPv4* valide du *Linode*, l'administrateur se rend à la page d'accueil de l'application. À partir de cette page, l'administrateur peut accéder à la page de changement du mot de passe, la page d'affichage des messages de journalisation des engins de données ainsi que la page de consultation des résultats du sondage. Chaque page contient un bouton « retour vers la page d'accueil ». Tout d'abord, la page de changement de mot de passe permet de changer le mot de passe du compte administrateur. Par la suite, la page des messages de journalisation contient trois onglets, soit un par engins de données. Pour faciliter un repère plus rapide des messages importants, les messages d'erreur sont en rouge et les messages d'avertissement (warning) sont en jaune-orange. La mise à jour de la page se fait à chaque trois secondes. Finalement, la page de consultation des résultats des sondages affiche le nombre de sondages qui ont été remplis ainsi que toutes les données. La page contient un bouton « mettre à jour la page » qui permet de demander de nouveau au serveur web tous les résultats de sondage.

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

Notre projet met de l'avant une intégration continue et les différents systèmes ont été testés avec rigueur. On peut donc affirmer qu'ils sont robustes. Lorsque nous avons

fait les essais finaux avec l'application *PC* et l'application *Android*, nous nous sommes assurés d'inclure tous les cas possibles que nous avions en tête. Bien sûr, vu le temps limité que nous avions, nous n'avons pas pu tout régler à 100%, mais nous sommes confiants de nous en être approchés.

À titre d'exemple, il nous est arrivé de voir l'application *Android* planter dans des cas plus particuliers, comme la rotation de la vue sur la tablette ou le changement de thème. Dans une utilisation plus normale, nous n'avons pas rencontré de tels problèmes. L'entrée de données erronées, très longues ou invalides ne produit pas un tel comportement. Les différentes vues ne s'adaptent pas correctement à des appareils plus petits, et n'ont pas été testées avec des appareils plus grands (relativement rares, cela étant dit). De plus, l'utilisation de l'algorithme des forêts aléatoires a permis d'obtenir des précisions relativement précises. Malgré les limites de cet algorithme, la précision se situait autour de 90%. Pour une première utilisation d'algorithme, ce résultat est convenable.

De façon plus générale, l'évaluation pratique de notre projet s'est déroulée de façon très satisfaisante au cours de la session. Les résultats obtenus en témoignent d'ailleurs avec grande éloquence. En d'autres mots, il n'y a pas de requis qui ont échappé à notre attention, et nous avons répondu aux exigences avec brio.

4. Déroulement du projet (Q2.5)

D'un point de vue technique, toutes les spécifications ont été respectées. Certains aspects étaient plus difficiles que prévu, mais nous avons été capables de surmonter les défis. Par exemple, l'implémentation du certificat *SSL* et le *multithreading* des engins de données furent plus complexes que ce qu'on croyait. Nous nous sommes assurés de prendre le temps de comprendre la documentation pour assurer une implémentation optimale.

D'un point de vue fonctionnel, tout était bien réussi. Nous n'avons pas eu de difficultés à utiliser la tablette pour notre application Android ou à créer une interface *PC* pour l'administrateur.

D'un côté de gestion, notre seule difficulté était de mettre à jour *Redmine* le plus souvent possible. Puisque nous avions toutes sortes de technologies pour notre projet, il nous arrivait de délaisser *Redmine*, car c'était nouveau pour tous. On faisait les mises à jour dans les temps, mais on devait se le répéter deux à trois fois. Chacun est habitué à additionner les heures consacrées à un projet. Cependant, chacun à une technologie

propre à lui et différente de tous. Ainsi, en se retrouvant obligé d'utiliser *Redmine*, cela nous a pris du temps à nous y habituer et à ne pas oublier. En dehors de cela, la gestion du projet était réussie sans problèmes. Nous avons toujours effectué le travail dans les temps qu'on s'était fixés. Nous avons, chacun, partagé nos incertitudes lorsque cela arrivait et nous avons offert notre aide pour nous entraider. Notre atout majeur est que nous connaissons les capacités de l'autre et que nous avons confiance l'un envers l'autre. Ce fut la base de notre bonne communication.

D'un point de vue de test, malheureusement nous n'avons pas eu la chance d'écrire des scripts de test pour l'application mobile. Au départ nous voulions faire nos propres tests pour nous assurer qu'à chaque modification du programme, toutes les autres composantes soient toujours fonctionnelles. Cependant, par manque de temps nous nous sommes limités à faire des tests nous-mêmes sur la tablette directement. Nous avons quand même été en mesure de couvrir le plus de cas possible en effectuant ces tests. Il est important de noter que les autres modules ont été bien testés.

5. Travaux futurs et recommandations (Q3.5)

Afin de sécuriser et d'optimiser le système dans son ensemble, il serait intéressant de séparer les sous-systèmes en nœuds, de façon à assurer un bon niveau de performance et une certaine mise à l'échelle. Cela nous permettrait d'intégrer un nouveau nœud qui servirait de pare-feu, et cela faciliterait l'intégration d'une base de données dynamique, fondée sur des données constamment rafraîchies par l'utilisation des usagers.

Pour le serveur web, si le nombre de clients accédant aux ressources devient trop grand, il est recommandé de changer l'algorithme du serveur pour que celui-ci exécute les différentes requêtes dans des *threads* d'exécution en *userspace*. Avec une telle implémentation, l'auteur du serveur estime qu'il est facilement possible de gérer dans les dizaines de milliers voire les centaines de milliers de requêtes par seconde. De quoi servir l'ensemble des possibles usagers de la grande région de Montréal.

Par rapport à l'application Android, il faudrait améliorer la gestion des thèmes clair et sombre, améliorer la gestion du clavier virtuel pour le cacher ou le montrer lorsque c'est pertinent. De plus, il y aurait de l'optimisation de code à faire dans les services, notamment dans le service qui gère les requêtes *HTTPS*. De plus, il aurait été intéressant d'ajouter un système de recherche de station la plus près de l'usager. L'usager aurait pu permettre que l'application le localise et une recherche pour la station la plus proche aurait été faite. Ainsi un utilisateur aurait pu rapidement repérer la station la plus proche.

De plus, pour de meilleurs graphiques, il aurait été intéressant d'ajouter une option pour entrer une plage de dates. Les statistiques auraient donc été encore plus révélatrices qu'ils le sont actuellement.

Pour la prédiction, bien que l'algorithme de la forêt aléatoire fonctionne bien, il ne prédit pas au-delà de la plage des données d'entraînement. Ainsi, une extrapolation peut être faite, mais ce n'est pas la méthode la plus précise. En revanche, cet algorithme fonctionne comme une boîte noire. En d'autres mots, les résultats sont peu lisibles et peu explicatifs. Donc, il est difficile d'améliorer la prédiction. Une des méthodes serait de rajouter des données des années précédentes. Sinon, il serait intéressant aussi d'utiliser un autre algorithme de prédiction qui offrirait plus de précision.

En ce qui concerne l'application *PC*, il aurait été intéressant de permettre à l'usager d'agrandir et de rapetisser les fenêtres de l'application. La solution qui a été développée ne permet pas encore cette manœuvre. Ce choix a été fait comme la mise en page des widgets est statique. En d'autres mots, leur position est fixe et non dynamique. De plus, afin d'ajouter une flexibilité de plus, il aurait été intéressant d'ajouter un calendrier qui permet à l'utilisateur d'aller chercher les messages de journalisation des engins de données d'une journée ultérieure.

6. Apprentissage continu (Q12)

Pour Olivier, la gestion d'un environnement d'intégration continue était quelque chose de nouveau. En effet, plusieurs lectures ont dû être faites pour mettre en place et améliorer le *CI* pour garantir une qualité des logiciels. Aussi, Olivier comprend que l'utilisation de librairies tierces aurait pu sauver du temps de développement, mais celui-ci est convaincu que le serveur web est robuste et stable, et qu'il pourra facilement être maintenu où être amélioré dans le futur, dû au fait qu'une seule librairie, *OpenSSL*, a été utilisée. De même, l'implémentation d'un analyseur lexical pour *HTTP* a amené Olivier à lire le document *RFC 2616* pour comprendre la rationalité du protocole. Pour terminer, Olivier a beaucoup appris sur la gestion de projet ainsi que le cycle de vie de logiciel.

Pour Jean-Olivier, la conception macroscopique d'architectures logicielles et l'autonomie dans la programmation sont des aspects des compétences qui sont à améliorer. Ce dernier a toutefois mis les bouchées doubles, au courant de la session, pour vaincre le plus possible ces barrières et les résultats obtenus pour l'application Android sont, somme toute, bien satisfaisants. Avant de demander de l'aide à son équipe, il a toujours essayé de son côté en persévérant, si bien que beaucoup des problèmes rencontrés ont été réglés. À des fins d'amélioration, il lui faudrait prendre davantage de

temps à lire et analyser la matière (librairies, *API*, exemples, etc.) plutôt que de tomber directement dans l'implémentation.

La simplification du code et des façons de faire dans le langage *Python* fut un défi au courant du projet pour Anastasiya. Elle devait gérer beaucoup de données et s'assurer d'une mise en forme optimale pour les calculs de prédiction. Cela lui a demandé à maintes reprises de revoir la logique pour s'assurer d'inclure tous les détails qui étaient demandés. En explorant les différentes librairies de *Python* et les fonctions, elle a réussi à trouver les réponses à ses questions. Dans d'extrêmes cas, pour éviter de perdre trop de temps, elle a demandé de l'aide à Olivier, car elle savait qu'il avait plus d'expérience dans ce langage et pouvait lui conseiller la méthode la plus optimale. Comme Jean-Olivier, elle sait qu'il faudrait lire et analyser la documentation avant d'implémenter directement pour éviter de perdre du temps dans le futur à possiblement devoir tout recommencer.

En ce qui concerne Samuel, c'était un premier projet en python. Il a voulu bien comprendre le langage et a tiré profit de certains avantages que ce langage peut donner. Entre autres l'ajout de décorateur de fonction. C'était plus un défi pour lui qu'une lacune. Il a pu compter sur l'aide d'Olivier qui avait de l'expérience sur ce sujet. Par ailleurs, une lacune pour Samuel a été l'interface utilisateur pour l'application mobile. Il a pu compter sur Jean-Olivier pour la majorité de l'interface, mais il a eu de la difficulté à ajouter la liste déroulante pour la recherche. Il a cherché beaucoup en ligne, mais même en suivant les tutoriels, il n'arrivait pas à faire fonctionner la liste déroulante. C'est en lisant davantage de tutoriels et la documentation officielle qu'il a su combiner les informations pour parvenir à ses fins. Au début, il n'a pas essayé de comprendre comment faire, mais plutôt essayer d'implémenter directement les tutoriels. S'il avait pris le temps de lire la documentation et les tutoriels avant d'essayer de l'ajouter dans le programme, le temps total aurait pu être réduit.

Finalement, pour Mariam, tout comme pour la majorité de l'équipe, c'était la première fois qu'elle travaillait en *Python*. Par conséquent, il a fallu une période d'apprentissage. En se comparant à quelqu'un qui a plus d'expérience en *Python* et *PyQt5*, elle croit que l'organisation du code aurait pu être mieux réalisée. Dans ce sens, son manque d'expérience dans le sujet a amené à une lacune. Cependant, il est important de noter que le code est quand même clair et les méthodes sont bien définies. Il était important de bien documenter le code afin de faciliter les travaux futurs. Chaque méthode possède donc une courte description. De plus, l'utilisation de la librairie *PyQt* n'était pas toujours simple. En effet, il y a certaines instances qu'elle a dû faire des

recherches exhaustives pour trouver comment faire une bonne implémentation. Bien qu'elle ait toujours trouvé la solution qu'elle cherchait, parfois elle cherche trop loin. Dans ce sens, elle aurait pu s'améliorer en prenant un pas en arrière avant de revisiter le problème ou même parler avec un de ses coéquipiers. Finalement, comme elle n'a jamais eu la chance d'implémenter un projet qui requiert du *multithreading*. Elle a donc dû faire plusieurs recherches afin de trouver la meilleure solution pour le projet. En effet, elle a essayé trois différentes solutions avant de retrouver la solution la plus optimale. Si elle avait fait quelques recherches de plus, elle aurait pu sauver du temps.

7. Conclusion (Q3.6)

Le projet étant complété nous pouvons confirmer que la conception imaginée et celle réalisée sont similaires. Il s'est avéré que l'architecture du serveur de départ était un bon choix. Cette architecture est facilement capable de gérer le faible taux de requête actuel, ce qui est crucial pour la réalisation du projet. Cette architecture a permis de déployer rapidement le serveur web dans les premières semaines et peu d'additions ont dû être faites par la suite. L'architecture des engins de données avec l'outil Docker s'est avérée, elle aussi, une solution convenable. Malgré les défis de *multithreading*, la gestion des engins de données avec l'application mobile fonctionne aisément. De plus, cette architecture a permis la réutilisation des modules pour simplifier le code. L'algorithme des forêts aléatoire fut un algorithme fiable pour ce travail. L'implémentation de celui-ci a permis une précision de la prédiction entre 85 % et 90 %. C'est une précision adéquate pour cette première application utilisant l'intelligence artificielle. Bref, parfois, nous avons eu à revoir l'architecture, mais ces changements étaient mineurs et dans l'ensemble, nous avons respecté la stratégie établie dans l'appel d'offres.

Les essais faits sur le système final démontrent la qualité du travail accompli. La réalisation d'un projet de cette envergure demande une discipline de chacun, une bonne communication et une bonne cohésion. Nous avons réussi à terminer ce projet sans imperfection et sans dépasser le nombre d'heures requis. Pour ces multiples raisons, nous pouvons confirmer que ce projet est une réussite pour notre équipe et nous sommes satisfaits du travail réalisé.

L'expérience acquise lors du projet et l'excellent niveau de qualité du produit final nous portent à imaginer la suite : quel projet d'entrepreneuriat pourrait-on vouloir entamer maintenant ou après nos études, fiers de notre récente expérience?

8. Références

Algar, D. (2017, Avril 18). *Kotlin With Volley*. Récupéré sur Varvet: https://www.varvet.com/blog/kotlin-with-volley/

AnyChart. (s.d.). *AnyChart Android Charts*. Récupéré sur AnyChart: https://www.anychart.com/technical-integrations/samples/android-charts/

CodePath. (2020). *Using the RecyclerView*. Récupéré sur CodePath: https://guides.codepath.com/android/using-the-recyclerview

Developers Android. (2020, Novembre 11). *Create dynamic lists with RecyclerView*. Récupéré sur Developers Android: https://developer.android.com/guide/topics/ui/layout/recyclerview

Docker. (2020). *Compose file version 3 reference*. Récupéré sur Docker Docs: https://docs.docker.com/compose/compose-file/

Fielding, & al. (1999, Juin). *RFC 2616 : Hypertext Transfer Protocol -- HTTP/1.1*. Récupéré sur Tools IETF: https://tools.ietf.org/html/rfc2616

GitLab. (2020). *GitLab CI/CD pipeline configuration reference*. Récupéré sur GitLab Docs: https://docs.gitlab.com/ee/ci/yaml/

Google. (2020). Récupéré sur Developpers Android: https://developer.android.com/

GuilhE, Liad0205, & Kraigolas. (2020, octobre 22). *AAChartCore-Kotlin*. Récupéré sur GitHub: https://github.com/AAChartModel/AAChartCore-Kotlin

Haque, Z. (2017, Novembre 26). *unittest: Python Testing Framework and the Requests library*. Récupéré sur Medium: https://haque-zubair.medium.com/the-power-and-flexibility-of-unittest-paired-with-requests-e3bfaf11ac3

Koehrsen, W. (2017, Décembre 27). *Random Forest in Python*. Récupéré sur Towards Data Science: https://towardsdatascience.com/random-forest-in-python-24d0893d51c0

Lua, D. (2017, Août 19). *Add SupportMapFragment (Android Google Maps) In Fragment*. Récupéré sur Lua Software Code : https://code.luasoftware.com/tutorials/android/supportmapfragment-in-fragment/

Matplotlib development team. (2020, Novembre 12). *Matplotlib: Visualization with Python*. Récupéré sur Matplotlib: https://matplotlib.org/

Muzaffar, A. (2018, Septembre 15). *Android — RecyclerView using MVVM and DataBinding*. Récupéré sur Medium: https://medium.com/@ali.muzaffar/android-recyclerview-using-mvvm-and-databinding-d9c659236908

pandas development team. (2020). *DataFrame*. Récupéré sur Pandas: https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

Paxson, V. (1995, Mars). *Flex, version 2.5*. Récupéré sur GNU: https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html

Prasad, L. (2018, Décembre 3). *Creating LineChart Using MpAndroidChart*. Récupéré sur Medium: https://medium.com/@leelaprasad4648/creating-linechart-using-mpandroidchart-33632324886d

Python Programming/Creating Python Programs. (2020, Juillet 30). Récupéré sur WikiBooks: https://en.wikibooks.org/wiki/Python_Programming/Creating_Python_Programs

Ramos, L. P. (2020). *Python and PyQt: Building a GUI Desktop Calculator*. Récupéré sur RealPython: https://realpython.com/python-pyqt-gui-calculator/

Refsnes Data. (2020). *Python MySQL*. Récupéré sur W3Schools: https://www.w3schools.com/python/python_mysql_getstarted.asp

Requests: HTTP for Humans. (2020). Récupéré sur Read The Docs: https://requests.readthedocs.io/en/master/

Ristić, I. (2018). *Documentation*. Récupéré sur OpenSSL: https://www.openssl.org/docs/

Traefik Labs. (2020). *Concepts*. Récupéré sur Traefik Labs: https://doc.traefik.io/traefik/

Zach, L. (2019, Septembre 8). *Building Python GUIs in 2020*. Récupéré sur Medium: https://medium.com/@lelandzach/building-python-desktop-app-guis-in-2019-macos-windows-linux-19dc71485d60