Université de Montréal

# Rapport de Projet

Par

Yan Zhuang 20146367

Département d'informatique et recherche opérationnelle

Faculté des arts et des sciences

Travail présenté dans le cadre du cours IFT3150

Projet en Informatique

Le 24 avril 2023

## Table des matières

# Introduction

## Introduction de la Maison Mona et l'application mobile Mona

La Maison Mona est un organisme à but non lucratif inclusif et dynamique, dont la mission consiste à « inviter à des rencontres avec l'art, créer un espace commun pour les échanges et faire résonner les sens et les sensibilités de chacun ».

L'application Mona est la façon principale pour atteindre les objectifs de la Maison Mona. Le projet a été initié par Lena Krause en 2015 dans le cadre du cours « Interfaces personne-machine ».

De nos jours, le développement de cette application est déjà une collaboration entre le Département d'histoire de l'art et d'études cinématographiques, et le Département d'informatique et de recherche opérationnelle (DIRO).

L'application mobile Mona donne accès à une base de données qui contient des informations publiques sur les œuvres d'arts, les lieux culturels et les patrimoines au Québec (Ces informations proviennent des autres bases de données publiques). L'application vise à encourager les utilisateurs à aller collectionner ces trois types de « découvert » en prenant des images du découvert correspondantes et à laisser un commentaire. La possibilité d'attribuer un commentaire personnalisé à chaque découvert permet à l'usager de s'exprimer bien son sentiment envers le découvert ciblé. De plus, l'usager peut trouver ces découverts en utilisant la carte dans l'application mobile, et il a aussi un accès à un découvert du jour où nous proposons un découvert différent chaque jour.

Étant donné que les données concernant la rétroaction du public par rapport à l'art est peu documenté de nos jours, l'application enverra les données (les photos et les commentaires) au serveur où se trouve une vue d'ensemble pour faciliter la recherche dans le domaine de l'art basant sur ces informations recueillies. Cela aidera aussi les chercheurs à comprendre comment le public apprécie ces découverts.

## Mon rôle et ma motivation

Mon rôle dans ce projet de MONA est celui de développeur iOS où je vais entretenir et ajouter des nouvelles fonctionnalités dans la version iOS de l'application MONA. La version iOS est développée en Swift, le nouveau langage de programmation créé par Apple en 2014, et leur outil de développement propriétaire Xcode.

Personnellement, je pense toujours que dans le domaine de l'informatique, les expériences de travail et de projets sont vraiment importantes. Ce que l'on peut apprendre en s'impliquant dans un projet ou un travail répond mieux aux besoins réels de notre vie que dans un devoir dans le cadre d'un cours. En conséquence, j'ai bien effectué un stage en été 2022 pendant lequel j'ai appris beaucoup de normes dans l'industrie et les méthodologies de travail. En automne 2022, quand j'ai appris que j'avais la possibilité de pouvoir contribuer au projet de Mona, j'ai envoyé immédiatement un courriel à Lena et Prof. Guy Lapalme en manifestant ma volonté. À part du stage que j'ai effectué, je ne suis jamais impliqué dans un vrai projet qui répond aux besoins réels et qui doit respecter les exigences d'une vraie application (ex : Le respect de la confidentialité de l'usager et l'utilisation de composants/logiciels de source ouvert.)

De plus, j'ai entendu Swift depuis la sortie de ce nouveau langage de programmation évolutionnaire pour iOS en 2014. Surtout, en comparaison avec l'ancien langage de programmation pour iOS – Objective-C, Swift remarque une nouvelle ère du développement mobile. J'ai déjà appris un peu de Swift lors de son premier lancement (bien que les syntaxes s'évoluent dès lors). Toutefois, faute d'idée et de temps pour créer un projet, je ne l'ai jamais eu l'opportunité de coder dans ce langage. Quand j'ai appris de l'application Mona, mon but était vraiment clair – travailler sur l'application mobile iOS de Mona pour que je puisse coder en Swift.

Ce rapport portera principalement sur les tâches que j'ai effectuées en tant que développeur iOS de l'application. Il n'y avait pas un but spécifique au début de ce projet, mais durant notre rencontre de mi-session avec Prof. Lapalme, nous avons tous décidé notre but pour cette application dans le cadre de ce projet.

# Changements effectués pendant mon mandat

## Image du tutoriel et son lancement

À mon arrivée, la version de tutoriel dans l'application était toujours la version que notre designer UI Barbara a conçue en 2021 et 2022. Toutefois, cette année Barbara a conçu une nouvelle version du tutoriel présentant mieux l'utilisation de l'application tout en soulignant la partie en question (et d'autres parties avec un fond flou). Cela était parmi une des premiers changements mineurs que j'ai effectués à l'application iOS de Mona. Plus précisément, j'ai remplacé les images du tutoriel actuelles par celles conçues par Barbara. Grâce à SwiftUI, un langage déclaratif pour concevoir l'interface graphique en iOS (qui remplacement « storyboard ») et aux travaux effectués par les anciens développeurs iOS dans l'équipe, j'ai pu facilement ajouter le bouton « Continuer » qui permet à l'usager de passer à l'image suivante du tutoriel.

De plus, j'ai aussi remarqué que le tutoriel ne serait pas automatiquement lancé pour un nouvel usager et que l'usager devait aller à l'onglet « Autres » pour pouvoir consulter le tutoriel. Cela n'était pas convivial pour l'usager. Par conséquent, j'ai décidé d'ajouter une variable « *firstTimeUser* » qui s'égale à « *True* » après l'inscription pour que le tutoriel puisse s'être lancé. La variable s'égalera à « *False* » après l'exécution du tutoriel pour qu'il ne soit pas lancé à chaque démarrage de l'application.


*Figure 1 Nouveau Tutoriel*

## Découvert du jour

Dans l'ancienne version, le découvert du jour était seulement capable d'afficher les œuvres d'art chaque jour. Même après l'ajout des lieux culturels et les patrimoines dans l'application durant la dernière mise-à-jour, la capacité d'afficher les deux autres types de découvert n'ont pas été ajoutées. Lena m'a donc demandé de le modifier en sorte que l'usager puisse voir des découverts de n'importe quel type chaque jour.

Pour commencer, j'ai bien passé sur les codes qui sont responsables de sélectionner un découvert du jour. En effet, le code est seulement capable de chercher parmi les œuvres d'art qui ne sont pas ciblées ou collectionnées. Grâce à cette structure existante, j'avais déjà une idée de la manière de le modifier pour qu'il fonctionne pour les trois types de découvert. Ma première pensée était de faire ce qui est déjà fait pour les deux autres types de découvert. Toutefois, cette solution s'est avérée très inefficace et a donné lieu à beaucoup de codes répétitifs ayant la seule différence était le type de découvert. En gardant le but de ne pas créer des codes redondants, j'ai décidé d'aller regarder le « DataModel » pour ces trois types de découvert. À ce moment, j'ai trouvé qu'on a créé trois « DataModel » distincts pour ces trois types de découvert même si c'est facile à remarquer qu'il y a bien des attributs en commun. Par conséquent, j'ai décidé d'extraire ces attributs en commun, et créer un nouveau « Protocol » qui s'appelle « MonaModel » auquel ces trois types de découvert doit se conformer en héritant ces attributs en commun ainsi que leurs attributs distincts. Le fait que ces trois « DataModel » se conforment maintenant tous à « MonaModel » me permet bien de pouvoir choisir aléatoirement le découvert du jour sans besoin de tenir en compte leur vrai type. Il faut simplement le convertir vers le bon type lors de l'affichage. Cela réduit un nombre important de codes répétitifs en comparaison avec seulement 3 types de « DataModel » distincts. Même si nous décidons d'ajouter d'autres types de découvert à l'avenir, cela sera plus facile avec moins de changements à faire.

Ensuite, il vient le problème de comment choisir aléatoirement le découvert du jour. Grâce à l'inspiration de Lena, j'ai choisi l'approche de numéro aléatoire. Cette approche consiste à d'abord choisir aléatoirement un nombre de 1 à 15. Si le nombre aléatoire tombe entre 1 et 5, nous allons ensuite choisir aléatoirement, grâce à une fonction existante, d'une œuvre d'art. Si le nombre tombe entre 6 et 10, c'est un lieu culturel qui

sera aléatoirement choisi. Finalement, si le nombre tombe entre 11 et 15, c'est un patrimoine qui sera choisi aléatoirement. Une lettre (« a » pour une œuvre d'art, « p » pour un lieu culturel et « h » pour un patrimoine) sera aussi transmise pour que nous puissions convertir le type de découvert vers le bon type à la fin afin d'extraire les informations dans son intégralité.


*Figure 2 Nouveau "Découvert du Jour"*

Afin d'assurer que l'usager voit le même découvert du jour pendant la journée, il existe aussi un mécanisme de sauvegarder le découvert du jour choisi. Je l'ai modifié pour stocker le nombre aléatoire pour que nous ne choisissions pas un nouveau nombre aléatoire si la date (ce qui est aussi stocké) est le même jour. J'ai dû aussi trouver une façon pour que la date, malgré le fuseau d'horaire réel de l'usager, soit toujours de celui de Toronto/Montréal où se basent la Maison Mona et le serveur.

Pour le côté UI, je l'ai modifié pour qu'il puisse convertir le découvert du jours depuis « MonaModel » vers le bon type grâce au nombre choisi aléatoire. La partie de la carte a dû être modifiée aussi pour qu'elle puisse afficher non seulement les informations en commun entre ces 3 types de découverts mais aussi leurs informations distinctes. J'ai

aussi modifié la couleur du séparateur pour qu'il soit jaune pour une œuvre d'art, mauve pour un lieu culturel et orange pour un patrimoine.

À part de cela, la fonctionnalité de « Découvert du Jour » fonctionne comme avant. Maintenant l'usager pourra voir la recommandation d'une œuvre d'art, d'un lieu culturel ou d'un patrimoine chaque jour quand il lance l'application.

## Connectivité : Inscription et téléchargement de données depuis le serveur

### Introduction

La connectivité est une partie importante pour une application mobile de nos jours. L'implémentation de cette compétence marque aussi une autre phase d'évolution pour l'application Mona. À mon arrivée, l'application Mona est toujours considérée comme « locale ». Bien que l'application utilise l'internet pour téléverser les images, les commentaires et les autres informations pertinentes vers le serveur, les bases de données sont toujours locales et les usagers ne sont pas en mesure de récupérer leur compte s'ils suppriment le compte ou changent leur cellulaire. En effet, un découvert tardif dans le code relève bien que seulement un compte local a été créé en simplement attribuant un nom d'usager à une variable *username* alors que le serveur reçoit des informations fausses (Le code était censé pour le développement, mais il se trouve toujours dans la version de production).

Durant une rencontre hebdomadaire, l'équipe était bien d'accord d'ajouter la fonction d'inscription et de se connecter à l'application Mona pour que l'usager puisse bien s'inscrire pour pouvoir utiliser le compte dans n'importe quel appareil et qu'il puisse récupérer ses données au cas où il changerait son cellulaire. De plus, il faut que l'application commence à télécharger les données (i.e. données sur les œuvres d'art, les lieux culturels, les patrimoines et les badges) directement du serveur au lieu de dépendre des fichiers locaux qui ne pourront pas être mis à jour facilement.

La connectivité représente un nouveau jalon pour l'application Mona. Donc, beaucoup de composants sont compris dans cette partie. À cause de la limite de ce cours de projet, il n'est pas possible pour nous de tout faire avant la fin du cours. Par conséquent, nous discutons ce que nous avons fait, et dans la section « Visions Futurs »,

nous allons parler de ce que nous pourrons encore faire pour la partie de connectivité de l'application Mona.

### Écran de démarrage et d'inscription

Une fois que j'embarquait sur l'ajout de la fonction de la connectivité pour l'application Mona, l'écran de démarrage et d'inscription était la première chose que j'ai commencé à modifier.

Comme mentionné dans l'introduction, l'usager pourrait simplement entrer un nom d'usager aléatoire pour conclure cette partie et tout ce que l'application a fait c'est de stocker le nom d'usager localement. En même temps, le serveur va recevoir un UUID comme nom d'usager, un courriel faux et un mot de passe faux. En effet, cette partie du code n'aurait pas dû être dans la version de production. Cela relève bien que peu de travail a été effectué sur cet aspect de l'application.

L'écran de démarrage affiche d'abord un message de bienvenue pour les nouveaux usagers de Mona. Au lieu de permettre les usagers de cliquer « Suivant » directement, j'ai ajouté une condition : Il faut qu'une connexion d'internet soit disponible. En effet, si l'usager cliquer le bouton « Suivant » sans une connexion internet, une alarme s'affichera en disant à l'usager qu'il faut bien une connexion Internet. Cela a été fait pour que l'usager soit conscient que les étapes suivantes requièrent bien d'internet. De côté code, cette fonctionnalité a été réalisée en utilisant la classe dans Swift, *NWPathMonitor(),* qui est capable de surveiller le statu d'internet. Tout ce qu'il faut est de mettre la procédure de surveillance dans un autre thread pour qu'il surveille le statut d'internet sans interrompre l'UI. Une variable de type « wrapper » *StateObject* sera capable de ré-évaluer l'expression Boolean utilisant cette variable chaque fois le statut d'internet change et décider si l'usager peut passer à l'étape suivante ou afficher une alarme.

Pour l'écran d'inscription, ce n'est qu'un mock-up. La version définitive et finale doit être faite par notre designer UI Barbara. Malheureusement, elle était en vacances pendant deux/trois semaines et que nous devons continuer à travailler. J'ai donc décidé de faire un écran d'inscription temporaire pour pouvoir commencer à travailler sur la logique. Tel que montré, les éléments pour cet écran d'inscription est assez simple. J'ai décidé d'ajouter la possibilité que l'usager puisse voir le mot de passe entré (i.e. l'icône de

l'œil). De plus, les champs de textes seront en rouge si le contenu entré ne respecte pas bien l'exigence demandée. Du côté de code, chaque champ exécutera une vérification après que l'usager aura saisi l'information. Par exemple, le champ de courriel vérifiera bien si ce que l'usager a rentré concerne une adresse courriel valide, et le champ de mot de passe vérifiera si le mot de passe saisi contient bien 6 caractères ou plus (Ce qui est notre politique de mot de passe pour l'instant). Comme mentionné en haut, la décision finale doit être prise par Barbara qui doit bien trouver des façons pour donner des rétroactions claires à l'usager au cas des erreurs (ex : nom d'usager n'est pas disponible).

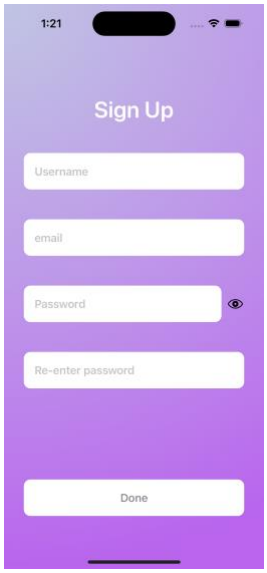


*Figure 3 Écran "mock-up" de l'inscription*

Comme l'écran de démarrage, il faut bien une connexion internet pour pouvoir continuer l'inscription. En utilisant toujours *NWPathMonitor()*, le bouton de « Finir » sera désactivé si l'usager désactive l'internet. Cela empêche l'usager de poursuivre l'inscription sans internet.

Après cette étape vient l'établissement de connexion avec le serveur et l'envoi de serveur. Grâce à des routes API bien établies dans le serveur, il nous faut simplement faire un appel vers le serveur avec les informations que l'usager a saisies. De mon côté, en tenant compte de l'évolution de l'application, j'ai décidé de créer une classe d'erreur personnalisée *RegisterError* qui peut bien aider à afficher des erreurs retournées par le serveur de façon « user-friendly » et « locale » (i.e. en anglais ou en français). Il contient 4 types d'erreurs qui représentent bien les 4 types d'erreurs possibles dans notre cas pour l'instant : 1) *duplicatedUserName* – Un nom d'usage qui existe déjà 2)

*passwordNotStrong* – Un mot de passe qui a moins de 6 caractères 3) *passwordNotMatch* – le mot de passe de confirmation n'est pas identique au mot de passe lors de la première saisie 4) *duplicatedEmail* – L'adresse courriel existe déjà. Pour l'instant, j'ai décidé simplement d'afficher une alarme avec les erreurs reçues. Barbara sera capable de trouver une meilleure façon d'afficher une rétroaction à l'usager. Cela sera donc modifié au besoin.

Grâce à la nouvelle fonctionnalité de *async/await,* c'est facile de créer des appels API asynchrones. Il faut simplement placer un *ProgressView()* dans l'interface pour montrer à l'usager que l'exécution est en cours, en même temps le code va envoyer des appels vers le serveur pour compléter l'inscription. Dès la réception d'une réponse du serveur, nous pouvons retirer le *ProgressView()* et afficher le résultat. Dans le cas où il y aurait des erreurs, nous allons afficher des erreurs reçues en utilisant *RegisterError*. Dans l'autre cas où l'inscription aurait été faite avec succès, nous allons cacher l'écran d'inscription et poursuivre avec l'affichage du tutoriel (En présumant qu'il s'agit d'un premier lancement) qui indique implicitement que l'usage s'est inscrit avec succès.

### Téléchargement de données depuis serveur

Comme mentionné en haut, la version actuelle de Mona dépend toujours des données locales. Cela veut dire que si on fait une mise à jour, il faut que l'usager fasse une mise à jour aussi (et que l'on doive aussi mettre à jour les fichiers dans le bundle de l'application). Cela est hyper inefficace, et ne fournit pas une bonne expérience d'utilisateur. Par conséquent, c'est une tâche importante de permettre à l'application de télécharger les données directement du serveur.

La première étape c'est de modifier les codes dans « XXXDataService » (Ex : PatrimoinesDataService.swift) où les données sont obtenues actuellement depuis les fichiers locaux. Tout comme l'inscription, j'ai bien profité du patron async/await pour pouvoir faire un appel API vers le serveur. La stratégie que j'ai choisie est de télécharger les données et les stocker dans un fichier local pour que le téléchargement ne se fasse qu'une seule fois (à part de la mise-à-jour de données), et que l'application puisse lire ce fichier pour charger les données.

La deuxième étape, qui est vraiment importante et qui est un peu difficile, c'est d'intégrer le téléchargement de données et l'inscription avec UI. Surtout vu qu'il s'agit bien des appels asynchrones, il faut bien pouvoir gérer cette situation. Il nous faut aussi bien tenir en compte des erreurs qui peuvent se passer lors de l'inscription et du téléchargement. J'ai décidé que le téléchargement serait fait après l'inscription (Sans intervention de côté usager). S'il y a des

erreurs d'inscription, celles liées à l'inscription seront affichées sous forme d'alerte et le téléchargement ne finira pas. L'usager aura l'opportunité de s'inscrire de nouveau. Cependant, si l'inscription a été faite avec succès, le téléchargement de données débutera tout de suite. S'il y a des erreurs du téléchargement, les erreurs seront affichées et l'usager peut réessayer de nouveau. Étant donné que l'inscription aura déjà faite à ce moment-là, l'usager ne pourra plus changer les informations saisies et que le processus d'inscription ne se passera pas de nouveau.

Pour ce faire, j'ai bien profité du patron « publisher/subscriber » du côté de la logique et « ProgressView » de côté de l'UI. Avec le patron « publisher/subscriber », l'UI peut être informé si le téléchargement ou l'inscription est finie ou non, avec des erreurs ou non. Avec « ProgressView » de SwiftUI, l'usager peut voir qu'il y a un processus en cours et il va bloquer l'interface pour que l'usager ne puisse rien appuyer durant le processus d'inscription et de téléchargement. J'ai dû passer beaucoup de temps là-dessus pour que le mécanisme fonctionne bien et que les erreurs soient bien gérées comme décrit en haut.


*Figure 4 Téléchargement de données*

J'ai aussi rencontré un problème avec « BadgesDataModel ». Quand la version d'API passe de V2 à V3, le format de JSON pour les Badges ont changé aussi. Cela n'était pas un problème pour les autres objets. Toutefois, avec les « BadgesDataModel », un décodeur a été créé et que beaucoup d'autre classes dépend de ce modèle de donnée. Donc, sans vouloir perturber le système stable, j'ai dû passer du temps pour trouver la correspondance entre les attributs en V2 et ceux en V3, et réécrire un nouveau décodeur pour les Badges.

J'ai dû aussi changer ce qui est appelé lors du démarrage de l'application et après la fin de l'inscription et du téléchargement de données. Auparavant, les données étaient initialisées tout de suite lors du démarrage des données. Les pins et les items de la liste l'étaient aussi. Toutefois, vu que les données ne seront pas disponibles avant la fin du processus, j'ai dû déplacer ces fonctions pour après l'inscription et de téléchargement.

Cette fonctionnalité a été difficile à compléter parce qu'il faut tout refaire sans perturber le mécanisme existant dans le système. Heureusement, tout est fait avec succès. Tout comme l'inscription, il y a toujours des décisions créatives à prendre par Barbara, ce qui peut potentiellement influencer le code.

## Vision Futures

Il y a toujours et encore des changements à apporter à l'application Mona pour la rendre plus efficace et lui fournir plus de fonctionnalités. Néanmoins, vu que ce projet a été fait dans le cadre du cours de projet, il y a une date limite et qu'il n'est pas possible pour nous de faire tout ce que nous voulons pendant ces 4 mois. Par conséquent, nous allons parler de ce que nous pouvons encore faire dans le futur pour l'application Mona.

Pour des objectifs à court ou moyen terme, la connectivité sera notre concentration. Comme mentionné dans la partie de la connectivité, cette partie est importante pour l'évolution de Mona. Par exemple :

1) Possibilité de se déconnecter et de se connecter : Maintenant l'usager peut seulement s'inscrire mais sans la possibilité de se déconnecter ou de se connecter dans un autre appareil
    a. Pour bien créer cette fonction, il faut bien trouver une façon pour pouvoir effacer les contenus locaux dans l'application une fois déconnecté, et de télécharger les données d'usager (ex : Photo prises, commentaires, badges collectionnés) depuis le serveur une fois connecté
    b. Il y a d'autres fonctionnalités standards qui doivent être ajoutées. Notamment, la possibilité de récupérer le mot de passe en cas de perte.
2) Permettre l'utilisation de mode hors connexion : L'usager peut bien collectionner les découverts même quand il n'y pas de connexion internet. Lors de la reprise de la connexion d'internet, les opérations effectuées par l'usager seront envoyées vers le serveur.

3) Permettre à l'usager de choisir si la connexion mobile est autorisée ou non.

Il y a aussi d'autres changements à apport à Mona, qui ne sont pas liés avec la connectivité, par exemple :

1) Rendre les commentaires et les photos modifiables
2) Pouvoir choisir les photos depuis l'album de photo directement
3) Amélioration de filtre : Plus adapté aux lieux culturels et les patrimoines (Ex : Utiliser des mots différents, pas de filtrage sur type d'art quand il s'agit d'un patrimoine)

## Test Interne

Le 12 avril, l'équipe de Mona a décidé de tenir une séance de test interne pour que nous puissions tester si l'usager puisse utiliser l'application de façon intuitive et s'il y a des problèmes que nous n'avions pas trouvés pendant notre développement de l'application.

Mon application iOS a été testée par trois utilisateurs. En effet, il s'est avéré que le test interne était vraiment utile. Un des trois utilisateurs a trouvé que l'application pouvait planter parfois dès le démarrage. Grâce à « crash log » transmis automatiquement par Apple, j'ai pu repérer directement le problème. Ce problème a causé par le mécanisme de « Découvert du Jour ». Comme mentionné en haut, le découvert du jour va choisir un nombre aléatoire pour déterminer le type de découvert. J'ai aussi une fonction qui va renvoyer explicitement le type de découvert pour que le SwiftUI puisse le convertir vers le bon type. Toutefois, à cause de la frappe d'erreur, l'intervalle de nombre pour chaque type de découvert dans ces deux fonctions n'est pas pareil. Cela cause qu'un découvert peut être converti vers un mauvais type. Cela entraîne directement un crash parce que l'application ne sera pas capable de gérer une telle erreur fatale. Heureusement, ce problème est facile à régler, grâce à ce test interne.

Il y a un autre problème qui a été relevé par ce test interne : Les données de badges ne seront pas chargées sans redémarrer l'application. Après avoir analysé les fichiers et le mécanisme actuel pour les badges, j'ai trouvé que la fonction de chargement de données de badges se passe une seule fois, pour chaque démarrage. Pour la toute première fois, la fonction aura été déjà appelée avant le téléchargement des données de badges. La

solution pour l'instant c'est d'appeler la fonction de nouveau après le téléchargement une seule fois pour que les badges soient chargés. Cela a marché sans problème.

D'autre problèmes mineurs ont été mentionnés par les testeurs, y compris la couleur de texte pour l'écran d'inscription, les liens non-cliquables dans l'écran « À propos ». Ces problèmes ont été résolus. Les autres problèmes dont nous avons connaissance seront laissés à la prochaine itération (ex. : Fonctionnement de filtre)

## Conclusion

Pour conclure, j'ai bien aimé mon expérience dans notre équipe Mona. Depuis le début, Lena est vraiment gentille avec tout le monde. Cela crée un environnement de travail vraiment agréable. Bien sûr, Natacha et Gaspards sont vraiment sympas aussi ce qui facilite bien les travaux surtout nos travaux sont pas mal liés à l'un à l'autre.

Les réunions hebdomadaires nous aident vraiment à se communiquer. Chaque fois, grâce à l'environnement relaxant, nous avons pu discuter beaucoup de choses, soit liés au projet soit liés à notre vie au Canada (Comme nous quatre sont tous immigrants au Canada). Je ne peux pas penser à une réunion hebdomadaire où je n'ai pas trouvé des nouveaux problèmes avec l'application. Donc, je trouve que ces réunions rendent nos travails plus efficaces en nous incitant à réfléchir plus.

En travaillant sur ce projet, j'ai pu améliorer mes connaissances par rapport à Swift et de développer un logiciel qui doit prendre en compte des besoins réelles. Aussi, vu que j'ai travaillé exclusivement avec Xcode. Je suis maintenant plus à l'aise avec l'outil Xcode. J'ai aussi beaucoup appris des différents aspects dans Swift : 1) Comment lancer des appels API avec la fonctionnalité de *await/async* 2) Création d'interface avec SwiftUI 3) Exécution de tâche avec multi-thread.

Heureusement, Lena m'a bien proposé un court contrat de 40 heures. Cela me fait bien plaisir de pouvoir continuer à travailler sur ce projet, même sous un contrat court. Je vais commencer à implémenter ce que j'ai mentionné dans « Visions Futurs ». Bien sûr, tout d'abord, il faut bien prioriser les fonctionnalités à implémenter.