TP2 – Application Android pour organisation de rencontres

INF8405 – Informatique Mobile

Hiver 2016

Par

Clément Gamache - 1642792

Cédric Noiseux - 1588195

Soumis à

Aurel Josias Randolph

Polytechnique de Montréal

23 Mars 2016

Introduction

Le but de ce travail pratique était de concevoir une application mobile permettant de faciliter l'organisation et la gestion de rencontres pour un groupe de travail. Cette application prend en compte les préférences, la position géographique et le calendrier de chaque utilisateur pour proposer des horaires et des lieux de rencontres optimales.

L'application doit répondre à plusieurs exigences fonctionnelles. Chaque utilisateur doit créer son profil (photo, groupe, courriel, organisateur, préférences) avant de pouvoir utiliser l'application. L'application doit pouvoir accéder au calendrier et au service de localisation des appareils mobiles pour pouvoir afficher la position de chaque membre du groupe sur une carte (Google Map). Elle doit aussi pouvoir recommander des options de rendez-vous aux membres selon leurs profils, leurs positions et leurs préférences. Une fois l'option décidée à l'aide d'un vote, l'organisateur du groupe doit pouvoir organiser la rencontre en y incluant un bref descriptif (photo, description, coordonnées) et en le partageant aux autres membres.

Présentation Technique

L'application mobile *Organisapp* a été construite avec 6 activités et 5 classes Java, respectivement : *Menu*, *Profile*, *MapActivity, CalendarActivity*, *LocationActivity*, *Utility*, *MultiSelectionSpinner*, *ActivityType, Event*, *Group*, et *User*. Ces onze composantes seront présentées individuellement plus en détail.

Menu

L'activité *Menu* affiche les différentes options que peut choisir l'utilisateur pour naviguer dans l'application. Il y a cinq boutons qui permettent d'accéder aux autres activités mentionnées précédemment : *Profile, Calendar, Map, Meeting* et *Recommend.* Chaque bouton appelle sa propre méthode.

La méthode *goToProfile* permet de démarrer l'activité *Profile*. La méthode *goToCalendar* permet de démarrer l'activité *CalendarActivity*. La méthode *goToMap* permet de démarrer l'activité *CalendarActivity*. La méthode *goToRecommend* permet de démarrer l'activité *Utility*. La méthode *goToLocation* permet de démarrer l'activité *LocationActivity*.

Profile

L'activité *Profile* affiche le profil du propriétaire de l'appareil en utilisation. En effet, lorsque l'utilisateur complète son profil, ses informations sont enregistrées sur son appareil mobile. Cinq champs sont modifiables : la photo, le nom du groupe, l'adresse courriel, l'organisateur et les préférences. Ils permettent de créer le profil utilisateur. Deux boutons sont présents, *Menu* et *Save*, ils permettent respectivement de revenir au menu principal et d'enregistrer les informations entrées.

La méthode *requestPermissions* permet de vérifier les permissions nécessaires au bon fonctionnement de l'application soit : l'accès à la localisation, au stockage interne de l'appareil mobile, au calendrier, à Internet et aux comptes utilisateurs. Dans le cas où la permission est requise, un message de requête apparait à l'écran et la méthode *onRequestPermissionsResult* est appelé pour afficher le bon message de confirmation.

La méthode *browsePicture* permet d'accéder à la gallerie photo de l'utilisateur et la méthode *onActivityResult* permet la sélection de la photo et son affichage à l'écran. Il est important de noter que le chemin vers l'image est stocké dans la variable *Path* pour usage futur.

La méthode *saveClicked* permet d'enregistrer le profil utilisateur dans son appareil. Le fichier de sauvegarde est d'abord créer s'il n'existe pas. S'il existe déjà, il est lu pour soutirer le chemin de la photo (*Path*), la position précédente (*Position*) et le calendrier lors de la dernière synchronisation (*Calendar*). Le fichier est ensuite mis à jour avec les informations présentement entrées dans les différents champs (*Path, Group, Email, Organizer, Preferences*) et avec la plus récente valeur de position et calendrier (*Position, Calendar*).

La méthode *goToMenu* contient deux scénarios principaux : naviguer vers l'activité *Menu* ou *MapActivity*. Si le fichier de sauvegarde est complet et possède une position et un calendrier non nul, la méthode démarre l'acitivité *Menu*. En général, ce scénario est appelé si l'utilisateur provient précédemment du menu. Le deuxième scénario est appelé si le fichier de sauvegarde est complet mais la position ou le calendrier est nul. L'activité *MapActivity* est ainsi appelé pour définir la position de l'utilisateur et l'activité *CalendarActivity* ensuite pour obtenir le calendrier. En général, ce scénario est appelé lorsque l'utilisateur démarre l'application, de manière à mettre à jour les informations de positionnement et de calendrier. Le troisième scénario afficher un message demandant de compléter le profil, ce qui veut dire que les cinq champs n'ont pas été complétés.

La méthode *getLines* permet de compter le nombre de lignes non nulles dans le fichier de sauvegarde.

La méthode *checkProfile* permet de vérifier la complétude du fichier de sauvegarde lors de l'ouverture de l'application. Si le fichier est incomplet, un message demande à l'utilisateur de compléter son profil. Si le fichier est complet, la prochaine méthode *readFileInEditor* est appelé et *goToMenu* ensuite si l'application vient d'être démarré.

La méthode *readFileInEditor* est généralement appelée au démarrage de l'activité et permet de remplir les cinq champs avec les informations de l'utilisateur. Elle permet aussi de donner les bonnes valeurs aux variables *Position* et *Calendar*.

MapActivity

L'activité *MapActivity* permet d'afficher la position de chaque membre d'un groupe sur une carte de type Google Maps, et ce, en temps réel. Chaque membre est identifié avec un marqueur personnel sur la carte grâce à leur adresse courriel.

La méthode *loadMap* permet de démarrer la GoogleMap supporté par le MapFragment. Cette opération est réalisée si le MapFragment et la GoogleMap ne sont pas nuls.

La méthode *getMyLocation* permet de fixer la position de l'utilisateur, de créer le client Google API et de le connecter avec la méthode *connectClient*. Cette méthode remplie son rôle seulement si le client n'est pas nul et si les services Google Play sont accessibles. Si, par exemple, le client Google API est incapable de se connecter, la méthode *onActivityResult* essaye de le connecter à nouveau, mais seulement si l'activité n'a pas subi d'erreurs lors de son démarrage.

La méthode *isGooglePlayServicesAvailable* vérifie si les services Google Play sont accessibles. Si c'est le cas, un message de confirmation est écrit dans le log, sinon, un dialogue d'erreur est affiché à l'écran.

La méthode *onConnected* est appelée lorsque le client Google API est connecté avec succès et elle permet de calculer la position initiale de l'utilisateur si le GPS est activé. S'il est activé, la longitude et la latitude sont calculées. Si l'application vient de démarrer, la position est mise à jour dans le fichier de sauvegarde de l'utilisateur par rapport à sa position lors de la dernière utilisation. Puisque l'application vient de redémarrer, une fenêtre demande ensuite d'accéder au calendrier de l'utilisateur pour mettre à jour ses

événements futurs. Si l'utilisateur provient du menu, la position n'est pas mise à jour dans le fichier de sauvegarde, des marqueurs initiaux sont plutôt affichés pour chaque membre du groupe.

Finalement, la méthode *startLocationUpdates* est appelée peu importe le scénario. Cette méthode permet de créer un nouvel objet de type *LocationRequest* qui permet d'envoyer des requêtes de localisation à intervalles fixes.

La méthode *onLocationChanged* est appelée à chaque fois que l'objet *LocationRequest* envoie une requête de localisation avec succès. Elle agit de manière similaire à *onConnected*. La longitude et la latitude sont calculées et enregistrées dans le fichier de sauvegarde de l'utilisateur. Les marqueurs pour chaque membre du groupe sont ensuite affichés sur la carte.

Les méthodes *onConnectionSuspended* et *onConnectionFailed* permettent de gérer les erreurs de connections avec le client de localisation par l'entremise des services de localisation.
ErrorDialogFragment

CalendarActivity

L'activité *CalendarActivity* permet d'afficher à l'écran les dix prochains événements de l'utilisateur et de les sauvegarder dans son fichier de sauvegarde. Le but principal de cette activité est d'accéder au calendrier Google des utilisateurs. Un bouton *Menu* permet également de retourner au menu principal avec une méthode assez standard *goToMenu*.

La méthode onActivityResult permet de gérer les résultats de plusieurs activités. Lorsqu'une requête pour connaitre l'accessibilité des services Google Play est envoyé, la méthode *isGooglePlayServicesAvailable* est appelée. Elle agit de manière très semblable à celle présente dans *MapActivity*. Lorsqu'une requête pour sélectionner un compte Google est envoyée, les préférences sont enregistrées de même que le compte principal sélectionné pour les usages futurs. Lorsqu'une requête pour autoriser l'accès à un compte Google est envoyée, une fenêtre présentant les comptes accessibles est affichée à l'écran. Cette fenêtre est affichée en appelant la méthode *chooseAccount*.

La méthode *refreshResults* permet d'accéder à un set de données du calendrier Google pour qu'il soit affiché à l'écran. Si aucun compte Google n'est sélectionné, une demande est envoyée avec la méthode *chooseAccount*. Si un compte est sélectionné et que l'appareil est connecté à Internet, une requête est

envoyé pour accéder aux données du calendrier, un message d'erreur est affiché sinon. La méthode *isDeviceOnline* permet de vérifier la connectivité de l'appareil au réseau.

La classe interne *MakeRequestTask* est importante car elle permet de réaliser la tâche asynchrone qui gère l'appel au Google Calendar API. Son constructeur permet l'initialisation et la construction du Google Calendar API. La méthode *doInBackground* permet de retourner les données retirées du calendrier. La méthode *getDataFromApi* permet de lister les dix prochains événements du calendrier Google de l'utilisateur. La méthode *onPreExecute* permet d'initialiser des variables avant l'utilisation des services. La méthode *onPostExecute* permet d'afficher à l'écran les événements du calendrier, si possible, et les enregistre dans le fichier de sauvegarde de l'utilisateur. Si l'application vient de démarrer, l'utilisateur est immédiatement dirigé vers le menu. La méthode *onCancelled* permet de gérer l'annulation de l'appel asynchrone au Google Calendar API et les erreurs.

LocationActivity

GetData

Cette classe instancie un processus qui permet l'accès à la base de données de l'application par le biais d'une simple requête *http*. Nous avons utilisé le paquet *okhttp* qui permet de générer ce type de requête en un minimum de lignes de code. Le processus prend en entrée le nom du fichier se trouvant dans le répertoir *tp2* du serveur et la requête est lancée. Lorsque terminée, elle retourne le contenu du fichier demandé.

SetData

Cette classe instancie un processus qui permet l'écriture entière du contenu d'un fichier sur le serveur. Pour ce faire, elle instancie une requête *http* qui appelle un script *php* qui ajuste le texte du fichier en question à la valeur donnée.

Utility

La classe Utility est une classe utilitaire ayant comme fonction d'appeler les deux processus présentés précédemment. Ces deux processus doivent absolument être asynchrones à l'application principale parce qu'ils fonctionnent avec les outils de réseaux du téléphone. La version actuelle d'Android ne permet pas qu'une application ne fasse rien en attendant des tâches de réseautique, alors nous avons dû les instancier

de cette manière. Cependant, dans notre cas, lorsque ces processus sont appelés, nous voulons recevoir une réponse le plus vite possible. Nous devons appeler la méthode *get* de ces processus pour attendre qu'ils soient terminés pour acquérir leur valeur de retour. Un appel de fonction doit dans les principaux cas consister en une seule instruction si elle est appelée plusieurs fois, donc nous avons créé la classe *Utility* pour englober ces fonctions en une seule.

MultiSelectionSpinner

La classe *MultiSelectionSpinner* permet d'afficher une sélection de préférences sous forme de *CheckBox* à l'intérieur d'un *Spinner*. Les préférences sélectionnées sont affichées à l'écran en ordre de sélection. Le constructeur permet de créer un nouveau *ArrayAdapter*.

La méthode *onClick* permet de confirmer la sélection d'une préférence et l'ajoute à la liste à l'écran. La méthode *performClick* permet d'afficher à l'écran le *Spinner* contenant la liste de préférences et leur *checkBox*. Les méthodes *setItems* permettent de spécifier les différents choix de préférences offerts à l'utilisateur. Les méthodes *setSelection* permettent de cocher ou non les préférences qui ont été sélectionnées précédemment lorsque l'utilisateur ouvre le *Spinner*. La méthode *getSelectedStrings* permet de retourner les éléments sélectionnés par l'utilisateur en *List<String>*. La méthode *getSelectedIndicies* permet de retourner les éléments sélectionnés par l'utilisateur en *List<Integer>*. La méthode *buildSelectedItemString* permet de retourner les éléments sélectionnés par l'utilisateur en *String*.

ActivityType

Cette classe instancie un type d'activité à laquelle un groupe peut participer. Elle est très simple et ne contient qu'un nom (*String*) comme attribut. Cependant, pour conserver la consistance de notre code et pour pouvoir utiliser l'opérateur « == » sur ces objets, nous avons dû les instancier sous forme de classe. De plus, le fait qu'elle soit une classe a permis de déterminer un emplacement où rédiger ses méthodes spécifiques telles que les méthodes d'accès et d'écriture dans la base de données de l'application.

Group

Cette classe instancie un groupe auquel les utilisateurs peuvent faire partie. Tout comme *ActivityType*, elle n'a comme attribut qu'un nom et son environnement sert à des méthodes spécifiques.

User

Cette classe instancie toutes les propriétés de l'utilisateur, soient son adresse email, le groupe duquel il fait partie, son rôle au sein du groupe, ses intérêts et sa position actuelle. Tout comme les deux classes précédentes, toutes les instances de cette classe sont enregistrées sur le serveur de l'application.

Difficultés Rencontrées

Plusieurs difficultés ont été rencontrées lors de la conception de cette application. Premièrement, trouver le moyen de stocker et de lire les données utilisateurs tant sur son appareil que sur le serveur externe.

Deuxièmement, trouver le moyen de gérer les nombreuses permissions a causé problème également. L'API 23 d'Android requiert que les permissions soient demandées explicitement à l'utilisateur et cet aspect a été inconnu pendant une bonne partie de la conception. Ce problème a impacté le développement d'une fraction importante de l'application puisque des permissions sont requises pour accéder au stockage de l'utilisateur, à ses comptes, à sa galerie photo et à sa localisation.

Troisièmement, faire fonctionner l'activité Google Map a été difficile considérant notre inexpérience avec les API Android (API Google Maps, API Google Calendar, etc). Des problèmes avec les services de localisation ont aussi rendu la tâche plus difficile. Toutefois, une fois cette activité complétée, il a été plus simple de concevoir l'activité Calendar puisque la configuration de cet autre API s'est fait sensiblement de la même façon.

Quatrièmement, il a été difficile de configurer correctement Android Studio et l'émulateur pour démarrer l'application sans problème. Différents outils SDK ont dû être installés et différents émulateurs testés avant de pouvoir utiliser l'application. De plus, les fichiers *build.gradle* et AndroidManifest ont dû être modifiés. En somme, se familiariser avec les différents API a été aisément la difficulté principale rencontrée.

Critiques et Suggestions

Quelques décisions auraient pu être faites différemment au niveau de l'implémentation et l'exécution. D'abord, trouver un moyen d'accéder à la localisation et au calendrier d'un utilisateur sans avoir à démarrer les activités *MapActivity* et *CalendarActivity.* À chaque fois que l'utilisateur démarre l'application, ces deux activités sont accédées avant de se diriger vers le menu. Ceci peut déranger un

utilisateur et augmente le temps de démarrage. Il serait plus intéressant de pouvoir accéder à intervalle régulier au calendrier et à la position de l'utilisateur sans avoir à accéder à ces deux activités.

Ensuite, la manière de lire et écrire dans le fichier de sauvegarde aurait pu être plus efficace. Le fichier est ouvert, lu et réécrit à chaque fois que des modifications doivent être faites dans celui-ci. D'avantage de recherche serait nécessaire pour pouvoir simplement éditer le fichier directement sans avoir à réaliser ces opérations de lectures/écritures supplémentaires.

L'obligation d'avoir un compte Google pour accéder au calendrier peut aussi être un irritant pour les utilisateurs. L'option d'accéder au calendrier de l'appareil directement devrait être proposée également. Toutefois, il était plus simple de centraliser les opérations lors de la conception et d'utiliser les API fournis par Google. Pouvoir accéder à plus que les dix prochains événements d'un utilisateur serait aussi intéressant pour fournir des options de rencontre optimales. Finalement, les fonctionnalités de l'application sont remplies mais l'efficacité pourrait être améliorée avec d'avantage de travail d'optimisation.

Une autre imperfection de notre projet est le mode de sauvegarde des données de l'application. En effet, la façon dont nous avons implémenté cette dernière est par des fichiers enregistrés sur un serveur public. Ces fichiers sont des fichiers texte qui respectent une nomenclature propre à notre application. Cette pratique est très mauvaise. Elle rend publique l'accès à des données personnelles des utilisateurs de l'application. De plus, notre application réécrit le contenu de chaque fichier texte chaque fois qu'elle écrit dedans. Cette pratique est faite à partir de fichiers *php*, eux aussi enregistrés sur notre serveur. L'application appelle ces fichiers en leur spécifiant le nom du fichier à modifier, ainsi que son nouveau contenu. Cette pratique est la moins sécuritaire possible. En effet, n'importe qui sachant légèrement le *php* pourrait envoyer des requêtes à notre application, faisant en sorte que les fichiers de notre application puissent se perdre en un clin d'œil.