Faculté des Sciences
Université de Montpellier

Projet Android
E-Swipe

Auteurs :
Anthony Carmona
Fati Chen
Jean-François Ramos

31 mai 2017

Table des matières

Table des figures

1 Presentation

L'application E-Swipe est une application de rencontre intelligente. Elle est inspiré de l'application de rencontre Tinder, à la différence que E-Swipe permet de réaliser des rencontres entre les utilisateurs en fonction des évènements proche de chez eux.

Disponible à partir de l'Api 19 (Kitkat), cette application présente plusieurs fonctionnalités qui vont être présentées dans ce rapport comme :

- La connexion d'un utilisateur
- L'édition de profil d'un utilisateur
- La participation d'un utilisateur, ou non, à un évènement
- La possibilité de *swipe* des profils d'utilisateurs pour pouvoir rencontrer d'autres personnes utilisant l'application.
- L'interaction social entre 2 utilisateurs qui *matchent* après s'être acceptés via la fonctionnalité Swipe. Ces derniers pourront communiquer via un chat intégré à l'application. Ce chat leur sera exclusivement réservé. (Voir le Glossaire pour les termes en italique)

2 Fonctionnalités

2.1 Connexion

Dans le but de rendre sociale notre application, nous permettons à l'utilisateur de se connecter via Facebook ou par une identification classique email/mot de passe. La connexion via facebook est disponible à partir de l'API 15 : Android 4.0.3 (Ice Cream Sandwich).

FIGURE 1 – Activité de connexion

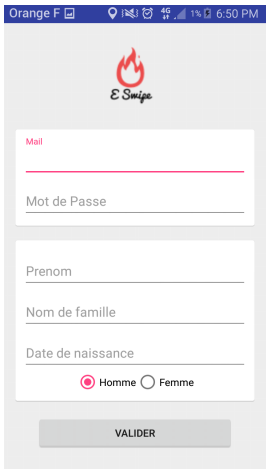

FIGURE 2 – Activité pour l'inscription

Si l'utilisateur n'as pas de compte Facebook, il peut se créer un compte à l'aide de la page d'inscription. En appuyant sur le bouton Register. Dès que l'utilisateur s'est identifié par Facebook, s'est inscrit, ou s'est connecté. Il est redirigé vers sa page de profil.

2.2 Profil Utilisateur

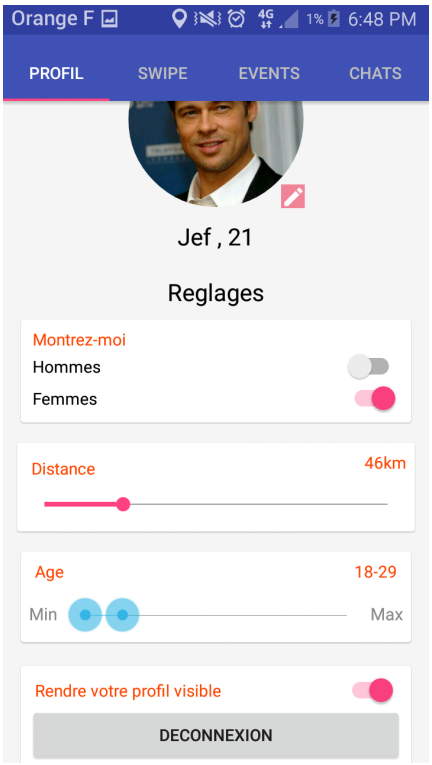

FIGURE 3 – Profil de l'utilisateur

L'utilisateur s'est connecté, il a accès aux réglages suivants :

- Les personnes a recherché avec des paramètres tel que :
  - Le sexe des utilisateurs a recherché (Homme, Femme ou les 2)
  - L'intervalle d'âge des utilisateurs a recherché
  - Le rayon de recherche en kilomètre autour de sa position
- La possibilité de rendre son profil visible ou non des autres utilisateurs
- Déconnecter son compte de l'application.

A partir de cette activité, l'utilisateur peut se rediriger vers les autres fonctionnalités de l'application ou il peut :

- modifier l'apparence de son profil en appuyant sur le logo rouge (Voir 4). Cette activité va permettre à l'utilisateur de modifier les photos de son profil. On peut également modifier son sexe, ainsi que sa description de profil. (Voir
- visualiser l'apparence de son profil en appuyant sur sa photo principale (Voir 5). Cette activité va permettre à l'utilisateur de voir comment est organisé son profil avec ses photos, sa description. Cette activité permet également à l'utilisateur de voir l'ensemble des évènements auxquels il a signifié qu'il participe.

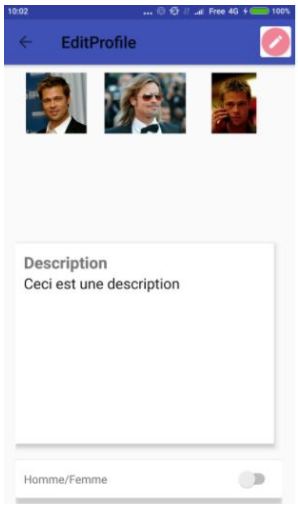

FIGURE 4 – Edition du profil

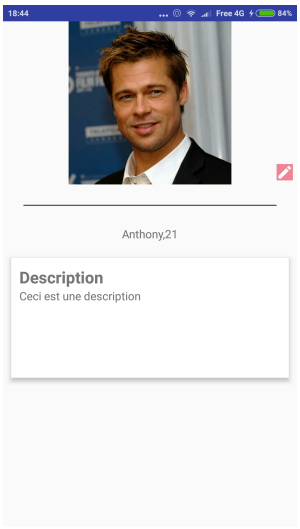

FIGURE 5 – Partie publique du profil d'un utilisateur

2.3 Social

2.3.1 Evenement

Un utilisateur peut voir l'ensemble des évènements disponible autour d'un certain périmètre, définit au préalablement dans les réglages de l'utilisateur. En appuyant sur l'un d'entre eux, il a accès au profil d'un d'évènement.

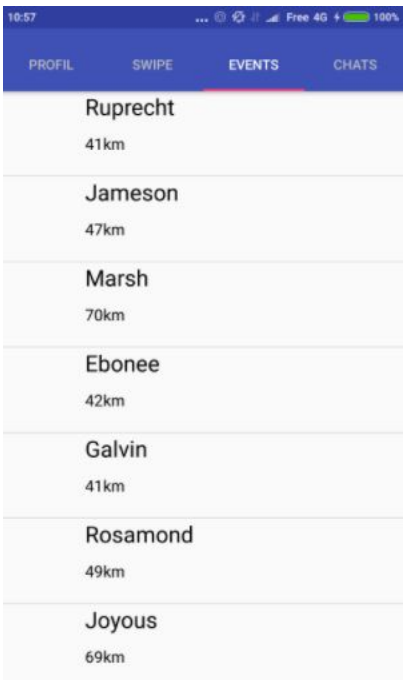

FIGURE 6 – Liste des évènements

Sur le profil d'un évènement, un utilisateur peut savoir la date et l'heure de l'évènement, sa position à l'aide de l'application Google Maps ainsi que le nombre de participants à cet évènement. Via cette activité, l'utilisateur peut accepter de participer à cet évènement, ou de ne plus y participer si celui-ci avait déjà notifié qu'il participait à cet évènement. Bien que cette activité a été créé dans le code, nous n'avons pas eu le temps de lier le contenu d'un évènement à la liste des évènements

2.3.2 Swipe

L'utilisateur peut swipe des profils vers la droite s'il apprécie ce profil ou vers la gauche s'il n'apprécie pas ce profil. Il peut également appuyer sur les boutons en dessous des profils pour accepter ou refuser les profils. S'il y a match entre 2 utilisateurs, c'est-à-dire qu'ils se sont accepté (swipe vers la droite) mutuellement. Alors un chat est créé entre les 2 utilisateurs pour que ces derniers puissent communiquer.

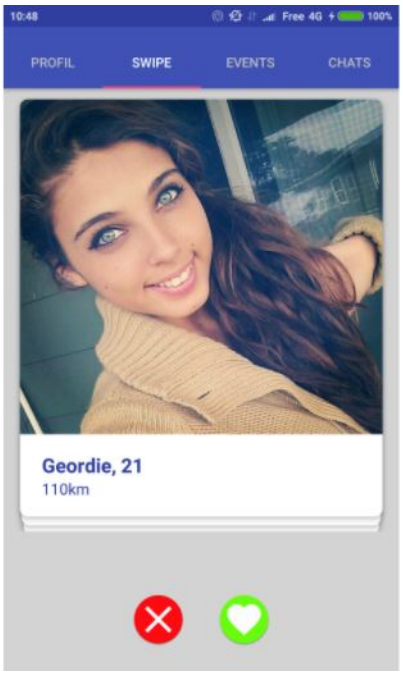

FIGURE 7 – Le swipe

2.3.3 Chat

L'utilisateur peut communiquer avec un autre utilisateur via un chat lorsque 2 utilisateurs se sont acceptés. Un utilisateur a accès à la liste de ses chats, c'est-à-dire à la liste des utilisateurs qu'il a accepté et que la réciproque est vrai.

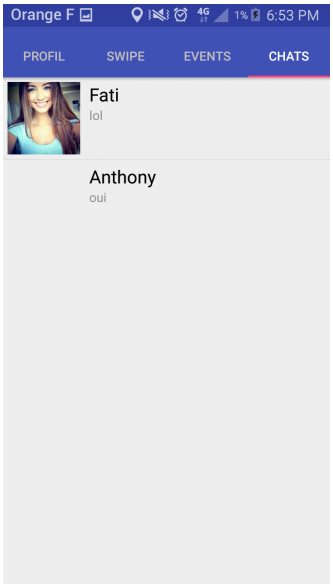

FIGURE 8 – ListChat

FIGURE 9 – Chat entre 2 utilisateurs

Un item de cette liste contient un chat avec un utilisateur, en appuyant dessus, l'utilisateur accède à ce chat et peut communiquer avec son vis-à-vis.

2.3.4 Mise en place d'une application internationale

Toutes les chaînes de caractère de l'application sont externalisées dans des fichiers XML. Actuellement, il existe un fichier ou les chaînes de caractères sont en anglais (fichier appelé par défaut) ce fichier est dans un package spécifique *res/values*. Si l'on souhaite ajouter une nouvelle langue à notre application, il suffit de créer un nouveau package *res/values_langue* puis un fichier XML pour cette langue et de reporter tous les identifiants et d'effectuer la traduction de chaque chaîne de caractère.(Voir un extrait de string.xml)

Listing 1 – string.xml

```
1 <resources>
2     <string name="app_name">E-swipe</string>
3     <string name="title_activity_login">Sign in</string>
4
5     <!-- Strings related to login -->
6     <string name="prompt_email">Email</string>
7     <string name="prompt_password">Password</string>
```

```
8       <string name="action_sign_in">Sign in</string>
9       <string name="action_register">Register</string>
10  </resources>
```

3 Rapport Technique

3.1 API REST

3.1.1 Présentation

Une API[1] REST permet de définir plusieurs contraintes inhérentes aux données nécessaires à l'application et à l'hôte. Celles-ci sont définies par l'utilisateur et doivent respecter plusieurs conditions.

Nous implémenterons une API de ce type car elle réponds le plus à nos attentes : besoin d'un protocole asynchrone, où les deux partis sont indépendants et autonomes. Le fonctionnement interne du serveur est aussi masqué, cela permet une plus grande liberté au niveau du choix du serveur et de leur quantité.

La base du REST est de permettre aux applications de communiquer entre elles, par conséquent, il est nécessaires de mettre en place une série de directives. Toutes les interactions se faisant au moyen du protocole HTTP doivent être composés de :

Methodes : ils définissent le comportement effectué par le serveur. Les méthodes les plus courantes sont :
- `GET` : utilisé pour récupérer une ressource
- `POST` : pour ajouter une ressource ou à effectuer une action
- `PUT` : pour mettre à jour/remplacer la ressource
- `PATCH` : pour mettre à jour seulement les informations modifiées (delta-diff[2])
- `DELETE` : permet de supprimer la ressource

URI (chemins) : ils définissent le chemin vers la ressource. Une certaine convention est imposée par REST : le chemin doit être explicite,et les action du CRUD doivent y être contenus. Par exemple, pour ajouter un livre on utilisera `POST /books/`, par conséquent, on peux en déduire que `GET /books/` permet de récupérer un livre, que `PUT /books/:id` permet de le modifier.... Malgré tout, ceci reste très discutable et dépendant de l'implémentation : si l'on utilise un ensemble de livres (books) alors il sera alors nécessaire de redéfinir les requêtes précédentes à un chemin au singulier (book).
De plus, un chemin peux contenir des informations (e.g `:id` dans l'exemple précédent). Ces informations sont les informations permettant d'identifier la ressource. La clé primaire d'une base de données par exemple.

Entrée.s : On doit aussi définir les objets supplémentaires en entrée. La description de ces données incombe à l'entité chargé de la mise en place de l'API REST.

Sortie.s : permet de définir les objets supplémentaires en sortie. Comme pour l'entrée, cette tache incombe à la même personne.

Erreur.s de sortie : Les code de statut HTTP sont couramment utilisés car ils communiquent à eux seuls des informations utiles : par exemple une erreur 401 signifie que le client n'as pas la permission d'accéder à la ressource.

1. `https://fr.wikipedia.org/wiki/Interface_de_programmation`
2. `https://fr.wikipedia.org/wiki/Codage_différentiel`

3.1.2 Outils utilisés

La mise en place de l'API REST à été faite au moyen de swagger [3], qui permet via un fichier de configuration en YAML [4] de générer une page de presentation des différents fonctionnalités et de les tester. Swagger dispose d'un éditeur en ligne, qui permet d'afficher un rendu en temps réel de la configuration.

L'ensemble de l'api rest est disponible sur le git du serveur [5], dans le fichier `e-swipe.swagger.yml`.

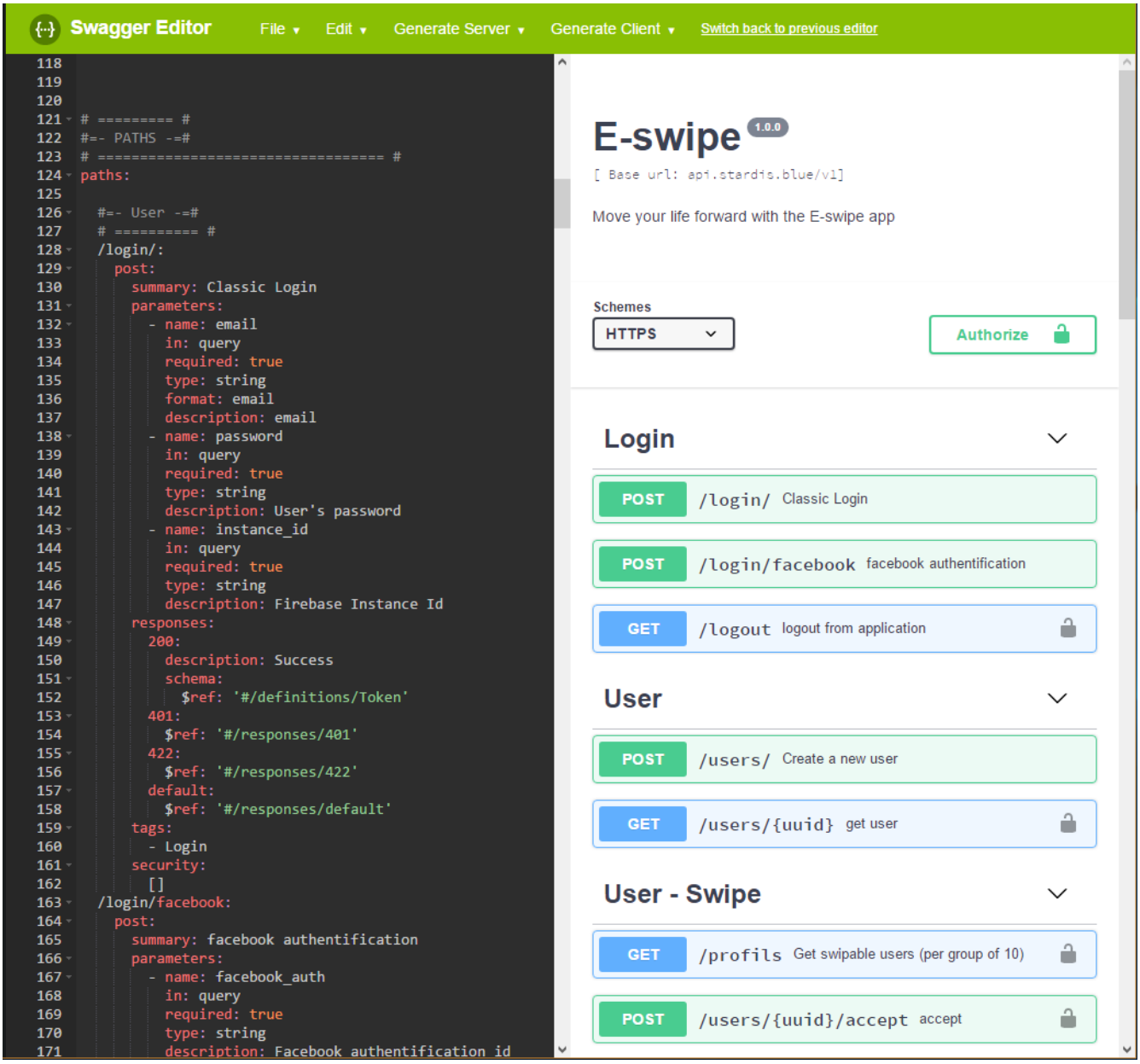

FIGURE 10 – Interface de l'éditeur

3. http://swagger.io/
4. http://yaml.org/
5. https://github.com/e-swipe/e-swipe-server/

Nous n'aborderons pas la création du fichier de configuration, car la spécification [6] est complexe et n'est pas pertinente dans le cadre de ce rapport. Malgré tout, nous présenterons les différentes avantages de cet outil ainsi que ce qui à été crée avec, c'est à dire, l'API.

Remarque : afin de visualiser le fichier de configuration, on peux importer le fichier de configuration dans l'éditeur (en haut à gauche `File>Import URL` ou `File>Import File`)

Remarque 2 : afin de récupérer l'url de la configuration directement depuis le git, utiliser le bouton `Raw` lors de l'a visualisation d'un fichier sur Github.

3.1.3 Structure de l'API

La structure globale de l'api suit les normes imposées par REST. Cependant, certains routes ont été dénormalisés pour un souci de structure et de compréhension.

Les fonctionnalités de l'application repose sur 6 points importantes :

1. La connexion/déconnexion
2. L'utilisateur connecté
   - Profil
   - Ses Chats
3. Les Utilisateurs à «swiper»
4. Les Utilisateurs
5. Les Évènements
6. Les Chats

Par conséquent, l'API possède la structure suivante :

1. `POST /login` : connexion classique (email, mot de passe)
   - `POST /login/facebook` : connexion via facebook
   - `GET /logout` : déconnexion
2. `GET|PATCH /me` afin de récupérer et mettre à jour le profil d'un utilisateur
   - `GET /me/chats` afin de récuperer la liste de ses discussions
3. `GET /profils` : les utilisateurs à «swiper»
4. `GET /users/:uuid` : récuperer les informations d'un utilisateur (:uuid est l'identifiant de l'utilisateur)
   - `POST /users/:uuid/accept` : like un utilisateur
   - `POST /users/:uuid/decline` : refuse un utilisateur
5. `GET /events/` : liste des evenements
   - `POST /events/:uuid` : Détail d'un évènement pas son id(uuid)
   - `POST /events/:uuid/participate` : notifier que l'on participe à un évènement
   - `POST /events/:uuid/refuse` : notifier que l'on ne participera pas
6. Les Chats
   - `GET /chats/:uuid` : récupere les informations du chat ainsi que la liste des messages

6. `http://swagger.io/specification/`

- `POST /chats/:uuid` : permet l'envoie d'un nouveau message au chat.

Afin d'effectuer l'ensemble de ces actions. Il a été nécessaire de concevoir l'ensemble des objets à envoyer et à recevoir. La structure de l'ensemble de ces objets est décrite dans le fichier de configuration swagger ou peux être visualisée à la fin dans l'éditeur.

3.1.4 Conceptions des objets

La conceptions des données en entrée et en sortie doit être détaillée. Par souci d'optimisation, il est aussi important que les données ne contiennent que ce qui est indispensable.

Par conséquent, Il y plusieurs sous-objets décrivant ce qui est nécessaire.

Par exemple, il y à un objet `Event` et il y a aussi un sous-objet `EventCard` qui contient bien moins d'informations que `Event` et qui est utilisé pour avoir la liste des évènements.

3.2 Android

Dans cette partie nous décrirons les différentes fonctionnalités implémentées coté Android en expliquant les choix techniques. Pour chaque fonctionnalités, nous donnerons l'ensemble des routes utilisées.

3.2.1 Connexion

Afin de réaliser notre système de connexion via Facebook et Mail/Password, nous avons dû définir un système permettant l'authentification d'un client via un identifiant unique. Cet identifiant est renvoyé au client lors de sa connexion. Il permettra d'authentifier le client lors de l'envoi de requêtes au serveur (POST, GET, PATCH ...). La mise en place du système se divise donc en deux parties distinctes :

- Un système classique de connexion grâce à l'email de l'utilisateur et son mot de passe. Lors de la connexion, le serveur renvoi au client un token d'authentification unique que l'application utilisera lors de l'envoi de requêtes au serveur.
- Le système de connexion via Facebook est plus complexe. Dans un premier temps grâce au Facebook Login Button nous pouvons si l'utilisateur est déjà connecté à Facebook récupérer un token d'accès (Dit AccessToken). Cet AccessToken est le token représentant l'utilisateur (Son compte Facebook) ainsi que les accès (Permissions) demandées. Dans le cadre de cet application, les permissions demandées par l'application sont : "id,name,email, gender , birthday , albums" Ce token d'accès est ensuite envoyé avec les informations reçu par Facebook au serveur qui autorisera ou non la connexion de l'utilisation grâce à ce token Facebook. Si la connexion est autorisé le client recevra le token qui lui permettra de communiquer avec le serveur.

Routes utilisées :

- POST /login
- POST /login/facebook

3.2.2 Visualisation et édition d'un profil

L'édition et la visualisation d'un profil sont des concepts assez simples mais qui peuvent se révéler très complexes en terme d'envoi de requêtes se modification des informations utilisateur. Lors de l'arrivée sur la fonction `onCreate()` du fragment, nous récupérons les informations client de façon asynchrone puis initialisons les vues en conséquences. Il aurait été possible de mettre en place un système de permettant l'état courant d'un profil (Les options sélectionnées) mais nous avons préférer éviter les possibles problèmes que pourraient emmener un décalage entre les informations coté client et celle coté serveur.

Routes utilisées :

- GET /me
- PATCH /me

3.2.3 Swipe

La mise en place d'un système de Swipe est assez complexe. Les CardView (Cartes) sont composés de deux informations importantes,les informations utilisateurs (Nom , Age) ainsi que la distance en fonction de la position courante de l'utilisateur. A l'arrivée sur le fragment de swipe, le client (l'application) demande en fonction de la position actuelle du client et de la distance, les goûts demandés lors de l'édition de son profil de nouvelles cartes (Représentant un utilisateur compatible). Lors de la réception d'une nouvelles listes de personnes à ajouter, nous utilisons la classe `Location` qui permet de calculer la distance entre la personne à swipe et la position de l'utilisateur.

Route utilisée : GET /profils

3.2.4 Évènements

Les gestion des évènements reprend les mêmes principes que celle des cartes utilisateurs. L'application récupère une liste d'évènements avec leur position et calcule la distance entre la position actuelle du client et celle de l'évènement. Nous avons également optimisé notre système notre système de récupération des événements. Afin de ne pas récupérer l'ensemble des événements lors de chaque appel, nous avons mis en place un système de requêtage dynamique.Lors du premier appel, l'utilisateur reçoit une liste de 10 évènements, ces évènements sont ensuite charger dynamiquement dans la Listview. Lorsque l'utilisateur a fini de scroll dans la première liste d'évènement, l'application demande au serveur les nouveaux évènements grâce à la mise en place d'un offset puis charge dynamiquement la nouvelle liste d'évènement en les ajoutant à la Listview. Si l'utilisateur ne souhaite pas utiliser la pagination dynamique il

Cet méthodes bien que coûteuse en nombre d'appel, permet de réduire la taille des données reçus et le temps d'attente (Temps de chargement) entre l'arrivée sur le fragment et le moment ou les données sont affichés au client.

Route utilisée : GET /events

3.2.5 Chat

Afin de réaliser une application de type Chat nous avons du développer une application reprenant les concepts de push notification/data. Ce système se différencie du http polling qui se base sur le fait de requêter des données au serveurs à un interval de temps régulier. Ici, c'est le serveur qui enverra les données au client par le biais de webSocket et d'un système d'authentification du client. Lors de la mise en place des webSocket, il est possible de faire de faire passer des données du client au serveur (Upstream) ou serveur au client (Downstream). En plus de la mise en place d'un système de cloud messaging, `Firebase` propose différents composants comme la détection de crash (Crash Reporting), authentification ... Dans le cadre de ce projet, nous allons principalement utiliser le `Google Cloud Messaging`.La mise en place d'un tel système ce divise en plusieurs parties :

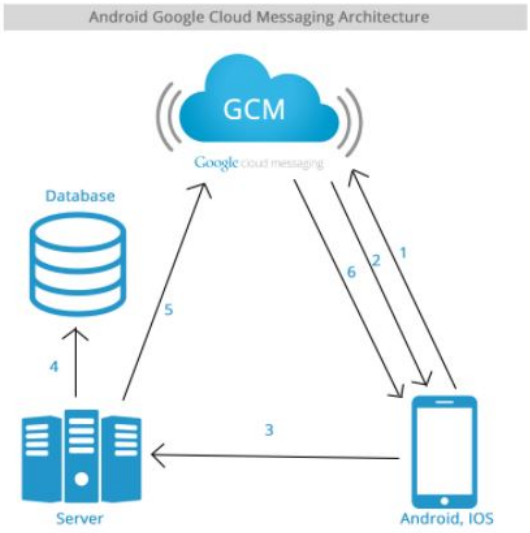

FIGURE 11 – Schéma de communication Firebase

- Première étape : La demande auprès du Google Cloud Messaging du token d'identification du client. Cette mise en place s'effectue grâce au service `FirebaseInstanceIdService`. Ce service Binder coté client possède une méthode principale : `onTokenRefresh()`. Cette méthode est appelé lors de la réception du token par le client. Le token étant également expirable coté serveur (Google Cloud Messaging), cette méthode sera également appelé lors du renvoi du token suite à une expiration. Le token renvoyé par le serveur est l'identifiant de l'appareil sur le serveur GCM, par conséquent il est unique. Grâce à l'initialisation de ce token connu coté client/serveur il sera possible d'initialiser le système de push notifications.
- Lorsque la demande de token est effectué l'utilisateur reçoit son identifiant unique (Token) la méthode `onTokenRefresh()` est appelé et le token doit être géré coté client.
- Lors de la réception coté client, l'utilisateur doit sauvegarder son token sur le serveur afin de pouvoir identifier le client sur le serveur (Association Compte <-> FirebaseId). Cette association sera alors sauvegardé sur la base de données lié à l'application.
- Afin de communiquer avec le serveur, il est nécessaire d'envoyer le token identifiant le client ainsi que le message entre le serveur applicatif et le serveur Google Cloud Messaging. La requête d'envoi est de la forme :

Listing 2 – Exemple de push data

```
{
    "notification": {
        "title": "Event␣Name",
        "text": "A,␣B␣et␣10␣personnes␣participent␣a␣cet␣
          ↪ evenement",
        "click_action": "EVENT_ACTIVITY"
    },
    "data": {
        "image_url":"l'url␣de␣l'evenement",
        "event_id":"10"
    },
    "to" : "to_id(firebase␣refreshedToken)"
```

```
12 || }
```

- Lors de la réception du message coté client, la service `FirebaseMessageService` est appelé sur la fonction `onMessageReceived()`. C'est ici que sera gérer la création d'une notification coté client et la mise en place d'un broadcast entre l'activité `ChatActivity` et le service. A la réception du service nous envoyons un broadcast sur l'activité chat afin de rajouter le dernier message à la liste des messages déjà affiché. Ce broadcaster nous permet de gérer la réception du message lorsque l'application est dite "onForeground" et "onBackground".

Ce système nous permet de faire passer le moins d'informations possibles (On envoi que le message envoyé par le client) et peu coûteux niveau batterie, utilisation d'internet (Pas besoin de redemander au serveur tout les messages à un interval régulier ...)

Routes utilisées :
- POST /chats/ :uuid
- GET /me/chats
- POST /chats/ :uuid

3.2.6 Informations supplémentaires

Afin de réaliser cette application nous avons utilisé de nombreuses librairies (Disponibles en annexe) et tenté d'optimiser le passage de données en utilisant l'interface Parcelable qui permet d'écrire un objet dans un `parcel` et donc permettre la récupération de données plus rapidement (Actuellement Parcelable est 10 fois plus rapide que Serializable). L'utilisation d'une TabbedActivty est un grand avantage car il permet de charger dynamiquement les fragments qui se trouve sur la gauche et la droite du fragment courant, ce qui permet un meilleur feedback pour le client qui n'a pas besoin d'attendre que le fragment soit chargés entièrement. Nous utilisons pour permettre l'envoi de requêtes et la récupération d'informations la librairie okHttp couplé avec avec la librairie gson qui ont le grand avantage de posséder un système de misse en cache des requêtes (Pour okHttp) et la possibilité de mapper le json obtenu depuis le serveur en un objet java.

3.3 Serveur

Dans cette partie sera expliqué comment ont été mis en place les différents chemins. La structure de la partie sera la meme que pour android afin de faciliter la lecture de ce rapport.

Avant-propos : Le serveur à été concu en `PHP` et à l'aide du framework «CakePHP» [7]. Par conséquent lors de l'explication de cette partie, si certains éléments exigent une connaissance particulière au niveau du fonctionnement de «CakePHP» des liens vers la documentation seront présente en bas de page, il SERA aussi expliqué leur utilité, mais leur fonctionnement ne sera pas présenté.

Remarque : La «base URL» de l'api est `https://api.stardis.blue/v1/`. Toutes les appels à l'api doivent être préfixés par cette URL.

3.3.1 Pré-requis

S'il est nécessaire d'exporter l'application ou de la réimplementer. Afin de garantir le fonctionnement de l'application, il est important de configurer les paramètres de cakephp [8]. Il faut définir les accès base de données et d'ajouter deux paramètres dans la configuration (`config/app.php`) :

- Facebook :

Listing 3 – configuration Facebook

```
'facebook' => [
    'id' => 'KEY_ID', // la app id facebok
    'key' => '', // la cle secrete de l'application facebook
],
```

- Google :

Listing 4 – configuration Google

```
'google' => [
    'key' => 'SECRETKEY', // la cle secrete de l'application
        ↪ de connexion a firebase
],
```

De plus, l'import des dépendances se fait via Composer [9]

routes : La liste des routes est définie dans `config/routes.php`

Validation des données : Afin de valider les données en entrée, la classe `App\Validator-\DataValidator` est utilisée. elle contient une serie de fonction statiques permettant : d'afficher les erreurs suivant la route (fonctions commencant par `validate<Controlleur><fonction>()` e.g `validateLoginFacebook()`) et de valider les objets en entrée (fonctions commencant par `<objet>Validator()`).

7. `https://cakephp.org/`
8. `https://book.cakephp.org/3.0/fr/development/configuration.html`
9. `https://getcomposer.org/`

Mapping API : les objets et leurs variables diffèrent entre l'api et la base de données. par conséquent, une série d'adapteurs ont été mis en place, il se trouvent dans le namespace `Eswipe\Model`

Notifications Push : La structure des notifications ainsi que le processus d'envoie se trouve dans `Eswipe\Android`

3.3.2 Authentification

Routes : `POST /login`, `POST /login/facebook` et `POST /logout`

`POST /login`, `App\Controller\LoginController::basic` : Le système de login classique est assez simple à mettre en place : lorqu'un utilisateur se connecte, il suffit de vérifier si l'email est dans la base, récupérer le mot de passe hashé dans la base et verifier si le mot de passe fourni est le même que celui de la base.

Il serait certes plus simple de vérifier si il existe une entrée dans la base avec l'email et le mot de passe crypté, mais ceci peux faire l'objet d'une attaque temporelle [10].

`POST /login`, `App\Controller\LoginController::facebook` : Le système de login facebook est plus complexe. On recoit de la part du client le acces_token facebook (input_token pour le cas du serveur), ainsi que les information de l'utilisateur. Afin de garantir la validité de la clé, on va envoyer une requete à l'api facebook [11]. Si celle-ci nous renvoie pas d'erreurs, cela signifie que cet input_token est valide.

On à alors deux possibilités. Soit l'utilisateur n'existe pas, dans ce cas, on le crée en s'assurant que l'utilisateur n'est pas inscrit. Une fois inscrit ou s'il existe, on l'authentifie.

Authentification : Une fois la connexion confirmée (qu'elle soit classique ou par facebook). On génère un token aléatoire unique qui identifie le client au niveau du serveur. Sa durée de vie n'est pas garantie. Ce token donne accès aux autres fonctionnalités de l'application.

Le reste de l'application est dépendante du token d'authentification. Toute action identifié par un token incorrect donnera lieu à une erreur 401.

Afin d'automatiser ce systeme. La classe `App\Auth\ApiAuthenticate` à été mis en place. Elle permet de vérifier au niveau du Controlleur [12] (MVC [13]) si celui-ci à le droit de s'exécuter [14].

Un controlleur parent est mis en place afin que toutes les controlleurs de l'application implementent cet automatisme.

`POST /logout`, `App\Controller\UsersController::logout` : afin de le déconnecter, il suffit

10. https://fr.wikipedia.org/wiki/Attaque_temporelle
11. https://developers.facebook.com/docs/facebook-login/access-tokens/debugging-and-error-handling?locale=fr_FR
12. https://book.cakephp.org/3.0/en/controllers.html
13. https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur
14. https://book.cakephp.org/3.0/en/controllers/components/authentication.html

d'invalider le token identifiant l'utilisateur.

3.3.3 Visualisation et édition du profil

Routes : `GET|PATCH /me`

Controlleur concerné : `App\Controller\ProfilController`

Lors de la requete de récuperation des données utilisateur, on charge les dépendances de celui-ci (ses images et les evenements ...) et on renvoie l'objet en Json.

L'edition de profil valide les données modifiés (en veriant leur type). Et renvoie un message d'erreur (avec le statut 422) ou le statut 204.

3.3.4 Swipe

Routes : `GET /profils`, `GET /users/:uuid/accept` et `GET /users/:uuid/decline`

Controlleur : `App\Controller\UsersControlleur`

Lorsque l'on récupère les profils. On doit restreindre cette recherche avec plusieurs contraintes :

1. dans un rayon (km) par rapport à ma position actuelle
2. qui sont interressés par ce que je suis (mon genre)
3. qui sont du genre qui m'intéresse
4. que je n'ai pas encore swipé («liké» ou «disliké»)
5. que je n'ai pas encore matché
6. qui sont dans l'intervalle d'age que je recherche
7. qui ne sont pas moi.

Le but à donc été d'optimiser la requete afin que tout calcul complexe soit fait une fois (lors de la requete) et non pas dynamiquement lors de la recherche dans la base.

Par exemple : l'âge d'une personne n'est pas stockée, seulement sa date de naissance. L'approche simpliste consiste donc dans ma requete, de convertir cette date en Age et de vérifier si cet age est dans l'intervalle désiré. Cette operation de conversion sera donc faite n fois où n est le nombre d'utilisateurs. L'approche optimisée est de convertir l'interval d'âge en interval de date (en soustrayant l'age min ou l'age max à aujourd'hui). De cette facon, même si cette conversion est plus couteuse. Elle n'est effectué qu'une seule fois, lors de la construction de la requete.

Recherche dans un rayon (km) : cette operation est couteuse, par conséquent il est important de la réduire, elle est dépendante de la haversine formula [15]. Afin de l'optimiser, on limite d'abord les utilisateurs à prendre en compte en récupérant le carré circonscrit du cercle (via haversine), on à alors la latitude(min-max) et la longitude(min-max). Cependant, on récupère aussi

15. `https://en.wikipedia.org/wiki/Haversine_formula`

les profils des utilisateurs hors du cercle. l'approche est donc de valider les utilisateurs dans cet surface. Par conséquent, on récupère aussi les utilisateurs du carré inscrit au cercle (on est sur qu'ils sont dans le cercle) et on les valide. Enfin reste les utilisateurs dans le cercle hors du carré inscrit. Ceux-ci, on les valide via la haversine formula. Ainsi on limite l'utilisation de la haversine formula de 70%.

Lorqu'un utilisateur est dislike, on vérifie d'abord que l'utilisateur ait la permission de le faire (qu'il ne l'ait pas swipe ou match, que l'utilsateur concerné est visible ....). Le comportement du like est à peu près le meme pour la vérification.

A la difference du dislike, le like va verifier plusieurs conditions supplémentaires : on vérifie si l'utilisateur liké nous a aussi liké. Si c'est le cas, alors on crée un nouveau chat, on crée un match auquel est associé le chat et on notifie les deux utilisateurs via push.

3.3.5 Evenements

Routes : `GET /events`, `GET /events/:uuid`, `GET /events/:uuid/participate` et `GET /events/:uuid/decline`

Controlleur : `App\Controller\EventsControlleur`

Il suffit ici de récuperer la liste des evenements, elle est restrainte par un systeme de pagination.

Lorsque l'utilisateur participe à l'évènement, on verifie et on supprime le fait qu'il l'ait refusé et inversement.

3.3.6 Chats

Routes : `GET /me/events/` et `GET|PUT /chats/:uuid`

Controlleur : `App\Controller\EventsControlleur`

L'utilisateur n'as accès qu'à son chat, par conséquent on limite cet accès via /me.

Lorsque un chat est récuperé, on envoie aussi l'autre utilisateur ainsi qu'une liste de messages. Il y a deux facons de limiter le nombre de messages : limit et since. Limit est la quantité de messages à recevoir alors que since est une date : seuls les messages écrits après cette date seront affichés.

4 Perspectives

Notre application bien que fonctionnelle pourrait être améliorer grâce à l'ajout de nouvelle fonctionnalités telles que :

- L'envoi d'un email de confirmation du compte
- La possibilité de changer son mot de passe via son email
- La possibilité de supprimer son compte si on ne souhaite plus que son profil soit sur l'application
- La mise à jour automatique des évènements
- La possibilité de connaître les évènements en commun entre 2 utilisateurs lors du swipe
- La possibilité d'accéder à la fiche d'un évènement
- L'affichage des photos Facebook que nous avons récupéré.
- La possibilité de récupérer les photos de la galerie photo de notre smartphone.
- Finaliser la partie évènement de l'application
- La possibilité d'accèder au profil d'un autre utilisateur à partir d'une userCard (cf. 2.3.2)

Glossaire

Swipe : Interaction avec l'écran tactile en déplaçant rapidement son doigt sur les profils des utilisateurs pour *match* avec ces derniers. Un swipe vers la gauche pour rejeter la personne et un swipe vers la droite pour accepter la personne.

Match : Correspondance entre 2 utilisateurs qui se sont *swipé* positivement. Lorsqu'il y a match, 2 utilisateurs peuvent se parler dans un chat.

Firebase : Firebase est un ensemble de services d'hébergement pour n'importe quel type d'application (Android, iOS, Javascript, Node.js, Java, Unity, PHP, C++ ...).

Annexes

Librairies utilisées

```
1
2       //GooglePlayServices
3       compile 'com.google.android.gms:play-services:10.2.0'
4       //Firebase and facebook : Firebase and facebook sdk
5       compile 'com.google.firebase:firebase-core:10.2.0'
6       compile 'com.google.firebase:firebase-messaging:10.2.0'
7       compile 'com.facebook.android:facebook-android-sdk:[4,5]'
8       //Circle Image View : Display image in a circle without crop
9       compile 'de.hdodenhof:circleimageview:2.1.0'
10      //RangeSeekBar : Seek bar that handle a range of values
11      compile 'org.florescu.android.rangeseekbar:rangeseekbar-library
          ↪ :0.3.0'
12      //PlaceHolder for TinderCards + Cardviews
13      compile 'com.mindorks:placeholderview:0.6.1'
14      //CardView : Cardview for tinder card layout
15      compile 'com.android.support:cardview-v7:23.2.1'
16      //Image Loading Library : Handle the download of an image from
          ↪ url and load it in imageView
17      compile 'com.github.bumptech.glide:glide:3.7.0'
18      //Gson Loader :Handle mapping object -> Json && json -> object
19      compile 'com.google.code.gson:gson:2.7'
20      //OkHttp : Handle http connexion to server
21      compile 'com.squareup.okhttp3:okhttp:3.8.0'
```

Story Board

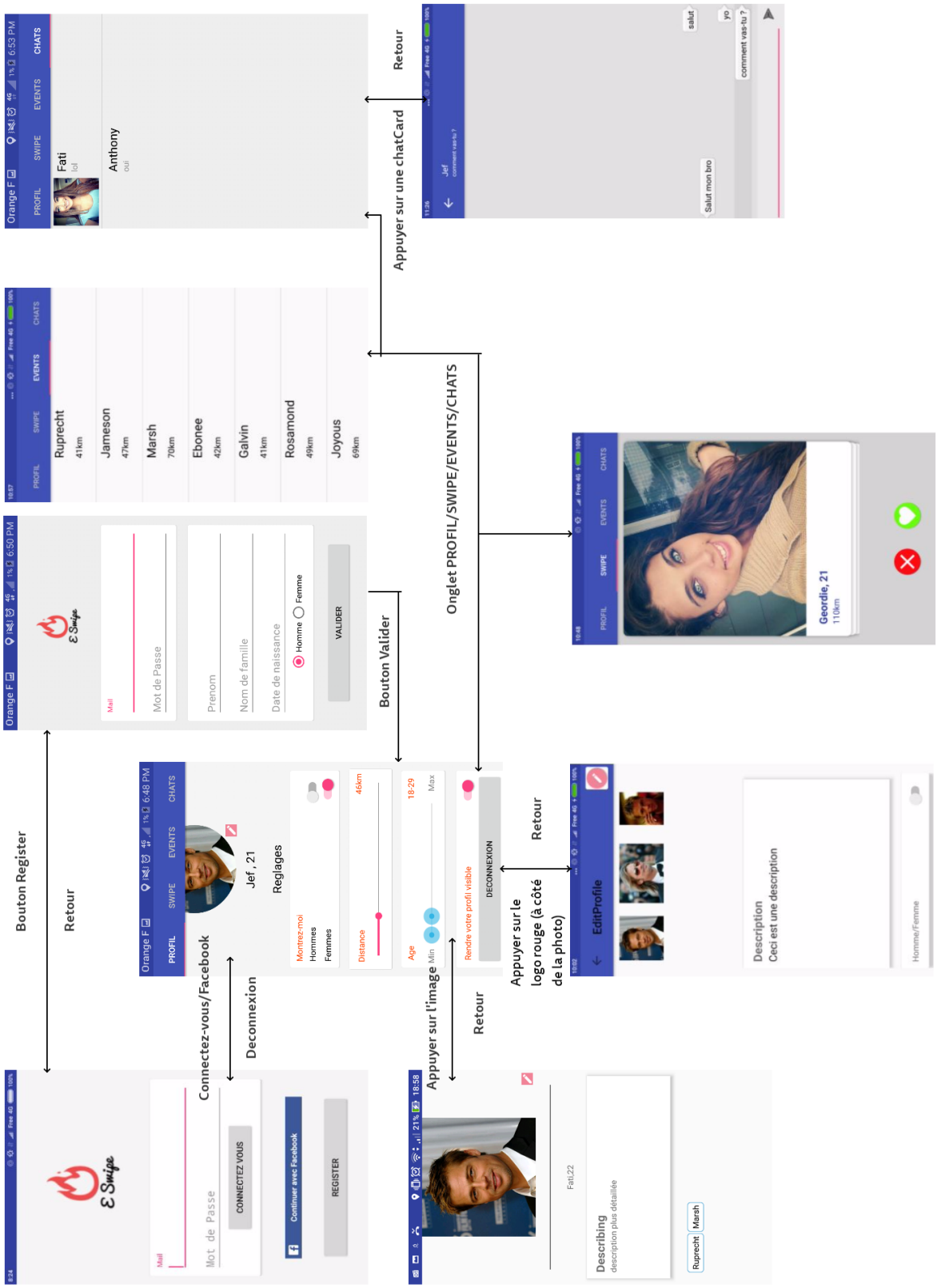

FIGURE 12 – StoryBoard