UNIVERSITÉ DE SHERBROOKE

IFT604 – Applications Internet et mobilité

Projet de session - Bingo

Travail présenté à
M. Frédéric Bergeron

Par

Guillaume Cadorette – 17 040 202
Francis Hamel – 17 092 370
Gabriel Lacroix - 17 096 517
Alexandre Larochelle - 17 048 703
Alexandre Péloquin-Emond – 17 159 110
Nikola Popovic – 17 084 058

7 décembre 2020

Table des matières

Contexte

Durant la crise sanitaire actuelle, nous avons dû rapidement trouver des moyens pour retrouver un équilibre de vie sain. Les évènements en ligne, les concerts virtuels, les appels vidéo, les jeux vidéo et beaucoup d'autres nous ont permis de rester connecter avec nos proches à distances.

Toutefois, pour plusieurs il n'est pas possible de visiter ses proches et une population particulièrement ciblée seraient les personnes âgées en CHSLD ou maisons pour personnes âgées.

Ainsi, nous avons décidé de concevoir une application de Bingo multijoueur qui pourrait autant plaire aux ainés qu'à leurs petits enfants pour un plaisir intergénérationnel assuré.

Architecture

Backend

Déroulement

Nous avons décidé de faire le backend en Node.js avec le serveur express.js puisqu'il s'agit d'une technologie avec laquelle tous les membres de l'équipe se sentait relativement à l'aise. De plus, avec nos expériences dans les devoirs précédents de la session, nous savions que nous pourrions réutiliser certaines parties de code, accélérant ainsi notre développement.

Modèles

Les modèles de notre application représentent les classes étant utilisé dans notre backend. Ils sont situés dans le dossier backend/src/models.

Carte

La classe Carte sert à générer et à stocker une carte de Bingo. La carte connait ses cases (qui sont représentées par une matrice 2D de nombre), d'un id et du id de son lobby. Elle nous permet aussi d'avoir directement accès aux informations nécessaires afin de pouvoir valider si un joueur a eu un bingo (tels que les colonnes, les lignes et les diagonales.)

Géolocalisation

La classe Geolocation sert à effectuer le calcul entre deux coordonnées pour savoir si on est à une distance acceptable d'une partie.

Joueur

La classe Joueur représente un joueur de notre application. Un joueur contient son id, le lobby auquel il appartient, le jeton Firebase qui lui est associé, le nom d'utilisateur ainsi que la carte qui lui est associé pour la partie en cours. On peut modifier son nom d'utilisateur.

Lobby

La class Lobby représente une partie. Il y a un id, un nom de partie, les boules pigées, le status de la partie (si elle est commencée et si elle est terminée) ainsi que les prochaines boules à envoyer. Lors de la création de la partie, on va générer les boules à envoyer (les 75) et les mélanger dès le début. Aussi, le

déroulement de la partie est géré grâce à startGame. On y retrouve un setInterval qui envoie une nouvelle boule aux 30 secondes (à l'aide d'un message Firebase).

Routes

Les fichiers de routes, qui sont à src/routes, contiennent les routes de notre API de jeu de Bingo afin que le frontend puisse communiquer avec le backend.

Lobby

Les routes présentes dans Lobby sont celles qui ont rapport à la création et au démarrage d'une partie. En cas d'erreur, toutes les routes suivantes retourne une erreur 400 avec une description de ce qui a causé l'erreur. Toutes les routes suivantes sont accessibles à partir de /bingo/lobby (donc par exemple localhost:3000/bingo/lobby). Voici la liste des routes présentes dans le fichier lobby avec leurs descriptions:

- / - GET : Permet d'obtenir les parties qui sont situées à une distance raisonnable. Elle prend en paramètre la longitude et la lattitude du client et retourne une liste de lobby.
- / - POST : Permet de créer une partie. Ici, le joueur créant la partie va être considéré comme étant le "host", ce qui lui donnera le droit de commencer la partie. Cette partie va être accessible aux joueurs étant dans un rayon raisonnable de l'host. Si la création s'est faite avec succès, on retourne les informations du lobby.
- /:id - GET : Permet d'obtenir un lobby selon son id. Si le lobby existe, on le retourne.
- /:id/start - POST : Permet de commencer un lobby quelconque si la personne ayant appelé cette route est un host. En appelant start, on envoie une carte à tous les joueurs étant abonné au lobby (voir /:id/user pour plus de détails) et on commence à généré des boules qui vont être envoyées au lobby.
- /:id/user - POST : Si un joueur n'est pas déjà dans un lobby et que la partie n'est pas commencé, on abonne le joueur au lobby désiré. L'abonnement va permettre à Firebase d'envoyer les boules qui seront générées lorsque la partie va être commencé, de plus que de recevoir sa carte. Une fois le joueur est abonné, on lui remet le id du lobby.
- /:id/user - DELETE : Si le joueur fait partie du lobby, on l'enlève du lobby et on le désabonne de la partie.
- /:id/win - POST : Valide si une carte est en état de victoire. Si c'est le cas, la partie se termine et on indique au joueur qu'il a gagné la partie.

User

Les routes présentes dans User sont celles qui ont rapport à la création d'un joueur et à sa modification. Les routes suivantes sont accessibles à partir de /bingo/user. Voici la liste des routes présentes dans le fichier User :

- / - POST : permet de créer un utilisateur. On reçoit le nom d'utilisateur et le jeton Firebase en paramètres et on retourne une représentation JSON du nouveau joueur.
- /:id - PUT : permet de modifier le nom d'utilisateur ainsi que le jeton Firebase d'un utilisateur.

Autre

Messaging

La classe Messaging comporte toute la logique concernant l'envoi de messages Firebase. Firebase permet l'envoi de messages de type PUSH sous 2 formats : *notification* et *data*. Dans notre cas, nous utilisons seulement les messages de type *data*. Chaque partie à son *topic* associé et on va donc cibler tous les joueurs d'une même partie avec un message. Lorsqu'un joueur se joint à une partie, le *backend* va abonner le joueur au *topic* de la partie. Le *backend* va aussi désabonner le joueur du *topic* si le joueur quitte la partie. Voici l'explication des différents messages utilisés :

- "carte": ce message va cibler chaque joueur d'une partie individuellement pour lui envoyer sa carte lorsque la partie début.
- "addedPlayer": ce message va notifier les joueurs d'une partie qu'un nouveau joueur vient de joindre le lobby.
- "removedPlayer": ce message va notifier les joueurs d'une partie qu'un nouveau joueur vient de quitter le lobby.
- "nextBoule": ce message va notifier les joueurs d'une partie de la nouvelle boule qui vient d'être pigée.
- "winner": ce message va notifier les joueurs que la partie vient d'être gagnée par le joueur présent dans le message.

Database

Puisqu'il n'était pas obligatoire d'avoir une vraie base de données, nous avons décidé de faire une classe émulant une base de données qui contient les données de l'application afin d'avoir de la cohérence durant une utilisation du serveur. Notre "base de données" contient les informations sur les cartes, les lobby et les joueurs faisant partie du système. Toutes les méthodes présentes dans cette classe ne servent qu'à:

- Accéder à un élément
- Supprimer un élément
- Ajouter un élément

Util

Util est une classe utilitaire qui permet simplement d'avoir accès à certaines constantes utilisées dans l'application ainsi qu'à une fonction de distance.

Frontend

Déroulement

Puisque nous souhaitions que l'application soit facilement accessible, peu importe où l'utilisateur se trouve, nous avons décidé de développer le client sur les appareils mobiles. De cette façon, le joueur peut jouer avec des personnes qui se trouvent dans le même quartier que lui, peu importe où il se trouve. Plus précisément, nous avons choisi d'implémenter une application Android.

L'application fonctionne donc sur n'importe quel appareil Android qui fonctionne sous l'API 27. Il est possible que l'application fonctionne sur d'autres versions de l'API, mais nous avons développé avec cette

version en tête. Afin de recevoir les données, l'application communique avec le backend grâce à des services web push et pull. Les communications sont effectuées par HTTP avec des objets JSON.

L'application comprend trois grandes activités

- Le menu principal (*MainActivity*)
- La salle d'attente (*WaitLobbyActivity)*
- Le jeu (*GameActivity)*

Menu principal

Tout d'abord, le menu principal est l'activité initiale que l'utilisateur verra lors du lancement de l'application. Dès l'ouverture de l'application, le joueur sera enregistré sur le serveur. Il sera également invité à entrer un nom d'utilisateur, qui sera visible par les autres utilisateurs et enregistré sur son cellulaire.

Le menu principal laissera trois choix pour l'utilisateur.

- Trouver une partie
- Créer une partie
- Rejoindre une partie

Chacune de celle-ci mène par la suite à l'activité du salon d'attente.

L'option de trouver une partie permet à l'utilisateur de voir toutes les parties non commencées, dans un rayon de 15km. L'utilisateur pourra par la suite choisir la partie qu'il souhaite rejoindre.

L'option *créer une partie* permet à l'utilisateur de créer son propre salon de jeu avec un nom donné. La partie est visible par tous les autres utilisateurs qui se retrouvent dans un rayon de 15km. Il est à noter que la personne créant la partie devient l'hôte par défaut. Lui seul a la possibilité de démarrer la partie.

L'option rejoindre une partie permet à un utilisateur de rejoindre une partie en contournant les limites du 15km. De cette façon, deux personnes habitant à une longue distance l'un de l'autre pourront tout de même jouer au jeu, même s'ils ne sont pas proches l'un de l'autre. Afin de rejoindre la partie, l'utilisation souhaitant la joindre aura simplement à connaitre le nom du lobby.

Salle d'attente

Dans cette activité, tous les utilisateurs ayant joint ce lobby seront visibles. Lorsque l'hôte est satisfait du nombre de participants, ce dernier (et seulement celui-ci) peut commencer la partie. Une requête post est alors envoyée à notre serveur afin de démarrer la partie. Tous les participants sont alors notifiés du démarrage de la partie. Le serveur génère par la suite les cartes de bingo et les envois par push à tous les participants.

L'activité de jeu est initialisée.

Il est également important de mentionner comment l'application gère les utilisateurs. Lorsqu'un utilisateur décide de joindre une partie, il s'inscrira sur le serveur à la liste des membres de cette partie. Ainsi, grâce à une communication par *push* à l'aide de *Firebase,* ce dernier sera notifié lorsqu'un utilisateur

joint ou quitte cette partie et la liste des joueurs en attente pourra donc être mise à jour de cette façon. Nous avons également géré tous les cas possibles afin de pouvoir quitter la salle d'attente. Ainsi, l'utilisateur peut soi appuyer sur le bouton de retour pour quitter la salle d'attente, appuyer sur le bouton *Quit* ou fermer l'application. (onDestroy)

Le jeu

Suite au commencement du jeu par l'hôte, tel que mentionné, chaque utilisateur reçoit sa propre carte de bingo. Ainsi, à chaque intervalle de 30 secondes, le serveur choisira aléatoirement une nouvelle boule. Cette boule est ensuite affichée au-dessus de la carte de bingo, durant le déroulement de la partie. L'utilisateur peut voir, en tout temps, les quatre dernières boules qui ont été tirées.

Afin de gagner une partie de Bingo, l'utilisateur à quatre possibilités. Il peut avoir obtenu une ligne horizontale, une ligne verticale, une ligne diagonale ou les quatre coins.

La validation de la carte est faite en deux temps. Lorsque l'utilisateur obtient une de ces quatre possibilités, le bouton bingo s'active et s'illumine de couleurs pour notifier l'utilisateur qu'il a un bingo dans sa carte. Toutefois, l'utilisateur peut s'être trompé lors de l'identification des cases tirées. Ainsi, une fois le bouton cliqué, une requête est envoyée au serveur afin de valider la carte du joueur.

Lorsque la partie est belle et bien gagnée par un utilisateur, tous les autres utilisateurs sont notifiés qu'un gagnant a été déclaré et cette dernière se termine. Lorsque confirmés, les utilisateurs seront remmenés à l'écran principal afin de commencer une nouvelle partie. Dans le cas contraire, l'utilisateur ayant déclaré un bingo invalide sera notifié que son bingo n'est pas valide et la partie continuera.

Structure du code

Dal

Le code a été séparé en package et fragments afin d'être le plus clair possible. Il respecte également le principe SOLID et permet une réutilisation du code facilement. Ainsi, tout ce qui appartient aux appels de services *Rest* a été placé dans le package dal. (Data access layer).

Nous avons décidé d'utiliser le patron de conception Repository afin de permettre le changement de la source de données facilement. Ainsi, une interface représente toutes les fonctions pour le fonctionnement de l'application. La source de données actuelles *(RestServiceDatasource)* est injectée par l'appelant. De cette façon, si nous décidons dans le futur de changer la source de données, il suffira simplement d'implémenter les fonctions de l'interface et l'application ne sera aucunement affecté.

Une classe, *GenericDataHandler, permet* l'appel des fonctions Rest (Post, delete, Put, Get). Chacun des DAO (DataAccessObject) se chargera donc de construire la requête à effectuer. (Paramètres et URL). Finalement, les classes *DataMappers* se charge d'effectuer le *mapping* entre l'objet Json reçu du backend ainsi que le modèle de l'application.

Fel

Pour ce qui est des éléments graphiques, cela a été placé dans le package fel (frontendlayer). Ainsi, ce package comprend tous les activités et fragments qui seront visibles par l'utilisateur.

Les composantes graphiques ont été séparées en fragment, afin de respecter le principe de *Single Responsability*. Par exemple, pour une carte de Bingo, un fragment a été créé afin de représenter une

seule et unique case ainsi que les actions pouvant être effectuées sur ce dernier. Ce fragment est réutilisé pour chaque numéro afin de créer la carte. Un autre exemple serait l'activité de la salle d'attente. Peu importe d'où l'utilisateur arrive, une seule activité sera appelée.

Model

Les objets métiers sont placés dans le package model. Ainsi, nous pouvons y retrouver les classes qui représentent les cases de Bingo, la carte du joueur, un participant, une coordonnée ainsi qu'une partie.

Service

Afin de ne pas engorger le ThreadUI, l'application utilise les IntentService, comme vu durant le cours. Pour permettre une bonne séparation du code, un service est chargé d'effectuer une seule chose. Ces services sont arrêtés lors de la fermeture de l'activité ou du fragment dans lequel ils ont été appelés.

De plus, afin de permettre une uniformité du code et une facilité de développement, une classe abstraite nommée *GenericRestService* a été créée. Cette classe permet la centralisation de la gestion du résultat des service Rest. Ainsi, si le service reçoit une erreur, l'intent de retour comprendra un Extra qui mentionnera cette erreur. Chacun des services aura donc tout simplement besoin d'hériter cette classe et de surcharger la méthode *OnSuccess* et *OnError.*

Util

Finalement, les fonctions utilitaires telles que l'accès aux *SharedPreference* et le formatage du nom des joueurs ainsi que certaines constantes ont été placées dans le package Util. Dans ce dernier, ce sont donc toutes les fonctions générales qui peuvent être utilisés au travers de l'application.

Librairie et architecture

Gson

Afin d'effectuer le mapping entre les objets Json reçut du serveur ainsi que le modèle du client, l'application utile la librairie Gson de Google, qui permet un mapping automatique d'un texte Json à un objet Java.

FireBase

Firebase est ce qui nous permet d'envoyer des données du serveur. Ainsi, en générant un jeton pour l'utilisateur de l'application et en l'enregistrant sur le serveur, ce dernier pourra envoyer des messages au client lorsque désiré.

Android Networking

Afin de faciliter les appels Rest, l'application utilise la librairie Android-Networking, disponible en OpenSource. Cette librairie permet d'effectuer facilement de construire les appels Rest avec une multitude de paramètres tel que le TimeOut, les paramètres, le type de requête, le Header, etc.

Jinatonic confetti

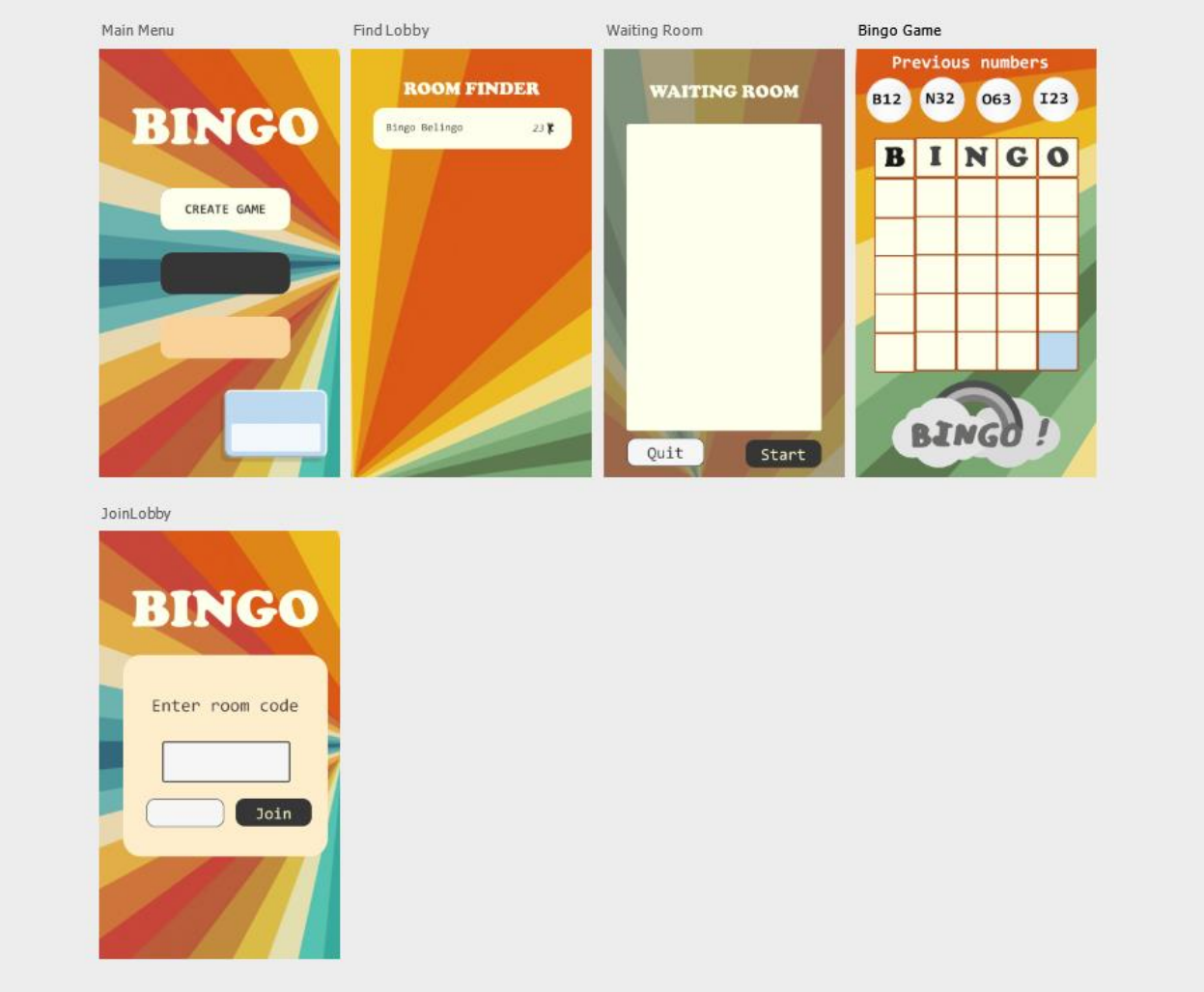
Jinatonic est une petite librairie disponible en source libre qui permet d'afficher des confettis dans l'application. Cette dernière est utilisée afin d'afficher des confettis au joueur qui gagne une partie.

Mise en œuvre

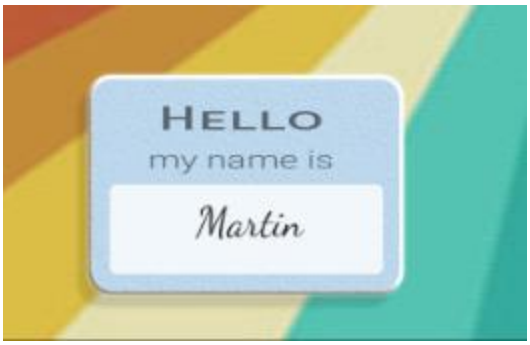
Design

Création des maquettes pour l'application

Afin d'obtenir une application qui plait autant aux personnes âgées que les plus jeune nous avons décidé de miser sur la conception d'une interface nostalgique et attrayante.



Un des choix de conception que nous avons choisi d'entreprendre est de n'avoir pas de service d'inscription ou de connexion sur l'application. Ainsi, dès la première ouverture de l'application, les joueurs seraient en mesure de jouer. Toutefois, nous voulions quand même leur offrir la chance de s'identifier. Nous avons ajouté un "Name Tag" modifiable à partir du menu principal. Encore une fois évoquant une certaine nostalgie et, espérons-le, donnerait un résultat dont le comportement est évident pour les utilisateurs.



Malheureusement, nous n'avons pas pu testé notre application avec des personnes âgés afin de valider que les interfaces sont adéquates pour leur usage. Ce serait une possibilité dans le futur.

Développement des interfaces sur Android Studio

Une fois la conception des maquettes effectuée, nous avons importé les images des composants conçues en tant que drawable. Les images sont ensuite chargées comme fond d'écran pour les composants.
Plus tard dans notre développement, nous avions créé les composants directement à partir de Shape Drawable d'Android. Si nous avions à recommencer ce projet, tous les composants aurait été conçu ainsi.

L'un des difficultés rencontrées durant le projet ciblait notamment la gestion de l'orientation dans le match du Bingo. En gardant tous les éléments à la verticale, lors du changement au mode paysage, il y n'avait plus d'espace pour accueillir la carte Bingo. Pour résoudre ce problème, nous avons lors d'un changement d'orientation (onConfigChanged), récupérer et modifié l'orientation de nos LinearLayouts à l'intérieur de la fenêtre ainsi que la RecyclerView qui contenait les boules des numéros tirés.

Processus

Création d'un utilisateur

Au lancement de la main activity, si un utilisateur n'a pas été créé pour l'appareil (gérer à l'aide d'un booléen sauvegardé dans les *SharedPreferences*), un service pour la création de l'utilisateur est lancé. Celui-ci génère un token Firebase et envoie une requête post au serveur de Bingo pour créer un joueur avec son token. Le serveur renvoie ensuite l'utilisateur créé avec son identiant unique. Une fois la réponse reçue par le client, l'identifiant du joueur est sauvegardé dans l'appareil.

Par défaut, le nom de l'utilisateur est Anonyme.

Mis à jour d'un utilisateur

Si un utilisateur souhaite changer son nom, un champ texte est disponible au menu principal. Une fois que le nom est écrit et confirmé à l'aide de la touche "Enter", le service d'update d'utilisateur est lancé.

Une requête post est envoyé au serveur pour mettre à jours les informations du joueur soit son nom et son token Firebase. En effet, le service d'update est également appelé lorsque le token Firebase expire.

Recherche d'une partie

Une fois l'activité FindLobbyActivity lancé, un service est démarré afin d'envoyer une requête get au serveur pour obtenir la liste des parties. Pour y accéder, nous devons avoir accès au service *Location* d'Android afin d'avoir notre longitude et notre latitude, ce qui va nous permettre de les envoyer au serveur afin qu'il puisse filtrer les parties qui sont à une position souhaitable pour nous.

Il est également possible d'accéder à une partie avec son Id directement. En entrant le id d'un lobby, le frontend envoie une demande au backend afin d'avoir le lobby correspondant au Id. Si la partie est trouvée, alors le service pour joindre la partie va être lancé.

La listes des parties contient également leurs identifiants uniques associé pour pouvoir les joindre. Cette liste est affichée ensuite à l'écran. Chaque rangée étant son propre fragment gérer à l'aide d'un *RecyclerView* et son *Adapter*. L'appui d'un membre de la liste lanceras le service pour joindre la partie.

Rejoindre une partie

Comme nous l'avions mentionné plus tôt, il y a deux moyens pour rejoindre une partie. Par la liste des salons de jeu ou par le menu principal avec le nom du salon. Les deux options utilisent un Intent pour démarrer un service qui envoie un put au serveur avec l'identifiant du salon et du joueur pour ajouter le joueur au salon. Une fois ajouté au salon, le joueur est ajouté à une liste de notification Firebase pour le salon. La requête renvois un objet Salon qui est utilisé pour afficher les autres participants et le nom du salon.

L'activité de salon est munie d'un *BroadcastReceiver* de tel sorte que dès qu'un joueur joint ou quitte le salon, tous les joueurs sont avertis du changement par une requête push de Firebase.

Créer une partie

À partir du menu principal, appuyer sur l'option Créer une partie lance une boite de dialogue afin que l'utilisateur nomme son salon de jeu. Ce nom peut être utilisé pour joindre la partie (mais n'est pas unique). Ayant eu plus de temps à investir, des codes uniques à 4-6 lettres auraient assuré une meilleure cohérence pour les données.

Une fois confirmé, un service est lancé pour envoyer un post au serveur pour créer la partie. La requête renvois un objet salon. L'utilisateur (l'hôte) est déplacé au salon d'attente dans lequel lui seul à la possibilité de démarrer la partie.

Lancer une partie

À partir du salon, l'hôte démarre la partie lorsqu'il le souhaite. Une requête post est envoyé au serveur pour démarrer la partie à partir d'un *IntentService*. Le serveur génère ensuite des cartes et les envois à tous les joueurs à l'aide de Firebase. L'activité de salon est munie d'un Broadcast Receiver et utilise

notre service Firebase pour déplacer les joueurs dans l'activité de jeu lorsque la partie a été lancé par l'hôte.

Jouer une partie

Dans l'activité de jeu nous utilisons encore notre service Firebase. En effet, une fois qu'une partie est lancée le serveur prépare les cartes de jeu pour les joueurs et les envois avec les canaux de Firebase. Ensuite, le serveur envoie périodiquement par Firebase les numéros tirés jusqu'à ce que tous les numéros aient été tirés. Ainsi, le joueur obtient sa carte et périodiquement les numéros, il est prêt à joueur.

Gagner/Perdre une partie

Comme nous l'avions mentionné plus tôt, une fois qu'une personne obtient un Bingo, le bouton s'active et celui-ci peut envoyer une validation de bingo au serveur. Un service est démarré pour envoyer la requête post pour la validation de la carte. Le serveur valide et envoie sa réponse. Si la carte n'a pas vraiment un bingo, la réponse est négative et le joueur est informé de son erreur. Néanmoins, si la carte est valide, tous les participants sont notifiés par une requête post que la partie est gagnée et terminé. L'utilisateur est retourné au menu principal.

Conclusion

Ce projet nous a permis d'en apprendre énormément par rapport à l'utilisation de divers services, autant utilisés par le frontend que par le backend. De plus, nous avons pu en apprendre sur le développement d'applications Android et avoir de l'expérience sur la création d'une application web faite de A à Z.

Bien que les circonstances aient fait que nous n'ayons pas pu faire tester le jeu aux gens qui l'apprécieraient le plus, nous sommes plus que satisfaits par l'expérience et nous espérons que des membres de l'âge d'or vont pouvoir, éventuellement, tester notre application.

Il serait définitivement intéressant d'avoir leur opinion sur le fonctionnement de l'application afin de voir si nos hypothèses quant à la simplicité de notre interface sont vraies.