
# Projet Android


- Rhiss David
- Latino Nathan
- Rosca Sol


Semestre d'automne 2019

# Table des matières

# 1. Introduction

Ce projet est réalisé dans le carde d'un cours de 3e année de Bachelor sur le développement mobile dispensé à la Haute-Ecole Arc à Neuchatel dans la filière Développement logiciel et multimédia.

Le but de ce projet est de se familiariser avec le développement d'applications Android et d'apprendre le langage de programmation Kotlin.

L'application doit utiliser au minimum trois capteurs (internet, GPS, appareil photo,...) et le code de l'application est ecrit dans son intégralité en Kotlin.

# 2. iGolf

Le but d'iGolf est de moderniser les parties de minigolf tout en les rendant plus écologiques en retirant la nécessité de garder les scores sur du papier. Toutes les parties d'un utilisateur sont enregistrées sur un serveur distant et accessibles aussi bien de l'application Android que du site site web (nécessite d'être authentifié pour consulter ses parties).

Des statistiques concernant la difficulté des trous d'un course sont faites à partir des résultats de tous les joueurs. Les résultats des différentes parties d'un joueur sont affichés sur l'application Android sous forme de tableau ou de vision récapitulative sous forme de bar chart.

Les minigolfs ayant souscrit au programme figurent dans une liste consultable sur le site web et font partie de la selection proposée à l'utilisateur de l'application mobile lors de la création d'une partie (suggestion basée sur la proximité physique avec le minigolf).

# 3. Architecture

Il ne nous semble pas pertinent de faire un diagramme complet de l'application, il serait illisible et n'aurait que peu d'interêt. Dans un premier temps nous vous proposons une vue haut niveau de l'architecture avec les différents éléments qui la compose et le flux de données qui les lient. Ensuite vous trouverez un diagramme spécifique aux éléments de la partie présentation (View).

MVVM est un paterne architectural qui a pour but la robustesse et la maintenabilité du code en favorisant au maximum la responsabilité unique de ses composants. Tout comme dans MVC la ségrégation de de la logique et de la présentation est totale.

Dans le diagramme suivant, chaque élément parent possède une référence directe vers son enfant alors que les enfants exposent leurs données via le paterne observateur employé par LiveData.

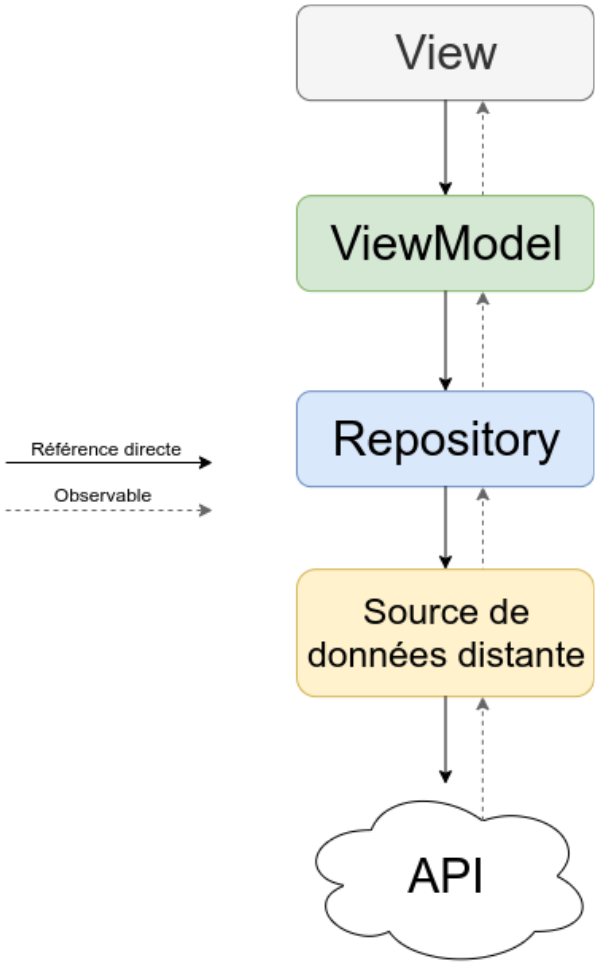


**Figure 1** Vue haut niveau de l'architecture

## 3.1. Vision haut niveau

Dans notre implémentation, les views sont soit des activity, soit des fragment et sont contenues sous `data/activities/` et `data/fragments/`. Les ViewModel sont construits par des factories et instanciés dans `InjectorUtils` qui comme son nom l'indique fait usage du patern d'inversion de contrôle pour injecter les dépendances requises dans les View correspondantes.

Grâce aux LiveData, les ViewModel observent les données contenues dans les classes du packages `/data/repositories` qui à leurs tours observent les données venant de la source distante contenues dans le package `/data/stores` qui utilisent les modèles contenus dans `data/models` pour modéliser l'état de la base de donnée distante. Tout ce circuit est répété indépendament pour chaque modèle.

## 3.2. ViewModel

Dans cette architecture les ViewModel sont la glue qui lie la présentation à la logique métier contenue dans les stores et repositories. Comme le sugère les diagramme, les ViewModel assurent que l'état d'une donnée est propagé dans chaque view l'utilisant lors d'un changement de cette donnée si cette dernière est encapsulée dans un objet LiveData.

## 3.3. LiveData

Les objets LiveData sont des structures de données exposant une api haut niveau qui créent une abstraction autour du design pattern observer. Cela donne la possibilité aux différents composants de l'application d'observer un LiveData qui les notifie en cas de changement de la donnée qu'elles encapsulent. Aussi, LiveData prend en charge le cycle de vie des objets contenu et assure leur que leur destruction se fera correctement. Ce dernier point est souvent un problème dans le cas d'une implémentation maison du patern observer ou même en utilisant d'autres librairies comme Rx et Agera.

# 4. Présentation des données

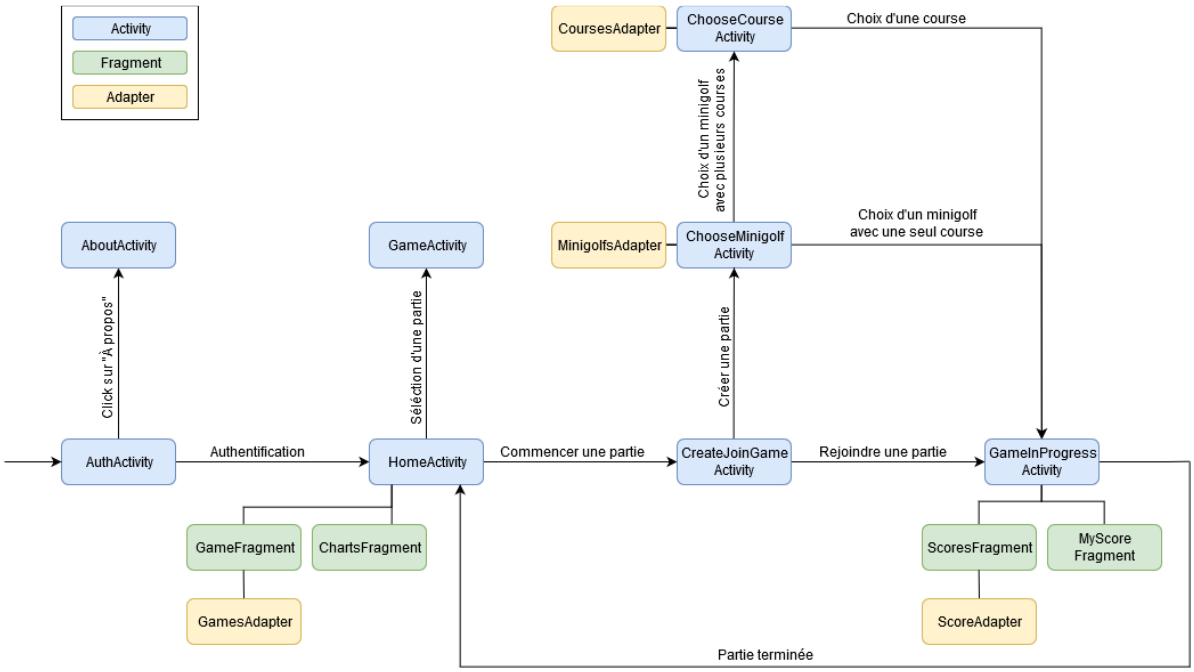


**Figure 2** Flow des données dans la partie View

## 4.1. Activities

### AboutActivity

Contient une `WebView` permettant d'afficher la page "about" du site web.

### AuthActivity

Activité lancée au démarage de l'application, elle permet de s'authentifier grâce aux `TextInputEditLayout`.

### ChooseCourseActivity

Reçoit un objet `Minigolf` et affiche ses courses dans une `RecyclerView` grâce à l'adapter `CourseAdapter`. Possibilité de choisir la course. Crée une nouvelle partie si une course a été choisie.

## ChooseMinigolfActivity

Contient un fragment Google Map qui affiche notre position et le minigolf le plus proche. Va chercher dans l'API les minigolfs et les affiches dans un `RecyclerView` en fonction de la proximité grâce à l'adapeur `MinigolfsAdapteur`. Possibilité de choisir le minigolf. Crée une nouvelle partie si le minigolf choisi n'a qu'une seul course.

## CreateJoinGameActivity

Permet de rejoindre une partie grâce à un code alphanumérique de six caractères que l'on peut entrer dans un `TextInputLayout`. Permet de créer une partie.

## GameActivity

Reçoit un objet `Game` et affiche les scores de chaque joueur dans un `TableLayout`.

## GameInProgress

Reçoit le code de la partie et l'affiche. Affiche les fragments `MyScoreFragment` et `ScoresFragment` dans un `ViewPager`.

## HomeActivity

Affiche les fragments `GamesFragment` et `ChartsFragment` dans un `ViewPager`.

## 4.2. Fragments

### ChartsFragment

Affiche un graphique en barres grâce à l'objet `BarChart` de la bibliotèque `MPAndroidChart` où chaque barre correspond au score total du joueur pour une partie.

### GamesFragment

Affiche la liste des parties du joueur (date, lieu et son score total) dans un `RecyclerView` grâce à l'adapeur `GamesAdapter`. Permet de commencer une nouvelle partie grâce à un `MaterialButton`.

### MyScoresFragment

Reçoit le code de la partie et affiche une liste qui contient le numéro du trou ainsi qu'un `EditText` qui permet de rentrer le score pour ce trou grâce à l'adapteur `ScoresAdapteur`.

### ScoresFragment

Reçoit le code de la partie et affiche les scores de chaque joueur dans un TableLayout. Raffraîchit les scores toutes les 5 secondes.

## 4.3. Adapteurs

### CoursesAdapter

Lie le `RecyclerView` à une liste de `Course`, affiche le nombre de trou, le nom de la course et sa description. Il prend en paramètre un objet `OnCourseClickListener` qui permet de récupérer l'objet `Course` cliqué.

### GamesAdapter

Lie le `RecyclerView` à une liste de `Game`, affiche la date, le lieu et le score total du joueur. Il prend en paramètre un objet `OnGameClickListener` qui permet de récupérer l'objet `Games` cliqué.

## MinigolfsAdapter

Lie le `RecyclerView` à une liste de `Minigolf`, affiche le nom du minigolf et son adresse. Il prend en paramètre un objet `OnMinigolfClickListener` qui permet de récupérer l'object `Minigolf` cliqué.

## ScoresAdapter

Lie le `RecyclerView` à une liste de `Score`, affiche le numéro du trou ainsi que son score qui peut être modifié. Il prend en paramètre un objet `ScoreAdapterListener` qui permet de récupérer l'object `Score` lorsque un `EditText` est modifié et aussi de savoir si tous les `EditText` ont été remplis.

# 5. Communication

Les données distants sont exposées par une API REST. Cette API est consommée par le **site web** et **l'application android**. Cette partie décrit l'utilisation des différentes entitées qui traitent la communication et la gestion des données fournies aux ViewModels.

## 5.1. Repositories

Un repository a un rôle de médiateur entre le stockage local et le serveur (api). Il permet de vérifié que les données distantes correspondent à celles mises en cache localement. Le repostory fournit une unique source de données pour les ViewModels en créant une abstraction. Quand un ViewModel a besoin d'informations, il appelle le repository qui décide qu'elle donnée lui retourner. Le ViewModel n'a pas besoin de connaître la source de ses données ou comment elles sont gérés. Dans ce projet, les repositories sont sous formes de singleton. Ils appellent les différents stores qui correspondent à leur besoins (par exemple, le repository `UserRepository` appelle des méthodes du store `UserStore`).

## 5.2. Stores

Le but d'un store est de fournir des données qui seront exposées par le repository. Il est en charge des requêtes HTTP, de la désérialisation du résultat des requêtes ainsi que leur encapsulation dans un modèle et finalement du prétraitement des données. Au démarrage de l'application, il fait une demande au serveur pour récupérer les données nécessaires. Ensuite, la réponse sous forme de string (json stringifié) est désérialisé et sert à instancier un le modèle correspondant. Et finalement, ces objets sont retournés comme LiveData. Il est en mesure de faire des requêtes asynchones ou bloquantes en fonction du cas. Certaines requête comme l'authentification doit être bloquant pour pouvoir récupérer la validation du serveur et assurer que les permissions nécessaires sont accordées. Les requêtes qui n'a pas besoin d'une réponse immédiate sont asynchrone et une fois les objets observés par LiveData modifiés, l'état de l'application est propagé à tous les composants utilisant ces données. Pour ce qui concerne la communication avec le serveur web, la librairie FUEL est utilisée. La désérialisation quant à elle est assurée par GSON.

### 5.3. Modèles

Un modèle est une classe qui se charge de la représentation interne à l'application d'une entité de la base de données (backend). Une instance d'un modèle contient l'état d'une entrée de la base de donnée. Si une instance est modifiée par un des composant de l'application, son état est non seulement imédiatement propagé à toute l'application mais également au serveur grace à l'architecture MVVM. Un modèle encapsule uniquement des données et aucune logique métier. La seul logique dans nos modèles concerne la classe de désérialisation. L'utilisation de `Parcelable` permet de renseigner la désérialisation en exprimant les types de données des différents champs.

## 6. Problèmes et solutions

Aucune connaissance préliminaire d'Android et de Kotlin. Ne pas connaitre le framework n'est pas un soucis mais ne pas connaitre le langage qui l'utilise rend les tentatives peu productives. En effet, comment faire la différence entre les idiomes de Kotlin (qui vu son caractère très moderne et inspiré de tous les autres langages est justement très coloré) et des constructions spécifiques à Android...

Nous avons donc décidés de nous concentrer dans un premier temps sur Kotlin et pour nous aider, nous avons demandé à monsieur Chèvre si il était possible de faire le cours de JEE sur Kotlin, ce qui a été accepté et a permis d'ajouter une dose de pratique aux nombreux tutoriels que nous avons trouvé sur le net. Par la suite, au vue de la quantité astronomique de tutoriels que possède Android, nous avons juste fait le plus possible de petits projets isolés pour tester un maximum de mécanismes d'Android.

---

Le changement de projet en milieu de semestre n'a pas aidé pour la planification. Dans un premier temps il a été nécessaire de se concentrer sur la partie Web du projet pour assurer la deadline avant de pouvoir se concentrer sur la partie Android.

Le point positif de cette façon de faire séquentille a été le fait que la mise en place du backend et de l'API nous on permis de structurer le projet. En effet, une fois le travail commencé sur Android, il nous suffisait de respecter le contrat fixé avec l'API.

---

L'utilisation des meilleurs pratiques et des derniers composants architecturaux conseillés par Google (LiveData, ViewModel et même si ils ne sont plus si récents que ça, les Fragments et les RecylerView) le tout avec un pattern architectural qui nous était exotique ont rendu la mise en place de la structure longue et complexe.

C'est la raison pour laquelle Room a été mis de coté. En effet, malgré son utilité évidente, on pouvait s'en passer de part l'utilisation d'une permanence distante. Cela dit ce fut une fausse bonne idée basée sur l'apparente complexité de sa mise en place dans des petits projets isolés. Avec un système MVVM en place ça n'aurait pas été la même chose. En effet, il s'intègre parfaitement dans notre stack de technologies et de rapides tentatives nous ont montré qu'en peu de temps nous aurions pu avoir un système de permanence local qui aurait pu nous éviter quelques bugs de la version rendue (crash éventuel lors du changement d'utilisateur et le crash lors du redémarage d'une partie alors que la précédente n'est pas terminée).

# 7. Critique et conclusion

La gestion du temps en cette année particulièrement chargée mélangé à la totale méconnaissance des technologies du secteur ont rendu la planification très compliquée. Comment faire pour découper efficacement les tâches ou estimer leur durée alors qu'on a aucune idée de ce que sont les bonnes pratiques et les architectures qui s'utilisent dans ce secteur.

La partie intéressante a été les stratégies adoptées pour palier à ce problème. La construction d'un savoir faire en effectuant une multitude de petits projets isolés s'est révélé être un très bon tampon. Tous ces minitutoriels standalone qui visait à chaque fois un élément spécifique de l'écosystème Android étaient autant d'aperçu de ce qui se fait dans le domaine en plus de nous créér une banque de code dont on a pu à s'inspirer par la suite.

De plus, c'est probablement cette façon de faire qui nous a permis d'encaisser le changement de projet en plein milieu de semestre. Fondamentalement tout ce qui avait été fait avant à l'exception de la planification du premier projet n'était absolument pas perdu.

Au final, nous sommes contents de notre résultat même si la frustration de ne pas pouvoir profiter du caractère très maintenable que nous offre notre architecture pour ajouter tous les petits détails qui rendrait l'application publiable et ainsi avoir un aperçu de toutes les facettes du développement Android.

En parallèle de cette aventure Android, l'expérience Kotlin fut très interessante également. Nous sommes tous les trois sous son charme et il est certain que nous allons faire en sorte de pouvoir continuer à l'utiliser par la suite.