Helbet

Dans le cadre du cours de programmation mobile I

Table des matières

[Description 2](#_Toc132236365)

[Application de paris sportifs fictifs de football 2](#_Toc132236366)

[Liste de fonctionnalités 2](#_Toc132236367)

[Création de compte 2](#_Toc132236368)

[Système d'abonnement aux clubs 2](#_Toc132236369)

[Paris sur le résultat d'un match 2](#_Toc132236370)

[Système d'alertes 2](#_Toc132236371)

[Historique & analyse des paris 3](#_Toc132236372)

[Scénario 3](#_Toc132236373)

[Informations supplémentaires 3](#_Toc132236374)

[Application similaire 3](#_Toc132236375)

[API 4](#_Toc132236376)

[Informations 4](#_Toc132236377)

[Données requises 4](#_Toc132236378)

[Requêtes 4](#_Toc132236379)

[Récupération Championnats 4](#_Toc132236380)

[Développement 5](#_Toc132236381)

[Login – Register & Realtime database 5](#_Toc132236382)

[Requêtes API 6](#_Toc132236383)

[Classes 7](#_Toc132236384)

[User 7](#_Toc132236385)

[Classe DBModel - abstract 7](#_Toc132236386)

[Classe DBManager 7](#_Toc132236387)

[Classe APIConfig 9](#_Toc132236388)

[Classe APIManager 9](#_Toc132236389)

# Description

## Application de paris sportifs fictifs de football

L’utilisateur devra posséder un compte. Il pourra parier grâce à des jetons virtuels fictifs sur le résultat des matchs de football des grandes ligues nationales européennes[[1]](#footnote-1). Il peut suivre des clubs. Sur la première page sont d’abord affichés les matchs à venir et ensuite, les matchs à venir des équipes suivies. Il sera notifié des matchs auxquels les équipes qu’il suit participent et pourra localiser l’emplacement d’un match sur une carte. Enfin, avec ses jetons, un utilisateur peut investir sur une équipe. Investir sur équipe signifie que le joueur envoie une quantité de jetons à l'équipe. Le dernier jour de chaque mois, l'équipe ayant amassé grâce à ses fans, le plus de jetons, est proclamée équipe du mois jusqu'au mois suivant.

## Liste de fonctionnalités

### Création de compte

A l'aide de son adresse email et d'un mot de passe, un visiteur pourra créer un compte auquel il pourra se connecter par la suite.

<https://firebase.google.com/docs/auth/android/start#java>

### Système d'abonnement aux clubs

L'utilisateur a la possibilité de suivre des clubs. Les matchs auxquels les équipes suivies participent sont mis en évidence par rapport aux autres. De plus, il recevra une notification peu de temps avant le match pour lui rappeler de placer un paris s'il ne l'a pas encore fait.

Voir fonctionnalité annexe "Système d'alertes"

### Paris sur le résultat d'un match

L'utilisateur a la possibilité sur l'écran principal de parier un montant sur le résultat du temps réglementaire d'un match (victoire d'une équipe ou de l'autre ou match nul).

### Système d'alertes

Des alertes (notifications push) sont envoyées à l'utilisateur pour l'avertir :

* Des matchs prévus du jour
* Qu'une de ses équipes suivies a un match imminent
* Du résultat d'un paris (gagnant ou perdant)

### Historique & analyse des paris

L'utilisateur pourra consulter l'historique de ses paris et aura accès à des outils d'analyse encore à définir.

## Scénario

Tartempion crée un compte sur la plateforme. Etant fan du Real Madrid, Tartempion commence à suivre le club sur la plateforme. Le jour du match Real Madrid contre Liverpool, une notification l'avertit de la rencontre. Sur la vue du match, il peut placer une mise sur la victoire de son club préféré. Si son paris est gagnant, il recevra sa mise multipliée par la cote. Sinon, il perdra simplement sa mise.

## Informations supplémentaires

* Les matchs auxquels participent les équipes non gérées par la plateforme ne seront pas listés.

## Application similaire

Omada :

* [Site Web](https://www.omadagame.com/en/home)
* [Google Play](https://play.google.com/store/apps/details?id=com.omada.social&hl=fr&gl=US&pli=1)

# API

Avant de commencer le projet, je dois m'assurer qu'il existe une API pour me fournir les informations nécessaires.

## Informations

|  |  |
| --- | --- |
| Nom | API Football |
| Site Web | https://www.api-football.com/ |
| Headers | 'X-rapidAPI-Key':'df9daf891fmsh4f0f0483c2d1a31p1321dajsn6b919996842a' 'X-RapidAPI-Host':'api-football-v1.p.rapidapi.com' |

## Données requises

* Championnats :
  + Liga Santander (Espagne)
  + Premier League (Angleterre)
  + Serie A (Italie)
  + Bundesliga (Allemagne)
  + [ Ligue 1 (France) ]
  + [ Jupiler Pro League (Belgique) ]
  + [ Eredivisie (Pays-Bas) ]
* Les équipes de ces championnats
* Les matchs prévus dans ces championnats
* Les cotes de ces matchs
* Les résultats de ces matchs
* [ Les match prévus de certaines autres compétitions européennes ]

## Requêtes

J'effectue ces tests grâce au logiciel *Postman* qui me permet de sauvegarder les requêtes.

### Récupération Championnats

<https://api-football-v1.p.rapidapi.com/v3/leagues?current=true&type=league>

|  |  |
| --- | --- |
| Liga Santander | 140 |
| Premier League | 39 |
| Serie A | 135 |
| Bundesliga | 78 |
| Ligue 1 | 61 |
| Jupiler Pro League | 144 |
| Eredivisie | 88 |

# Développement

## Login – Register & Realtime database

### Méthodologie

Pour l'enregistrement et la connexion à un compte, j'utilise le système d'authentification du service **Firebase**. Pour ce faire, j'exploite deux activités :

* **RegisterActivity** qui se charge de créer un compte à un nouvel utilisateur avec les informations fournies (email et mot de passe).
  + **onCreate()** : récupération des variables relatives au service **Firebase** et aux vues dans les variables grâce à **findViewById** :

auth = AuthManager.*getInstance*();  
db = DBManager.*getInstance*();  
  
back = findViewById(R.id.*back\_activity*); // retour arrière  
emailInput = findViewById(R.id.*input\_email*);  
pswdInput = findViewById(R.id.*input\_password*);  
progressBar = findViewById(R.id.*progressBar*); // chargement enregistrement   
submit = findViewById(R.id.*button\_submit*);

* + **onStart()** :

1. Vérification de l'état de connexion de l'utilisateur (grâce à la méthode **isAuthenticated()** de l'objet **auth** ; instance de la classe **AuthManager** qui sera abordée plus bas) :

if (auth.isAuthenticated()) {  
 Intent intent = new Intent(this, MainActivity.class);  
 startActivity(intent);  
 finish();  
}

1. Je désactive le bouton de validation des données **submit** (car je veux qu'il ne soit cliquable que lorsque l'utilisateur a rentré un nom d'utilisateur et un mot de passe corrects :

submit.setEnabled(false);

Ensuite, je crée une instance l'interface **AuthFormTextWatcher** (interface propre au projet héritant de **TextWatcher**) dont je surcharge la méthode **afterTextChange(Editable e)** avec un code qui ne rend le bouton **submit** cliquable que si :

* + - * La valeur contenue dans le champ "email" **emailInput** matche l'expression régulière respective générée par ChatGpt ([lien du prompt](https://pastebin.com/K8vmYfwb)).
      * Idem pour le champ "mot de passe" **pswdInput**.

Cette instance d'interface sera ensuite passée en paramètre **TextWatcher** à la méthode **addTextChangedListener()** des vues **emailInput** et **pswdInput**.

1. Ajout d'un **OnClickListener** au bouton **submit** pour qu'une fois cliqué, un utilisateur soit créé dans la partie authentification de **Firebase**

auth.createUser(email, pswd, authTask -> {

avec l'email et le mot de passe introduits et qu'une instance d'objet **User** respective soit sauvée dans la référence des utilisateurs de la **Realtime Database**. Il est important de noter que si l'instance pour quelle que raison que ce soit n'est pas correctement sauvée dans la référence, l'utilisateur côté authentification sera supprimé grâce à la variable de retour **authTask** qui représente le résultat de sa sauvegarde et l'activité sera relancée lui permettant de retenter l'opération. Sinon un **Toast** alertera l'utilisateur du succès de la création de son compte et il sera redirigé vers l'activité de connexion, **loginActivity** :

if (dbTask.isSuccessful()) {  
 Toast.*makeText*(this, "Compte créé avec succès !", Toast.*LENGTH\_LONG*);  
 Intent intent = new Intent(this, LoginActivity.class);  
 startActivity(intent);  
 finish();  
} else {  
 authTask.getResult().getUser().delete();

Toast.*makeText*(this, "Erreur !", Toast.*LENGTH\_LONG*);  
 recreate();  
}

* **LoginActivity** qui se charge de récupérer et valider les informations de connexion rentrées par l'utilisateur (email & mot de passe) dans le but de le connecter à son compte et de récupérer ses informations de la base de données.
  + **onCreate()** :

Pour ces deux activités je vérifie d'abord que l'utilisateur ne soit pas déjà connecté dans la méthode **onCreate()**.

### Difficultés techniques

suivi d'abord le tutoriel fournit par mon professeur (lien [ici](https://www.youtube.com/watch?v=WykjSz9K05A)) mais au moment d'enregistrer l'utilisateur dans la **Realtime Database**, rien ne se passe et un message d'avertissement (et non d'erreur) me signale que la localisation de ma base de donnée n'est pas correcte voir *Challenges techniques > C1 – Realtime Database* pour la résolution du problème.

### Apprentissages

Cycle de vie des activités

### Dépendances

**Services Firebase :**

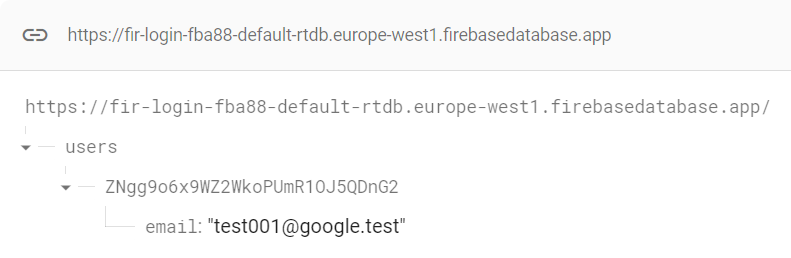
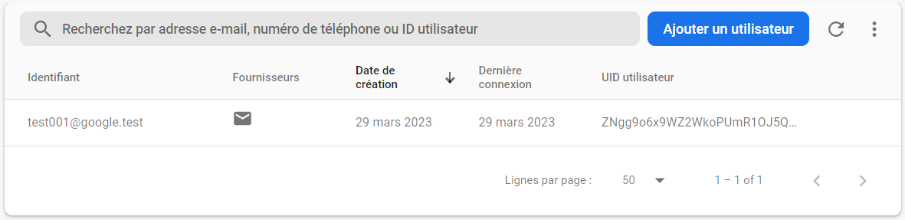
* **Authentification** représenté par la classe ***FirebaseAuthentication***.
* **Base de données en temps réel** représenté par la classe ***FirebaseDatabase***.

**Classe User**

La référence de la clé "users" dans la Realtime Database est une liste d'utilisateurs conceptualisés par la classe User :

public class User {  
 private static DatabaseReference *mDatabase* = FirebaseDatabase.*getInstance*().getReference();  
  
 public String email;  
  
 public User() {  
 // Default constructor required for calls to DataSnapshot.getValue(User.class)  
 }  
  
 public User(String email) {  
 this.email = email;  
 }  
}

Pour l'instant, elle n'est composée que d'un attribut **email** qui se réfère à l'email entré lors de l'inscription. Quant au mot de passe, il est enregistré dans la partie **Authentication**. L'**UID** (identifiant unique) de l'utilisateur est utilisé comme clé, ce qui veut dire qu'on peut directement lier les données de la partie authentification et celles de la Realtime Database.



## Requêtes API

Pour faire les requêtes API nécessaires à mon application, j'utilise le module **Volley** avec lequel j'ai pu me familiariser lors d'une activité autonome proposée par mon professeur.

Mon but ici est :

1. Récupérer certaines données de l'API et les stocker dans la Realtime Database
2. Afficher ces données dans un **RecyclerView** (servant à les lister)

# Classes

### Classe DBModel - abstract

Firebase stocke les données sous forme de nœuds accessibles grâce à leur clé. J'ai décidé de faire hériter toutes les classes dont les instances seront stockées dans la Realtime Database d'une classe abstraite **DBModel** dans le but d' :

* Implémenter une interface **DBExclude** qui intègre :
  + Une méthode **toMap()** qui retourne une Map<String, Object> contenant les attributs de l'instance de l'objet qui l'implémente et leur valeur en excluant ceux étant marqué de l'annotation **@DbExcludeField**..
  + Une annotation **@DBExcludeField** qui informe la méthode **toMap()** que le champ marqué n'est pas à prendre en compte dans le résultat.

if (field.isAnnotationPresent(DBExcludeField.class)) {  
 continue;  
}

* + Cette interface et son utilisation m'ont été respectivement généré et expliqué par ChatGpt ([lien du prompt](https://pastebin.com/xi9Y0H6d)).
* Automatiquement attribuer la clé du nœud de la valeur à l'attribut id de l'instance la représentant (sans pour autant qu'elle ait besoin d'être sauvegardé additionnellement à la clé).

#### Attributs

|  |  |
| --- | --- |
| @DBEcludeField String id | Id du modèle faisant directement référence à la clé du nœud respectif dans la Realtime Database |

#### Méthodes publiques

* Constructeur par défaut
* Getters et Setters
  + { id }

#### Conséquences de l'implémentation

* Modification de la classe User (je le fais hériter de **DBModel**:

public class User extends DBModel{

### Classe AuthManager

Sert d'API propre au projet pour le service d'authentification **Firebase**. Elle fonctionne suivant le paterne **Singleton** car est partagée par toutes les activités (pour vérifier l'état de connexion de l'utilisateur par exemple).

### Classe DBManager

Sert d'API propre au projet pour la Realtime Database. Elle fonctionne en suivant le paterne **Singleton** car est partagée par toutes les activités (dont la récupération d'informations depuis la **RTDB** est nécessaire).

#### Attributs

|  |  |
| --- | --- |
| DBManager singleton | Le singleton de l'objet |
| FirebaseDatabase dbRealtime | L'instance faisant le lien direct à la Realtime Database du projet |

#### méthodes publiques

* **getInstance()** 🡪Obtention de l'instance.

Le mot-clé synchronized (associé à getInstance()) sert à empêcher deux threads d'accéder en même temps à la méthode.

##### storeObject(T o, String pathRef)

|  |  |
| --- | --- |
| <T extends DBModel> | Contrainte de type qui s'assure que le type générique T soit une sous-classe de **DBModel** |

Stocke l'objet **o** dans un nœud enfant dont la clé vaudra l'id de l'objet dans la référence de la Realtime Database correspondante à **pathref**.

##### fetchRef(String pathRef, Class<T> \_class, OnFetchMultCompleteListener listener)

Récupère toutes les valeurs d'une référence **pathRef** dans une **List** du type spécifié par **\_class** et la passe en paramètre dans la méthode **listener.onFetchMultComplete(List result)** qui sert de callback nécessaire au traitement des données récupérées de manière asynchrone.

##### fetch(String pathRef, String key, Class<T> \_class, OnFetchSingCompleteListener listener)

Récupère la valeur correspondante à la clé **key** dans la référence **pathRef** dans une instance du type spécifié par **\_class** et la passe en paramètre dans la méthode **listener.onFetchMultComplete(T result)** pour les mêmes raisons.

##### fetchFiltered(String pathRef, Map<String, String> params, Class<T> \_class, OnMultSingCompleteListener listener)

Récupère les valeurs des enfants qui contiennent les paires clé-valeur contenues dans **params** dans la référence **pathRef** dans une **List** du type spécifié par **\_class** et la passe en paramètre dans la méthode **listener.onFetchMultComplete(T result)** pour les mêmes raisons.

### Classe APIConfig

Pour pouvoir me connecter à l'API et effectuer des requêtes, je vais avoir besoin de ma clé d'identification. Cependant, mon projet est répertorié sur GitHub en publique. Je vais donc stocker cette information en tant que constante dans la classe dont le fichier sera référencé dans le **.gitignore** afin que la clé secrète ne se retrouve pas publiquement sur GitHub lors d'un push.

#### Constantes (attributs)

* L'URL de l'API : ***API\_URL***
* Les en-têtes nécessaires sous forme de HashMap ***HEADERS*** :
  + Clé "***X-RapidAPI-Key***"
  + Hôte "***X-RapidAPI-Host***"
* Les IDs des ligues concernées par l'application (voir [ce tableau](#_Récupération_Championnats)) sous forme d'Array : ***LEAGUE\_IDS***

### Classe APIManager

La classe APIManager me sert d'interface pour récupérer certaines données depuis l'API.

1. Les ligues nationales sont composées de clubs d'un même pays et se déroulent sur une saison. [↑](#footnote-ref-1)