

Movies App

Jeremy Favre - Steven Liatti

Janvier 2020

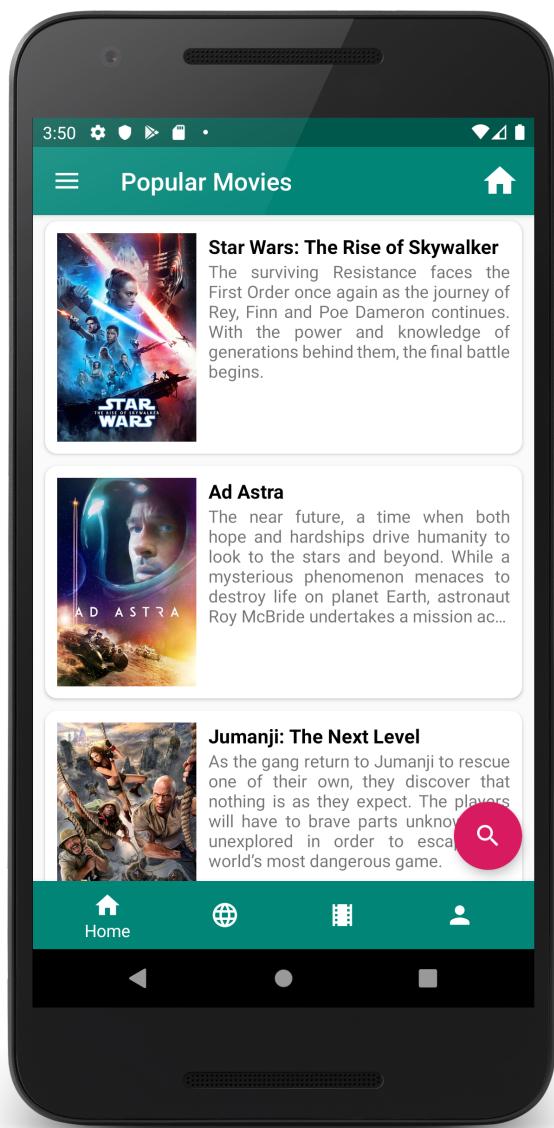


Table des matières

Table des matières	2
Table des figures	3
Table des listings de code source	4
Conventions typographiques	4
1 Introduction	5
1.1 Buts	5
1.2 Motivations	5
1.3 Méthodologie de travail	5
2 Conception et analyse	6
2.1 Films tendance	6
2.2 Détails d'un film	6
2.3 Recherche	6
2.4 "Peoples"	7
2.5 Détails d'un "people"	8
2.6 Mes films	8
2.7 Réseau d'amis	8
2.8 Authentification	8
2.8.1 Connexion	8
2.8.2 Enregistrement	8
2.9 Navigation	8
3 Implémentation	9
3.1 Architecture globale	9
3.2 APIs	9
3.2.1 API TMDb	10
3.2.2 API REST Backend	11
3.2.3 API YouTube	13
3.3 Fragments	14
3.4 Single Activity	14
3.5 Navigation component	15
3.5.1 Arguments	16
3.6 Volley	17
3.6.1 Généralités	17
3.6.2 Asynchronisme	17
3.6.3 Améliorations	18
3.7 Listes d'items et grilles	19

3.8 Toolbar	19
3.9 Drawer	19
3.10 Bottom tabs	20
3.11 View pager	21
3.12 Recherche	23
3.13 Shared preferences	24
3.14 FAB	25
3.15 Generic adapter	25
4 Conclusion	27
4.1 Bilan personnel	27
4.2 Problèmes rencontrés	27
4.3 Améliorations possibles	27
5 Références	28

Table des figures

1 Maquettes à main levée des vue de l'application	7
2 Usage des APIs	9
3 Architecture globale de l'application	10
4 Listes des films et des acteurs tendance	12
5 Détails d'un film et d'un acteur	13
6 Bandes-annonces dans les détails d'un film	14
7 Représentation d'une unique activité Android	15
8 Graphe de navigation de notre application	16
9 Différences entre un fil d'exécution synchrone et asynchrone	18
10 Onglets de l'application	20
11 Grille de films similaires dans les détails d'un film	20
12 Drawer de l'application	21
13 Appréciations des films dans l'application	22
14 Réseau social dans l'application	23
15 Vue de la recherche dans l'application	24
16 FAB de l'application	26
17 Principes de l'adaptateur générique	26

Table des listings de code source

1	Exemple de données retournées par TMDb	11
2	Exemple de navigation	16
3	Arguments du graphe de navigation en Kotlin	17
4	Usage de la librairie Volley	17
5	Améliorations usage de la librairie Volley avec <i>callback</i>	18
6	Exemple d'utilisation de VolleyRequestController	19
7	Création des <i>shared preferences</i>	25
8	Vérification des <i>shared preferences</i>	25
9	Destruction des <i>shared preferences</i>	25

Conventions typographiques

Lors de la rédaction de ce document, les conventions typographiques ci-dessous ont été adoptées.

- Tous les mots empruntés à la langue anglaise ou latine ont été écrits en *italique*.
- Toute référence à un nom de fichier (ou répertoire), un chemin d'accès, une utilisation de paramètre, variable, commande utilisable par l'utilisateur, ou extrait de code source est écrite avec une police d'écriture à chasse fixe.
- Tout extrait de fichier ou de code est écrit selon le format suivant :

```

1 fun main() {
2     println("Hello World!")
3 }
```

Acronymes

API *Application Programming Interface*, Interface de programmation : services offerts par un programme producteur à d'autres programmes consommateurs. 5, 8, 9, 12, 16

JSON *JavaScript Object Notation*, Format d'échange de données léger, facile à lire et écrire par les humains et les machines. 9, 16

JVM *Java Virtual Machine*, Exécute le bytecode Java sur différents systèmes d'exploitation. 10

XML *Extensible Markup Language*, Langage extensible de description de données. 13, 14, 18, 19, 22

1 Introduction

1.1 Buts

De nos jours, les médias sont à la portée de tous et accessibles très facilement notamment, grâce aux plateformes telle que Netflix, streaming et bien d'autres encore. Il est donc difficile d'effectuer un choix parmi ces milliers de films mis à disposition. C'est là qu'entre en jeu notre mini-projet que nous avons réalisé dans le cadre du cours de Systèmes d'exploitation mobiles et applications. L'objectif est de développer une application dédiée à la plateforme Android qui propose à ses utilisateurs divers services, tels que la liste des films à la une, obtenir les films similaires à un film, les acteurs concernés et la possibilité de rechercher un film en particulier. Pour rester dans l'ère du temps, un aspect social est également présent au sein de cette application, l'utilisateur pourra renseigner son appréciation personnelle à propos d'un film, mais également de pouvoir suivre d'autres utilisateurs afin de connaître leurs appréciations sur différents films qu'ils ont visionnés. La forme "nous" est utilisée tout au long de ce rapport étant donné que ce projet est réalisé par binôme.

1.2 Motivations

Le périmètre de ce projet semblait parfaitement adapté pour exploiter la majorité des techniques et méthodes vues durant le cours de développement d'applications mobiles. Ce projet est une formidable occasion pour relier la mise en pratique des connaissances acquises en cours et notre passion cinéphile commune. Nous sommes également convaincus que notre entourage (ou un réseau plus large) se servirait volontiers d'une telle application.

1.3 Méthodologie de travail

Sur la base d'une analyse préliminaire, nous avons séparé le travail en plusieurs tâches que nous avons assigné à chaque membre du binôme de manière équitable afin d'effectuer le travail en parallèle. Nous avons adopté une pseudo méthode "agile", en factorisant le projet en petites tâches distinctes et en nous fixant des délais pour les réaliser. Le partage du code s'est fait avec git et gitlab. Nous nous sommes servis des *issues* gitlab pour représenter nos tâches et du "board" du projet pour avoir une vision globale du travail accompli (par qui et quand) et du travail restant.

2 Conception et analyse

Nous avons commencé par dessiner les maquettes des vues de notre application afin de se faire une idée de l'aspect visuel de chaque vue et des relations entre elles, comme illustré à la figure 1. Tout cela dans le but de simplifier et clarifier la phase de développement proprement dite. Durant cette phase, nous nous sommes également documentés sur les APIs existantes proposant des services relatifs aux films, nous avons étudié les différentes technologies pour manipuler une API REST et finalement nous avons analysé les composants et méthodologies étudiées en cours qui seraient utiles d'intégrer dans notre application. Les sous-sections suivantes décrivent les fonctionnalités que propose l'application finale.

2.1 Films tendance

Lorsqu'un utilisateur ouvre l'application, la vue principale (*home*) lui affiche une liste des films à la une, classés en fonction de leur popularité. Chaque film affiche ses informations de base (poster, titre, courte description). L'utilisateur a la possibilité de cliquer sur un film afin d'obtenir d'avantage d'informations.

2.2 Détails d'un film

Comme décrit à la section 2.1, si l'utilisateur souhaite en savoir plus sur un film, il peut accéder à cette vue présentant les détails d'un film. Elle comporte les informations détaillées du film tel que le titre, la description complète, la liste des acteurs, la liste des réalisateurs, les différentes bande-annonces, les genres associés et une grille qui contient les films similaires au film concerné. La possibilité de cliquer sur un film de la grille afin de naviguer vers ses détails est également intégrée. Si l'utilisateur souhaite visionner une bande-annonce, il suffit de cliquer dessus afin que le *player* YouTube soit automatiquement lancé avec la vidéo en question. C'est également depuis cette vue qu'il pourra indiquer son appréciation pour le film ("*like*" ou "*dislike*"), appréciation qui sera sauvegardée dans son profil.

2.3 Recherche

Une interface de recherche est mise à disposition. La recherche peut être lancée parmi les films, les acteurs ou les autres utilisateurs de l'application selon le choix de l'utilisateur (bouton "radio" ou mécanisme similaire). Les résultats de la recherche seront présentés sous forme de liste, présentant différentes actions selon le type de recherche qui aura été effectué :

- Si l'utilisateur cherche un film, il obtiendra la liste des films qui correspondent aux critères entrés dans le champs texte de recherche.
- S'il cherche un acteur, il obtiendra une liste d'acteurs avec la possibilité de cliquer dessus (comme pour les films) afin d'afficher plus de détails à leur sujet.

- S'il cherche un autre utilisateur, il aura alors la possibilité de le suivre en l'ajoutant à son réseau d'amis.

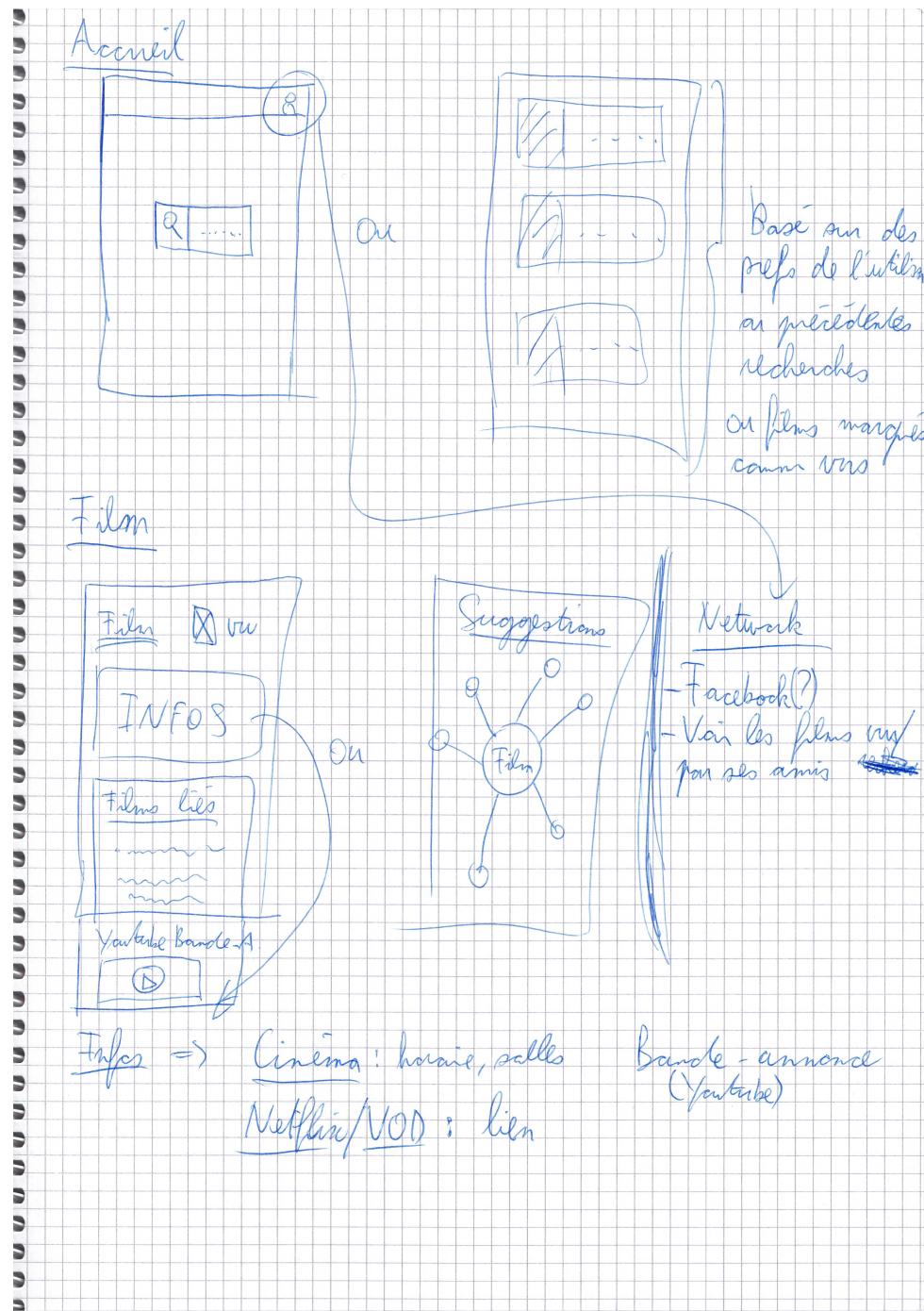


FIGURE 1 – Maquettes à main levée des vue de l'application

2.4 "Peoples"

Au même titre que la liste des films tendance, nous avons une interface proposant la liste des acteurs actuellement populaires avec la possibilité de cliquer sur chacun d'entre eux afin d'obtenir plus d'informations à leur sujet.

2.5 Détails d'un "people"

Comme pour les détails d'un film, nous avons également une vue présentant les détails d'un acteur, elle permet de consulter diverses informations, telles que sa biographie personnelle et la liste des films dans lesquels il a joué.

2.6 Mes films

Si l'utilisateur possède un compte dans l'application, il pourra retrouver sa liste de films appréciés ou non, sous une forme similaire à la liste des films tendance (sous-section 2.1).

2.7 Réseau d'amis

La vue réseau d'amis est disponible uniquement si l'utilisateur détient un compte. Son réseau d'amis sera constitué d'une liste d'utilisateurs le suivant (ses "*followers*") ainsi que d'une liste d'utilisateurs auxquels il est abonné (ses "*following*").

2.8 Authentification

Au sein de l'application, l'authentification est requise pour consulter certaines vues spécifiques, comme le réseau d'amis (2.7) et les films préférés (2.6). Les autres vues de l'application ne requièrent aucune authentification. Ce principe a été mis en place afin de ne pas reproduire un des principaux points faibles des applications actuelles qui est de forcer l'utilisateur à s'authentifier sous peine de ne pas pouvoir utiliser l'application. Notre application demande donc à l'utilisateur de s'authentifier uniquement lorsque cela est nécessaire.

2.8.1 Connexion

La vue connexion comporte simplement deux champs permettant de renseigner le pseudo et le mot de passe afin de s'authentifier.

2.8.2 Enregistrement

Cette vue comporte les champs nécessaires à la création d'un compte, à savoir un pseudo (unique), un email et un mot de passe.

2.9 Navigation

Afin de naviguer dans l'application, l'utilisateur dispose de différents mécanismes tels que les *tabs* de navigation (en bas de l'écran), la barre de navigation qui se situe en haut de l'écran et enfin un menu latéral mettant à disposition les différentes informations relatives à son compte.

3 Implémentation

Dans cette partie concernant l'implémentation, nous allons présenter les différentes technologies, méthodes et composants utilisés afin de mener à bien le développement de notre application. Des captures d'écran illustreront le résultat final.

3.1 Architecture globale

Nous avons architecturé notre application autour du composant de navigation (voir section 3.5), instancié par la *main activity*. En fonction du clic de l'utilisateur, le fragment associé est appelé, qui à son tour fait appel à une vue. Dans le cas des listes (films, *peoples* et utilisateurs), un adaptateur générique est appelé, pour éviter la duplication de code (voir section 3.15). La plupart du temps, les fragments se servent du contrôleur HTTP Volley (voir section 3.6) pour récupérer les données. Les données sont transformées en objets Kotlin à l'aide de *data classes*. La figure 3 illustre le schéma global de l'architecture.

3.2 APIs

Pour échanger les informations requises, notre application interagit avec des APIs externes permettant d'obtenir et persister les données nécessaires aux différentes fonctionnalités proposées, comme illustré à la figure 2.

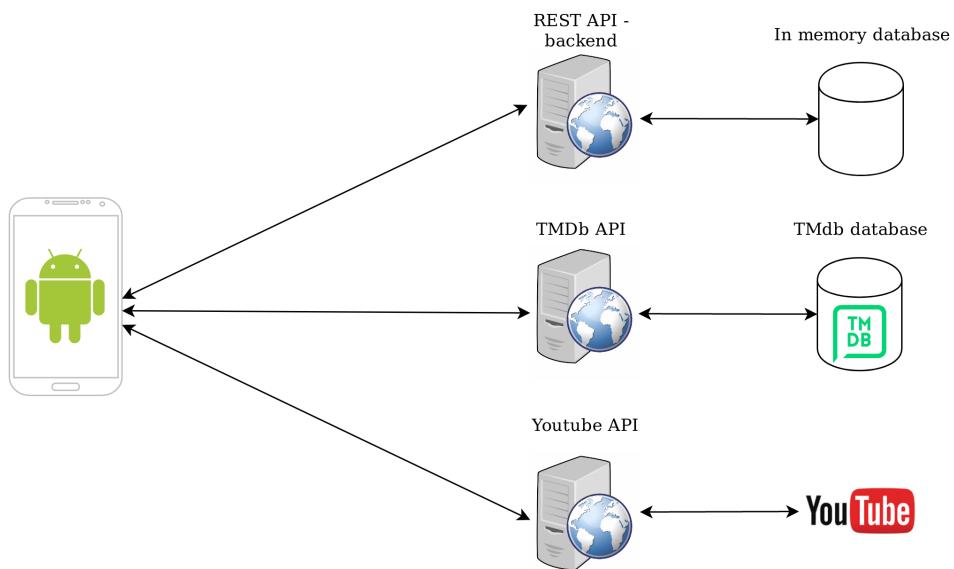


FIGURE 2 – Usage des APIs

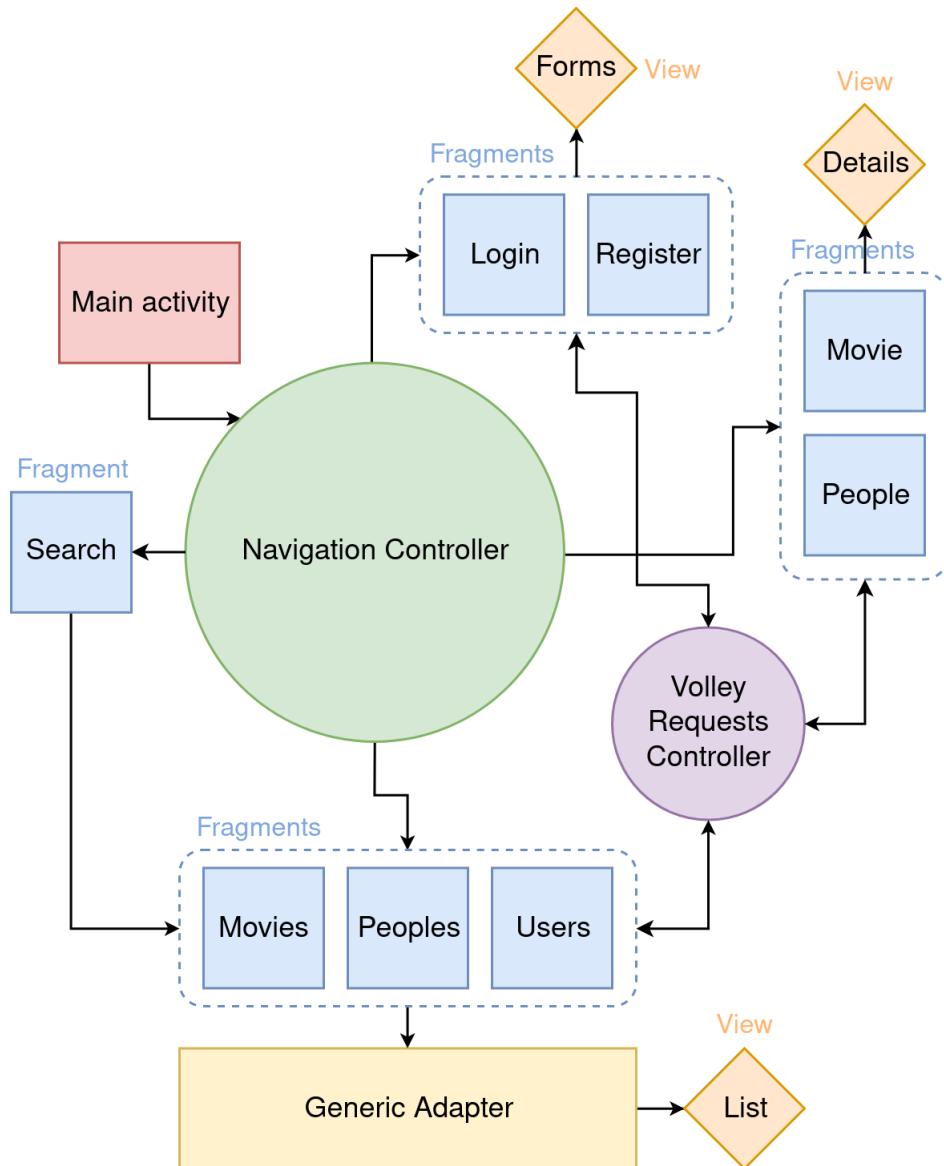


FIGURE 3 – Architecture globale de l'application

3.2.1 API TMDb

L'API principalement utilisée dans notre projet est celle proposée par *The Movie Database* (TMDb) [1] qui offre toutes les informations sur les films et les acteurs. Nous avons également considéré *The Open Movie Database* [2], mais elle offre beaucoup moins de possibilités. Nous interagissons avec TMDb en envoyant des requêtes HTTP en fonctions des besoins, nous recevons en retour les informations au format JSON. Par exemple, la route GET : /discover/movie nous retournera les informations sur les films actuellement populaires, visibles au listing 1 au format JSON. Avec ces différentes informations, nous pouvons construire les listes et les vues de détails des films et des "peoples", comme illustrés sur les figures 4 et 5.

```
1  {
2      "page": 1,
3      "total_results": 10000,
4      "total_pages": 500,
5      "results": [
6          {
7              "popularity": 533.832,
8              "vote_count": 1707,
9              "video": false,
10             "poster_path": "/db32La0ibwEliAmSL2jjDF6oDdj.jpg",
11             "id": 181812,
12             "adult": false,
13             "backdrop_path": "/j0zrELAzFxtMx2I4uDGH0otdfsS.jpg",
14             "original_language": "en",
15             "original_title": "Star Wars: The Rise of Skywalker",
16             "genre_ids": [ 28, 12, 878 ],
17             "title": "Star Wars: The Rise of Skywalker",
18             "vote_average": 6.7,
19             "overview": "The surviving Resistance faces the First Order ...",
20             "release_date": "2019-12-18"
21         },
22         ...
23     ]
```

Listing 1 – Exemple de données retournées par TMDb

3.2.2 API REST Backend

Certaines données de l'application nécessitent d'être persistées, c'est pourquoi nous avons choisi de mettre en place un backend basé sur Akka HTTP [3] en Scala. Le Scala est un langage relativement proche du Kotlin qui fait également usage de la JVM et offre de nombreux atouts comme un typage fort, abstraction et programmation orientée objet. Tout cela avec une syntaxe permettant d'écrire le code de manière simple, concise et déclarative. Pour toutes ces raisons, le choix de ce langage nous semblait judicieux. Les routes mises à disposition par cette API REST nous permettent de persister les informations telles que les comptes des utilisateurs, les relations d'abonnements entre eux et les appréciations des films. Elles permettent également de s'authentifier, s'enregistrer et rechercher un utilisateur.

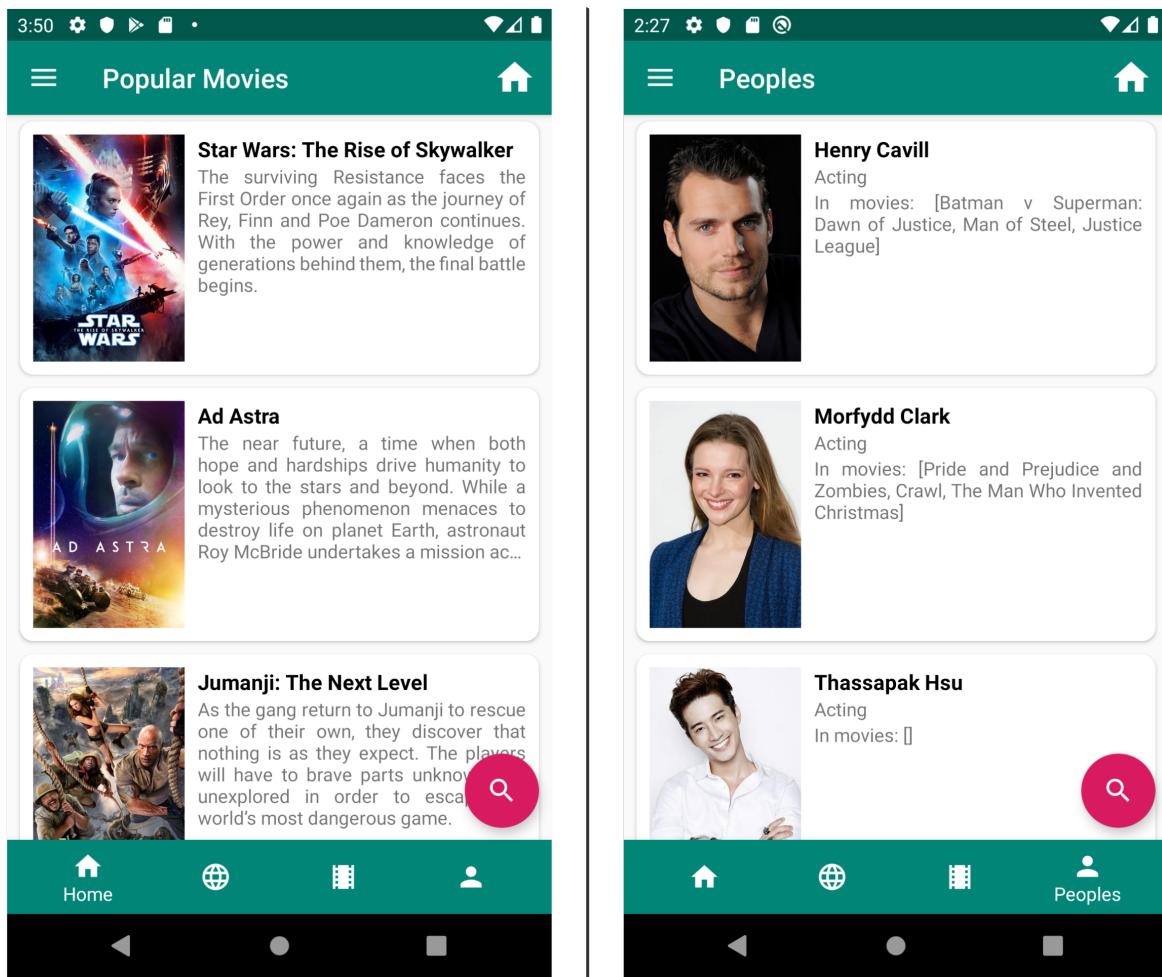


FIGURE 4 – Listes des films et des acteurs tendance

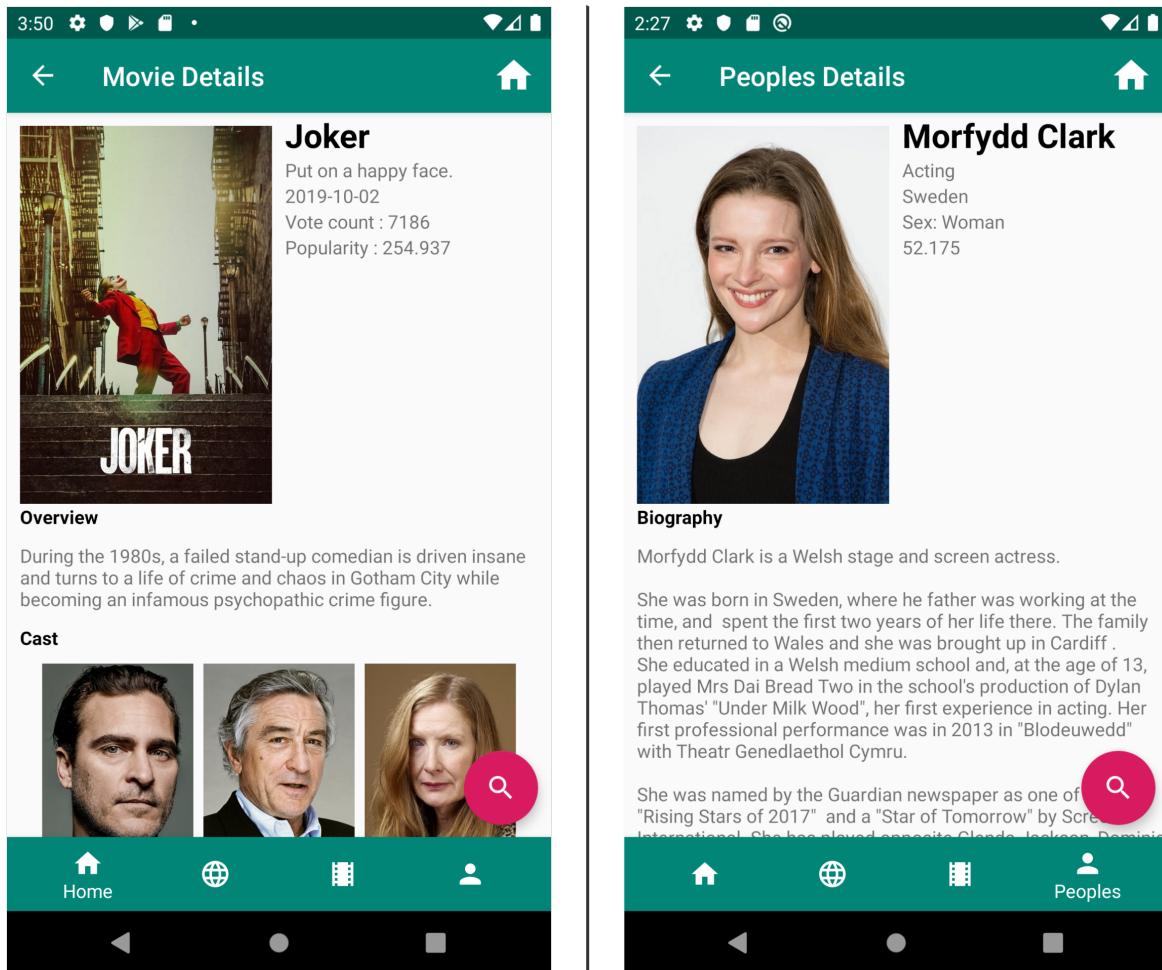


FIGURE 5 – Détails d'un film et d'un acteur

3.2.3 API YouTube

Pour chaque film, TMDb nous offre une liste de vidéos YouTube, comprenant des *trailers* et *teasers* du film. Nous avons alors fait usage de l'API officielle YouTube [4] pour intégrer ces vidéos. Il faut obligatoirement créer une clé API dans son compte Google pour pouvoir utiliser cette API. On peut utiliser cette API globalement de deux manières différentes sur Android :

- **YouTubePlayerView** et **YouTubePlayerFragment** : offrent un plus grand contrôle sur la gestion de l'interface et des contrôles d'une vidéo ainsi que de son cycle de vie au prix de plus de code à écrire et à gérer.
- **YouTubeStandalonePlayer** : la manière plus simple et rapide d'utiliser l'API, utilise l'application officielle YouTube. Le player attend l'identifiant d'une vidéo et la clé API et démarre l'activité de l'application officielle YouTube.

Nous avons décidé d'utiliser la deuxième méthode car nous n'avons pas besoin d'un contrôle fin sur le lancement des vidéos. Pour chaque vidéo obtenue sur la vue des détails

d'un film, nous commençons par récupérer un *thumbnail* de chaque vidéo avec ce lien : [https://img.youtube.com/vi/\\$youtubeId/1.jpg](https://img.youtube.com/vi/$youtubeId/1.jpg). Nous superposons ensuite une icône "play" et ajoutons un listener qui démarre la vidéo sur chaque *thumbnail*. Un **catch** de `ActivityNotFoundException` est inclus avec affichage d'un message dans un "toast" dans le cas où l'utilisateur n'a pas l'application officielle YouTube installée sur son appareil. La figure 6 illustre le rendu.

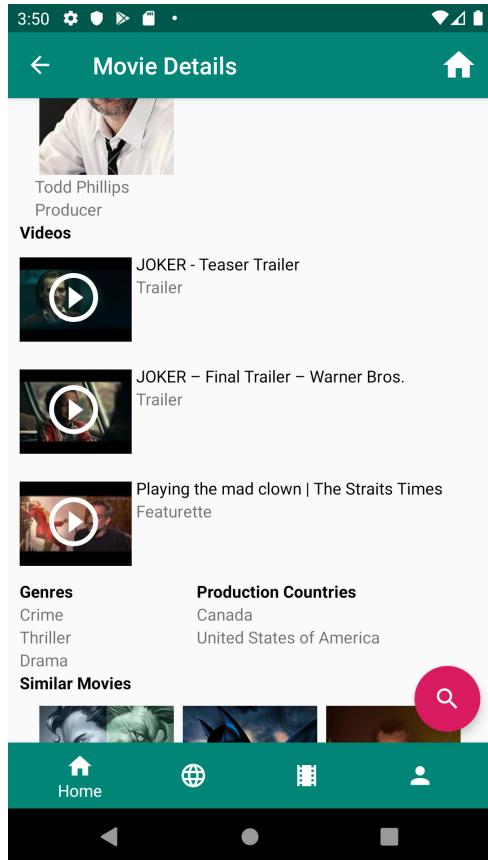


FIGURE 6 – Bandes-annonces dans les détails d'un film

3.3 Fragments

Toutes les vues de l'application sont implémentées à l'aide de fragments. Chaque fragment est associé à un layout (fichier XML) qui décrit son aspect graphique. Le code Kotlin propre au fragment nous permet uniquement de définir une implémentation qui permet l'interaction entre les différents composants graphiques de la vue et la logique métier de l'application. Notre code est ainsi réutilisable dans plusieurs vues différentes.

3.4 Single Activity

Selon le cours et la documentation officielle d'Android [5], l'architecture actuellement conseillée pour le développement d'une application repose sur le principe de n'utiliser qu'une

seule activité composée de fragments, représentant les vues de l'application, illustré en figure 7.

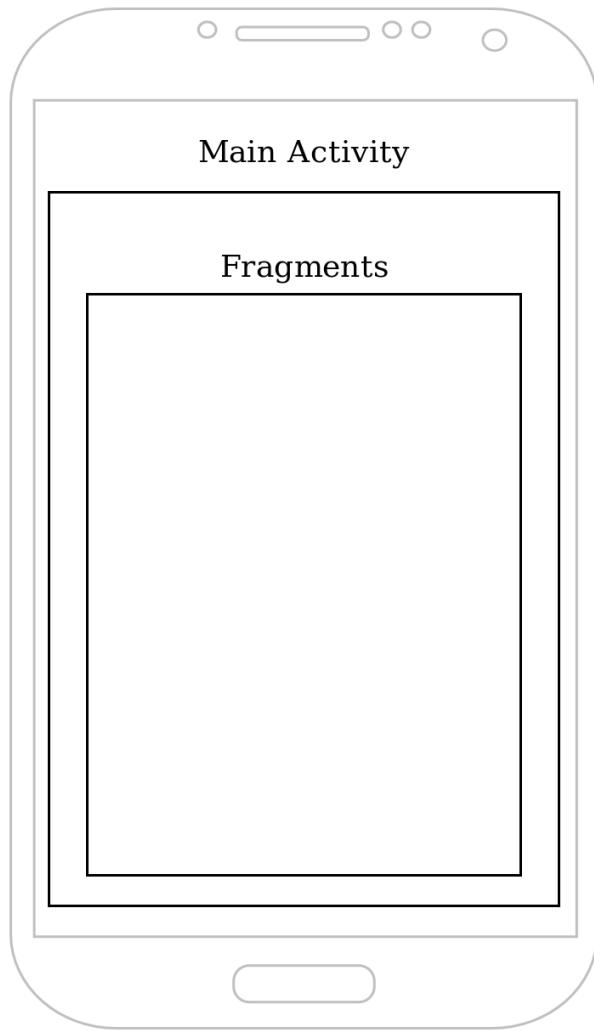


FIGURE 7 – Représentation d'une unique activité Android

3.5 Navigation component

Plusieurs possibilités s'offraient à nous concernant la navigation entre les différents fragments. Nous avons choisi d'expérimenter la nouvelle méthode proposée par la documentation officielle d'Android [5] qui est très intéressante et efficace. Cette méthode repose sur un graphe de navigation qui est éditable graphiquement ou en XML, comme illustré à la figure 8. Les fragments de l'application représentent les noeuds du graphe et les arcs symbolisent les actions de navigation d'un fragment à un autre avec des éventuelles données passées en arguments de ces actions. Ce composant de navigation génère également le code des actions de navigation en fonction de ce qui a été décrit en XML.

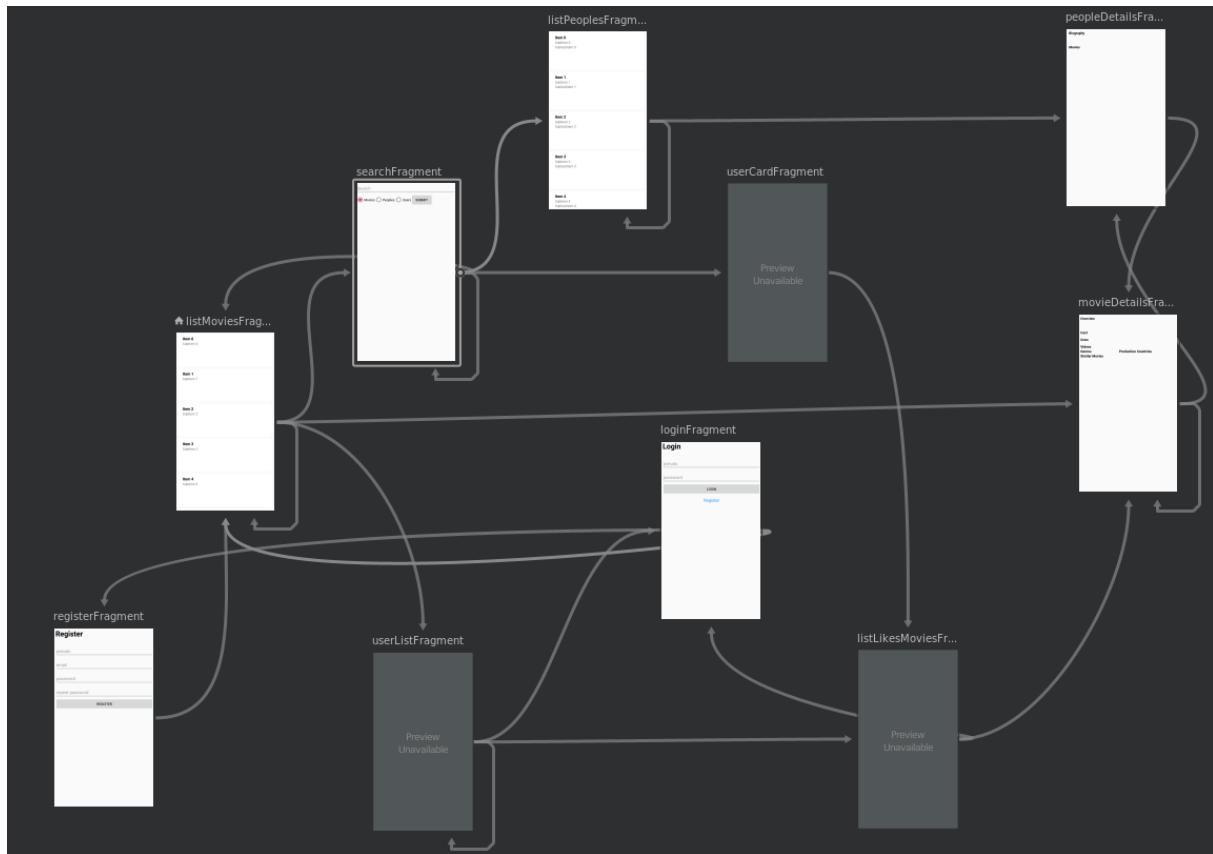


FIGURE 8 – Graphe de navigation de notre application

Un exemple de code qui permet de naviguer de la liste des films vers les détails d'un film en utilisant cette classe générée automatiquement est disponible au listing de code 2.

```

1 view.findNavController()
2     .navigate(ListMoviesFragmentDirections
3         .actionListMoviesFragmentToMovieDetailsFragment(item.id,
4             item.urlImg))

```

Listing 2 – Exemple de navigation

3.5.1 Arguments

Il est également possible de définir des arguments transmissibles entre les différents fragments, la classe générée automatiquement prendra en compte ces derniers. Du côté du fragment qui sera appelé, une méthode très pratique est à disposition, permettant de recevoir ces arguments, comme décrit dans le listing 3.

```

1 private val args: MovieDetailsFragmentArgs by navArgs()
2 movieId = args.id
3 urlImg = args.urlImg

```

Listing 3 – Arguments du graphe de navigation en Kotlin

3.6 Volley

3.6.1 Généralités

Notre application est majoritairement composée d'appels HTTP à diverses APIs. Nous avons utilisé la librairie Android conseillée dans la documentation officielle [5], Volley [6]. Elle supporte nativement les requêtes sur des chaînes de caractères bruts, des images ou du JSON. Le listing 4 décrit un exemple d'utilisation : tout d'abord il faut instancier une queue de requêtes, ensuite définir deux *callbacks* traitant les cas de réussite ou d'échec de la requête, enfin ajouter ces *callbacks* à la queue de requêtes. Volley exécute ensuite les requêtes se trouvant dans la queue de manière asynchrone (sans bloquer le *thread* principal d'affichage) et transparente pour l'utilisateur. Il est conseillé de déclarer une seule instance de Volley avec le *pattern singleton*.

```

1 val textView = findViewById<TextView>(R.id.text)
2 val queue = Volley.newRequestQueue(this)
3 val url = "http://www.google.com"
4 val stringRequest = StringRequest(Request.Method.GET, url,
5     Response.Listener<String> { response ->
6         textView.text = "Response is: ${response.toString()}"
7     },
8     Response.ErrorListener { textView.text = "That didn't work!" })
9 queue.add(stringRequest)

```

Listing 4 – Usage de la librairie Volley

3.6.2 Asynchronisme

Dans la majorité des cas, la logique de l'application requiert d'avoir reçu certaines informations en provenance des APIs avant de pouvoir les afficher sur la vue. Nous avons besoin de garantir que l'intégralité des données soient réceptionnées avant de les afficher, mais tout cela avec la contrainte de ne pas bloquer l'affichage, comme illustré à la figure 9. Nous avons donc utilisé le mécanisme de fonctions de *callback*. Le principe est simple, la fonction de *callback* n'est appelée que lorsque la requête HTTP est effectuée et les données réceptionnées.

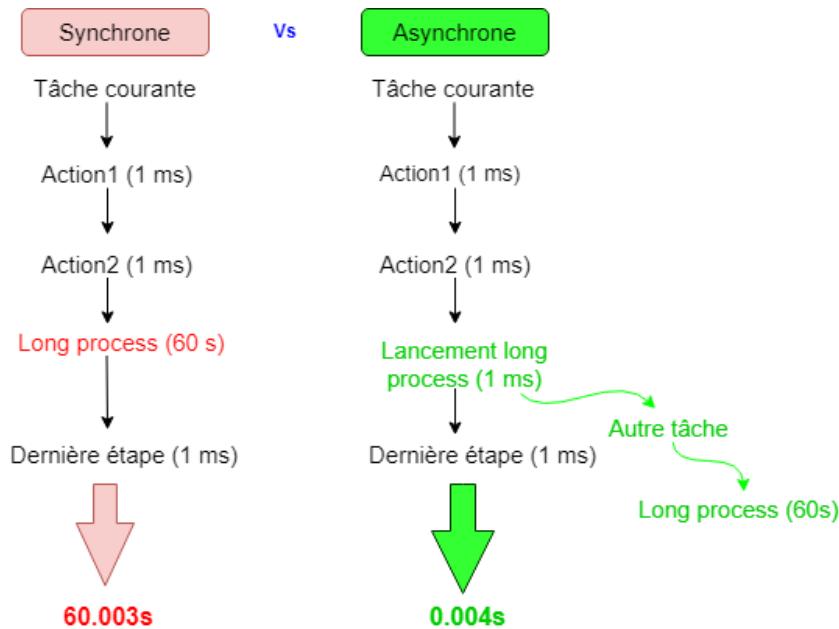


FIGURE 9 – Différences entre un fil d'exécution synchrone et asynchrone

3.6.3 Améliorations

Etant donné le nombre d'appels HTTP important dans notre application, le code devient rapidement "pollué" par ce code long et répétitif (voir listing 4). Nous avons suivi les bonnes pratiques et simplifié cela en créant une classe `VolleyRequestController` offrant les méthodes relatives à tous les appels HTTP requis par notre application. La fonction pour réaliser une requête HTTP GET est visible au listing 5 et son utilisation dans le code au listing 6. Nous avons également activé le cache réseau sur disque pour améliorer les performances d'affichage des vues.

```

1 fun httpGet(URL: String, context: Context, callback:
2     ServerCallback<JSONObject>) {
3     val jsonObjReq = JsonObjectRequest(Request.Method.GET, URL, null,
4         Response.Listener { response -> callback.onSuccess(response) },
5         Response.ErrorListener { error ->
6             Log.println(Log.DEBUG, this.javaClass.name,
7                 "error in httpGet : $error,\n$URL\n$callback")
8         })
9     HttpQueue.getInstance(context).addToRequestQueue(jsonObjReq)
10 }
```

Listing 5 – Améliorations usage de la librairie Volley avec *callback*

```
1 requestController.httpGet(url, context, object :  
2     ServerCallback<JSONObject> {  
3         override fun onSuccess(result: JSONObject) {  
4             // Traitement des données reçues dans result  
5         }  
6     }
```

Listing 6 – Exemple d'utilisation de VolleyRequestController

3.7 Listes d'items et grilles

Les listes présentes dans l'application sont réalisées avec des *recycler view*, comme vu en cours, pour améliorer la fluidité de l'affichage lors du défilement (comparé avec les listes classiques) et pour pouvoir au besoin afficher différents types d'items au sein de la même liste. Nous avons également utilisé un *GridLayout* pour afficher les grilles des films similaires et du *casting*, comme illustré à la figure 11.

3.8 Toolbar

En haut de l'écran se trouve une barre comportant le nom du fragment courant, un bouton "home" (2.1) et un contrôle faisant apparaître le menu latéral.

3.9 Drawer

Le drawer est un menu latéral qui apparaît de gauche à droite de l'écran. Ce menu permet d'afficher les informations relatives au profil de l'utilisateur courant. D'autre part, il propose les actions de connexion, création de compte et déconnexion. Si l'utilisateur est connecté, il pourra aussi naviguer vers les vues des films aimés et du réseau d'amis. Afin de rendre l'interface utilisateur plus conviviale, nous avons implémenté un *drawer*, illustré à la figure 12. Ce *drawer* repose également sur la dernière méthode proposée par la documentation d'Android [5]. L'interface graphique du *drawer* est définie dans le XML à l'aide d'un menu, et initialisée dans la *main activity*. L'identifiant des différents items est nommé selon les identifiants référencés dans le graphe de navigation ce qui permet de naviguer directement vers le bon fragment lors de l'interaction d'un utilisateur avec ces derniers. Pour ce faire, le layout de la *main activity* est de type *DrawerLayout*, elle inclut tout le contenu (*top bar*, fragments de navigation, et *bottom tabs*) ainsi que le *drawer* qui sera affiché.

3.10 Bottom tabs

En bas de l'écran, des onglets de navigation sont présents, permettant de naviguer entre les vues principales de l'application, ces *tabs* ont été implémentés à l'aide d'un menu classique Android défini au niveau XML, illustrés à la figure 10. En basant l'identifiant de chaque item du menu sur les identifiants référencés dans le *navigation graph*, chaque fragment est correctement appelé et affiché lors du clic sur un onglet.



FIGURE 10 – Onglets de l'application

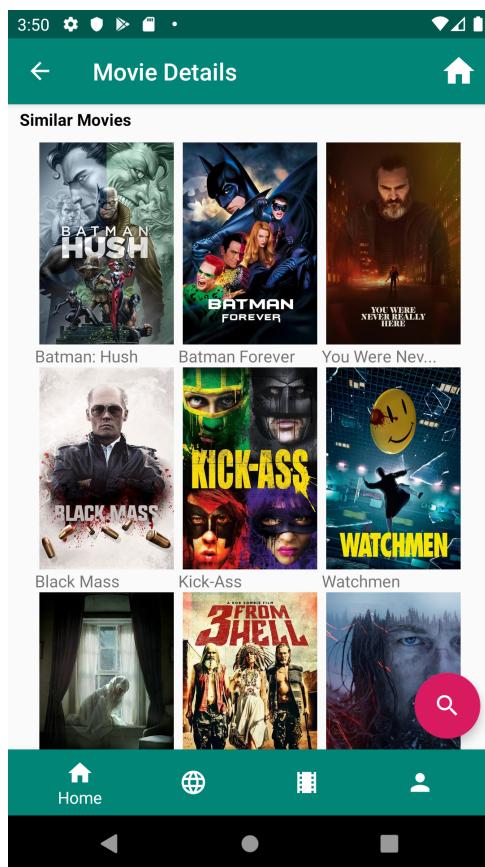


FIGURE 11 – Grille de films similaires dans les détails d'un film

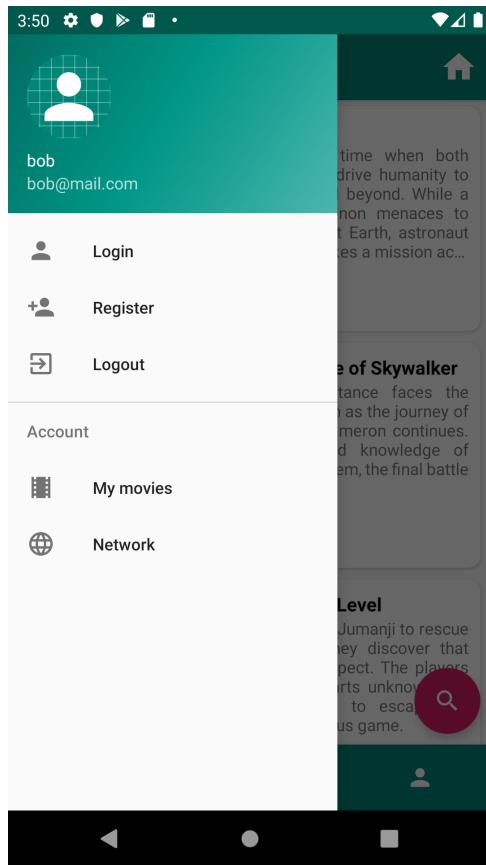


FIGURE 12 – Drawer de l'application

3.11 View pager

Pour les vues concernant les films appréciés/non appréciés et les utilisateurs suivis/abonnés, il était pratique de pouvoir passer d'une liste à l'autre rapidement et efficacement, comme illustré aux figures 13 et 14. Nous avons donc mis en place des *view pagers* qui sont en quelque sorte des sous-onglets permettant de "switcher" entre différents fragments assez rapidement. La réutilisation du code pour chaque sous-onglet (uniquement un changement de paramètre) est un avantage supplémentaire.

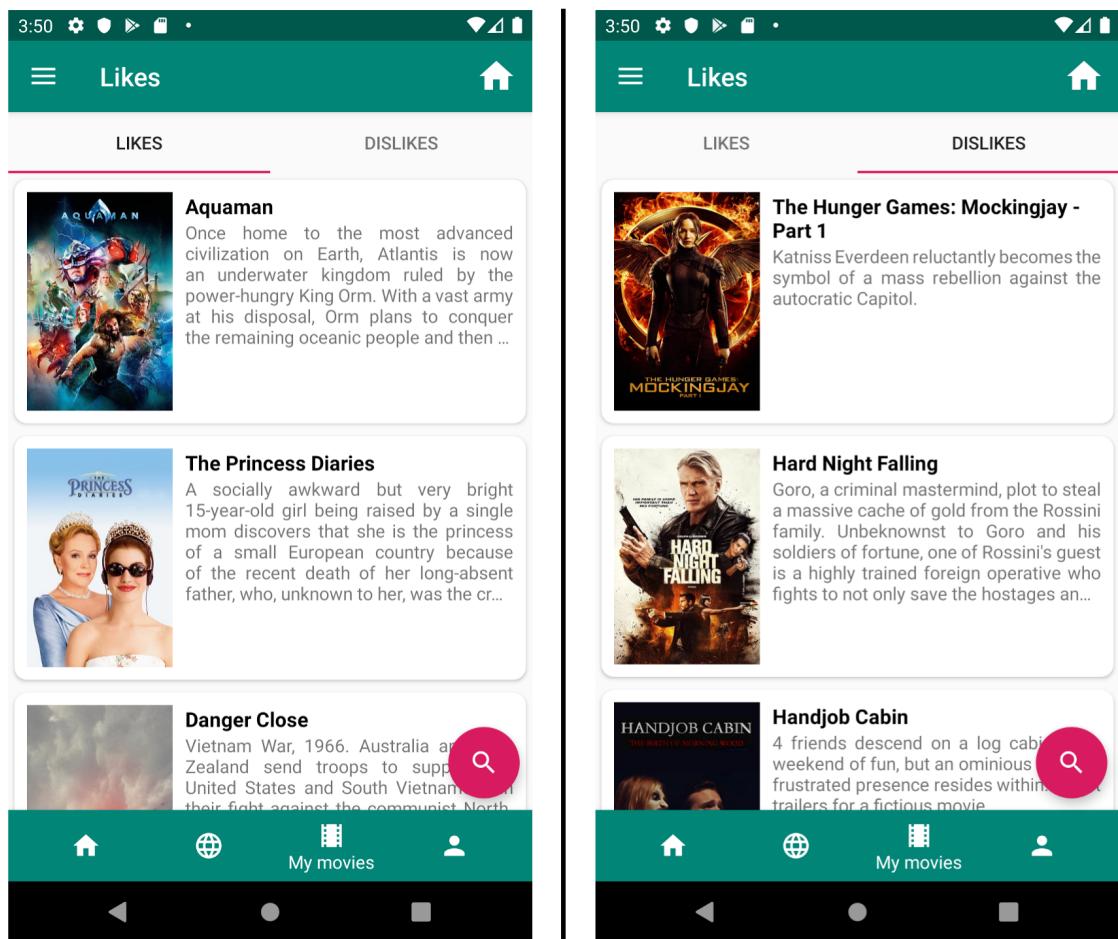


FIGURE 13 – Appréciations des films dans l'application

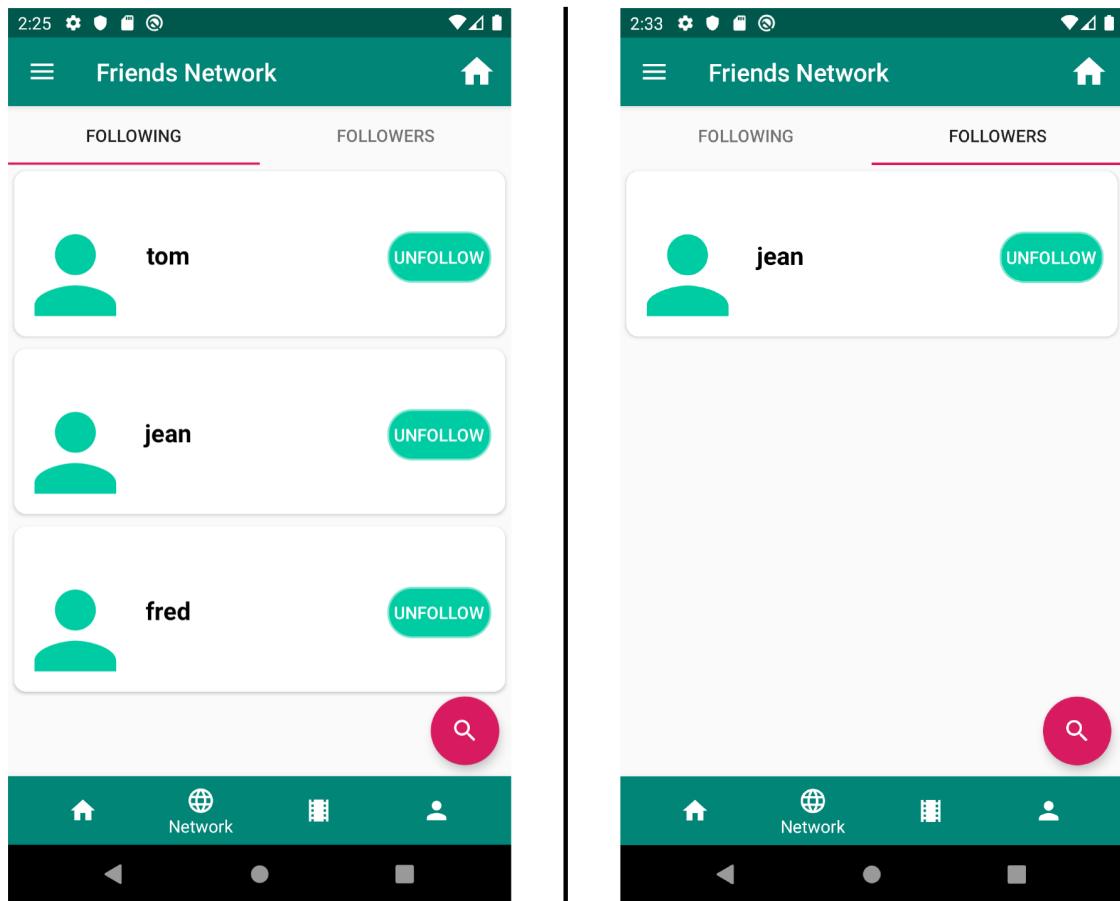


FIGURE 14 – Réseau social dans l'application

3.12 Recherche

La recherche est un fragment qui contient, au niveau XML, uniquement un champ texte dans lequel les mots-clé sont saisis et un bouton radio sélectionnant dans quelle rubrique rechercher. Lorsque l'action de recherche est effectuée, une *recycler view* présentant les résultats de la recherche est affichée. La *recycler view* appelée est toujours la même, nous avons uniquement le type des éléments affichés et le layout associé qui varient en fonction du type de la recherche (voir section 3.15). La figure 15 illustre ce comportement.

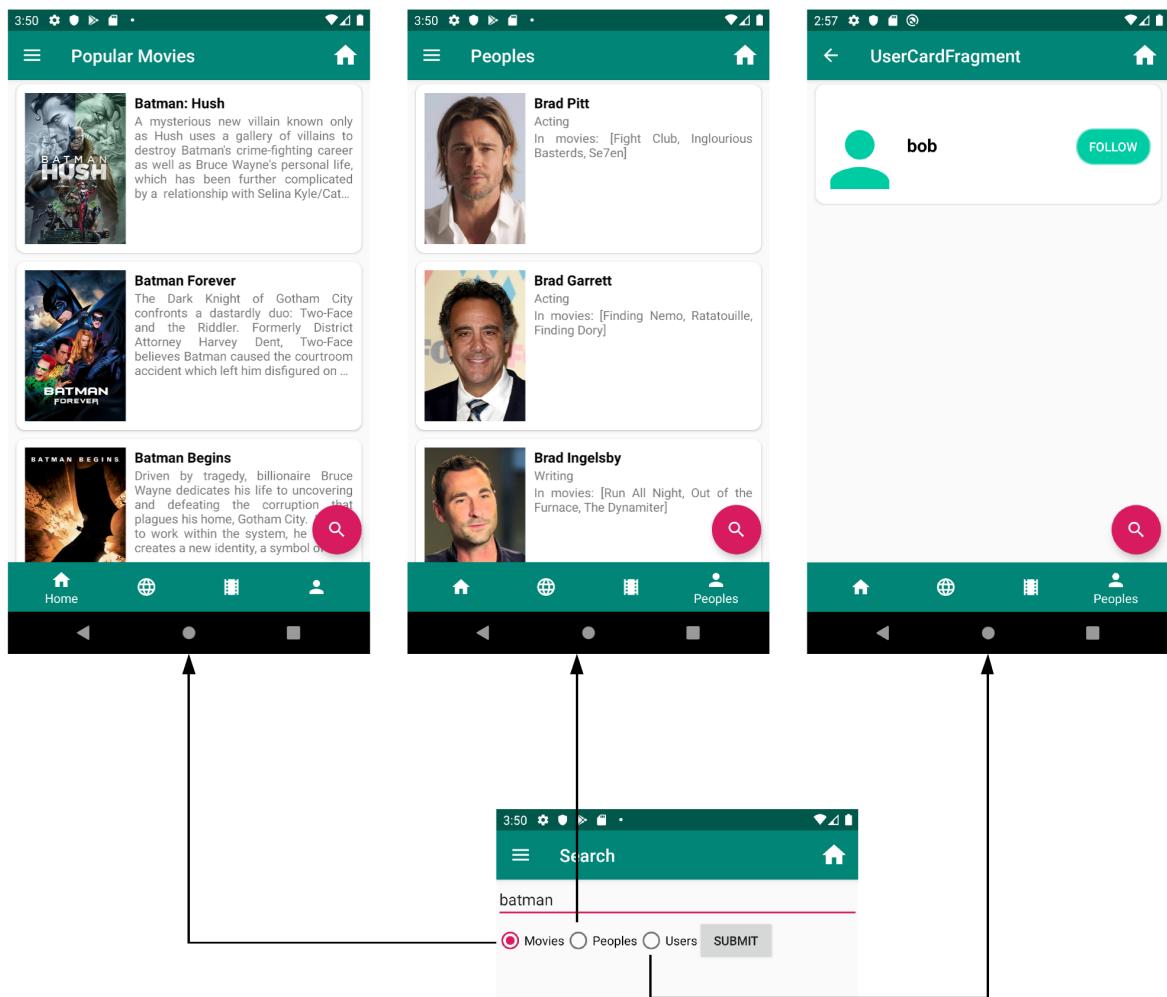


FIGURE 15 – Vue de la recherche dans l'application

3.13 Shared preferences

Afin de stocker la session d'un utilisateur connecté à l'application, nous avons fait usage des *shared preferences* qui sont un moyen simple (ensemble de clés-valeurs) et pratique de persister des informations dans l'application et d'y accéder depuis les différents fragments. Des exemples de leur utilisation sont disponibles aux listings 7, 8 et 9. Ce mécanisme permet également de vérifier qu'un utilisateur est authentifié et de le rediriger vers la page de login dans le cas échéant.

```

1  val sharedPref = activity?
2      .getSharedPreferences(getString(R.string.preference_file_key),
3          Context.MODE_PRIVATE) ?: return
4  with (sharedPref!!.edit()) {
5      putString(getString(R.string.pseudo), pseudo)

```

```

6     putString(getString(R.string.email), email)
7     putString(getString(R.string.token), token)
8     commit()
9 }
```

Listing 7 – Création des *shared preferences*

```

1 val auth = Common.getAuth((activity as MainActivity).
2     getSharedPreferences(getString(R.string.preference_file_key),
3         Context.MODE_PRIVATE), view.context)
4 if (auth != null) { ... }
```

Listing 8 – Vérification des *shared preferences*

```

1 with (pref!!.edit()) {
2     clear()
3     commit()
4 }
```

Listing 9 – Destructiction des *shared preferences*

3.14 FAB

Un *floating action button* (FAB) est disponible en bas à droite de l'écran, comme illustré dans la figure 16, un écouteur sur ce bouton est défini dans la *main activity* permettant d'intercepter les interactions des utilisateurs. Lorsqu'une interaction est détectée, une redirection vers le fragment dédié à la recherche est effectuée.

3.15 Generic adapter

Dans le cadre de cette application, nous travaillons très fréquemment avec la *recycler view* qui affiche une liste d'items, d'un même type pour toute la liste. Afin de ne pas devoir recréer une *recycler view* par type d'items, nous avons choisi d'implémenter un adaptateur générique permettant de réutiliser la même liste mais avec des items et *layouts* de type différents, comme illustré à la figure 17.

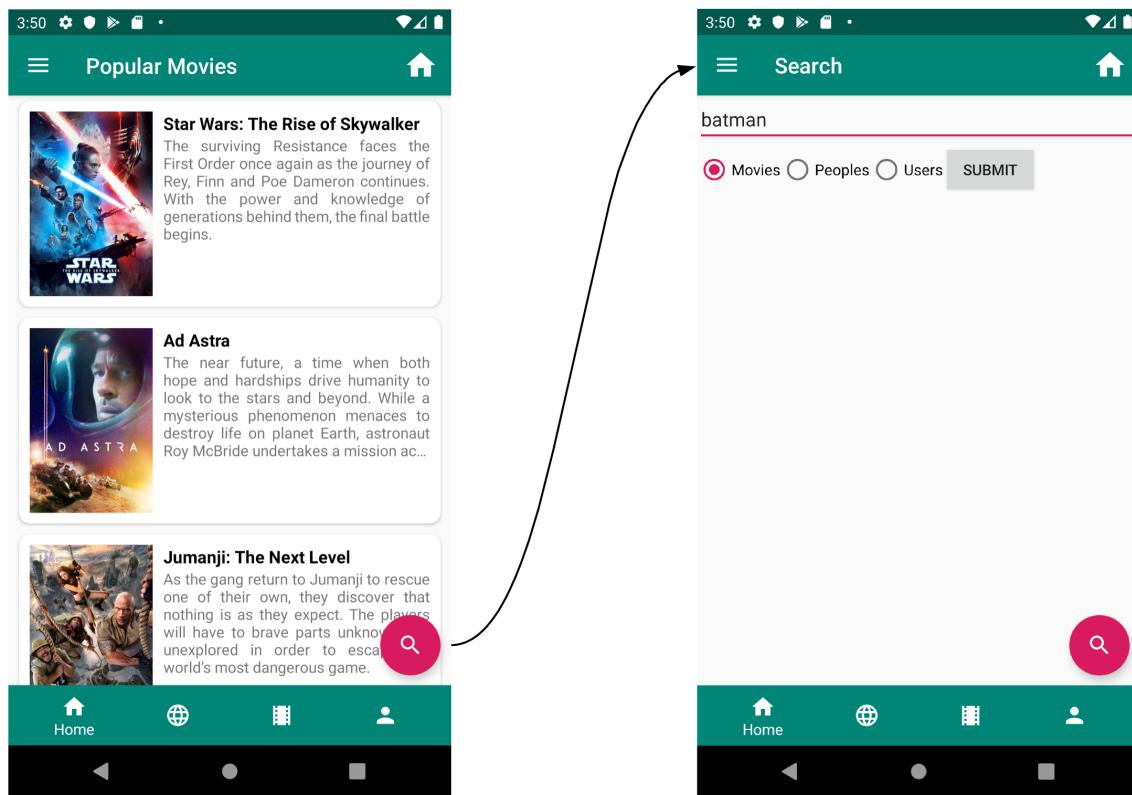


FIGURE 16 – FAB de l'application

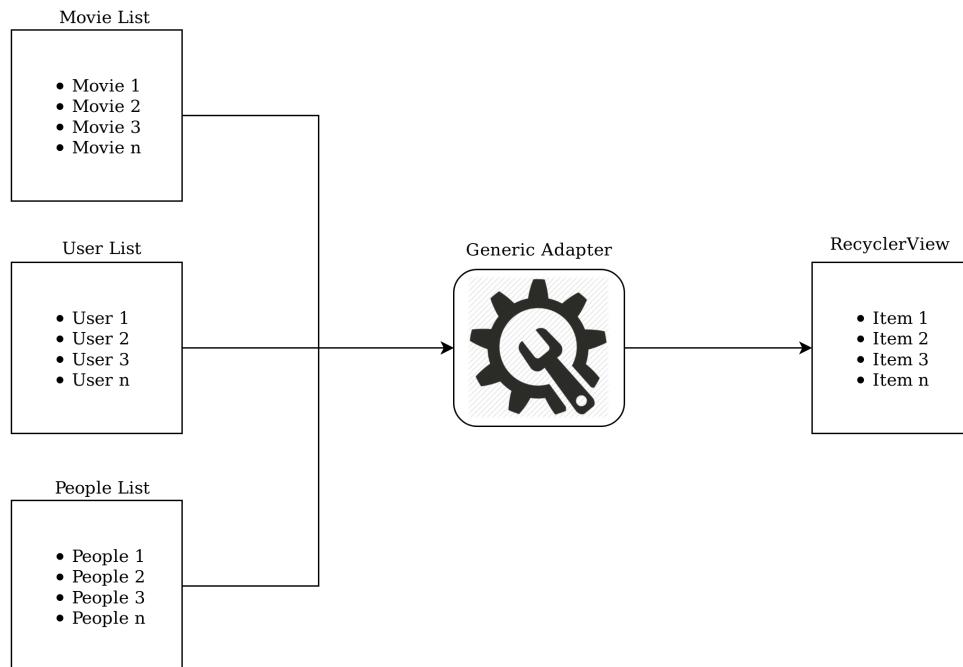


FIGURE 17 – Principes de l'adaptateur générique

4 Conclusion

4.1 Bilan personnel

Pour conclure, ce projet nous a permis d'apprendre et mettre en pratique de nombreux concepts propres à la programmation Android. Nous nous sommes également familiarisés avec le langage Kotlin, qui, une fois pris en main, simplifie et optimise grandement les opérations qui peuvent être plus complexes et fastidieuses en Java. Nous pensons avoir atteint les objectifs que nous nous étions fixés. Enfin, nous restons sur un sentiment très satisfaisant de cette première expérience dans le monde du développement Android.

4.2 Problèmes rencontrés

Les principales difficultés rencontrées durant le développement de ce projet sont les suivantes :

- Prise en main de la documentation Android officielle : par moments des raccourcis sont pris dans les explications, les morceaux de code fournis ne sont pas forcément entiers. Nous pensons que les rédacteurs partent du principe que le lecteur a fait une lecture linéaire et dans l'ordre, chose que nous n'avons pas fait.
- Compréhension totale des relations entre les layouts Android : des particularités de certains layouts demeurent un peu obscures.

4.3 Améliorations possibles

Les améliorations suivantes pourraient être apportées au projet :

- L'interface graphique qui est toujours optimisable.
- Interrogation de l'API Netflix afin de savoir si le film est disponible sur leur plateforme de streaming.
- Afficher les films à la une en fonction des préférences de l'utilisateur, par exemple déduites de ses films appréciés ou de ses recherches récentes.
- Afficher les informations relatives aux horaires des cinémas les plus proches pour les films actuellement dans les salles.
- Notifier les utilisateurs lorsqu'ils sont suivis par un autre utilisateur.
- Gestion de plusieurs langues.
- Reproduire les fonctionnalités des films pour les séries TV également.

5 Références

- [1] The Movie Database. The movie database (tmdb). <https://www.themoviedb.org/>. Consulté en 2019.
- [2] The Open Movie Database. Omdb api - the open movie database. <https://www.omdbapi.com/>. Consulté en 2019.
- [3] Akka HTTP. Akka http. <https://doc.akka.io/docs/akka-http/current/index.html>. Consulté en 2019.
- [4] YouTube. Youtube android player api | google developers. <https://developers.google.com/youtube/android/player>. Consulté en 2019.
- [5] Google. Documentation | android developers. <https://developer.android.com/docs>. Consulté en 2019.
- [6] Goole. Volley overview | android developers. <https://developer.android.com/training/volley/>. Consulté en 2019.