



Étapes du Dojo

Application de découverte de films:

Durant les prochaines heures, nous allons créer une application Android.

Cette application utilisera l'API mise à disposition par la base de données de films [The Movie Database \(TMDb\)](#) pour afficher sous forme de carte :

- Les films au cinéma actuellement
- Le top des films de l'année
- Les films à venir

En bonus (🎉), il sera aussi possible d'ajouter ces films en favoris afin de garder une sélection de film dans l'application.

Architecture du code

Model View Presenter (MVP) + Repository

- **View:**

La **View** correspond à l'interface que l'utilisateur aura sous les yeux. Elle est en charge d'informer le **Presenter** des actions de l'utilisateur et d'afficher les données récupérées. Elle est matérialisée ici par une interface `<name>ViewContract` qui permet d'abstraire les actions possibles sur notre vue.

- **Presenter:**

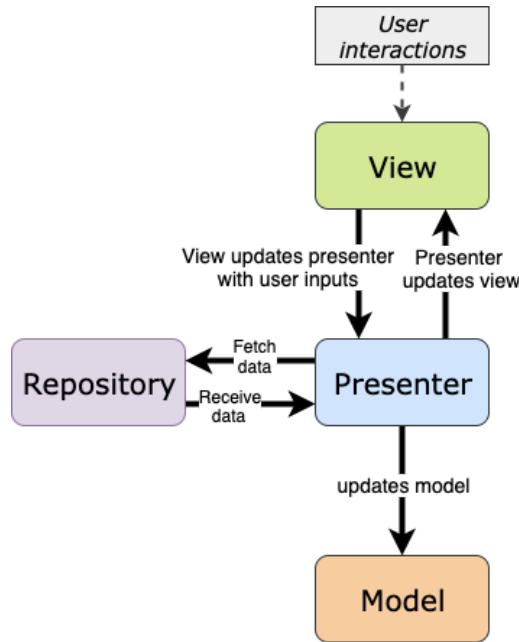
Le **Presenter** reçoit les actions de l'utilisateur et sera en charge de récupérer les données pertinentes du **Model** dans le **Repository**. Il permet de faire le lien entre la **View** et le **Model**.

- **Model:**

Le **Model** regroupe l'ensemble des ressources métiers de notre application qui seront affichées.

- **Repository:**

Le **Repository** va servir à récupérer les données de l'application de diverses manières, dans une base de données, en cache ou encore sur des API.



Executors (threading)

Sur Android, il est impossible de faire des appels réseaux sur le **main thread** (thread en charge de la vue, il est en charge d'afficher l'application avec un certain nombre de frame par seconde).

Il faut donc passer les appels réseau sur un autre thread puis envoyé la réponse sur le **main thread**.

Pour faciliter la gestion des threads, en s'inspirant des recommandations de Google en terme d'architecture d'application, la classe `AppExecutors` a été créée. Elle permet d'avoir accès à 3 types d'executors qui permet de lancer des actions sur 3 types de threads: le **main thread**, le thread **network IO** et le thread **disk IO**.

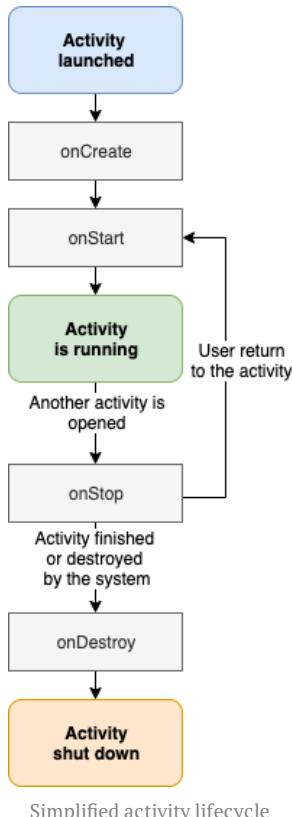
Activité:

An `Activity` is a single, focused thing that the user can do.
<https://developer.android.com/reference/android/app/Activity>

Une application Android, est composée d'une ou plusieurs activités (et de fragments).

Ces objets suivent un cycle de vie dès lors que l'application est lancée.

- `onCreate` :
C'est ici que nous allons créer la vue de notre activité et initialiser les vues de cette dernière.
- `onStart` :
Démarrage de l'activité, c'est ici qu'on va appeler le presenter pour aller chercher les données.
- `onStop` : (symétrique de `onCreate`)
Il faut informer le presenter que l'activité est arrêtée, et donc qu'il ne faut pas essayer de la mettre à jour si des données cherchées sont disponibles.
- `onDestroy` : (symétrique de `onCreate`)
L'activité est détruite.



Simplified activity lifecycle

<https://developer.android.com/guide/components/activities/activity-lifecycle>

1. Création de la carte d'un film



Pour commencer: `git stash -a && git checkout step1.0`



Pour la liste complète des raccourcis : <https://developer.android.com/studio/intro/keyboard-shortcuts>
 Vous trouverez le raccourci permettant de corriger une erreur ou un warning en étant guidé par Android Studio sous le nom *Project quick fix (show intention actions and quick fixes)*. C'est **Alt+Enter** sur Windows / Linux et **Option+Enter** sur Mac.

Inflate un layout dans une activité

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of View and ViewGroup objects. A View usually draws something the user can see and interact with.

(<https://developer.android.com/guide/topics/ui/declaring-layout>)

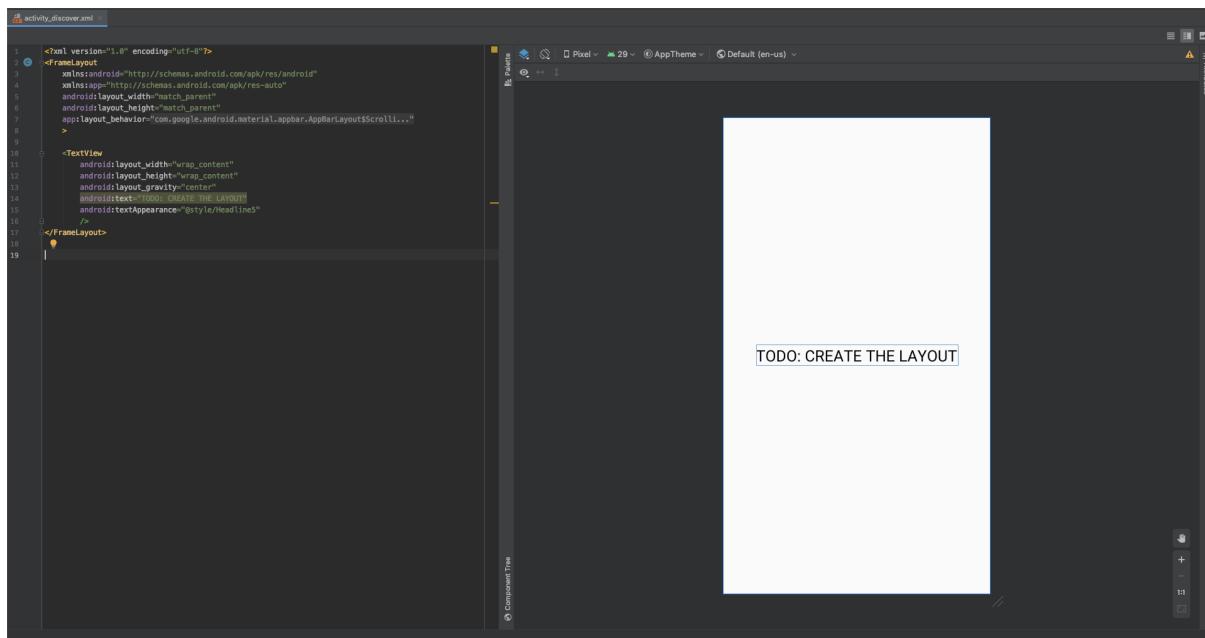
La façon la plus utilisée pour créer l'interface utilisateur de notre activité est d'utiliser un layout.

C'est un fichier XML qui définit la structure de notre interface utilisateur.



Ouvrir le fichier `app/res/layout/activity_discover.xml`

Vous devriez voir l'écran ci-dessous:



Comme vous pouvez le voir, un layout ressemble plus ou moins à un fichier HTML. On a ici un `ViewGroup`, le `FrameLayout` qui contient tout simplement des vues sans arrangement spécial, et une `View`, la `TextView` qui permet d'afficher du texte.

Pour utiliser ce layout dans notre activité, il faut utiliser la méthode `setContentView` lors de la création de l'activité.



Ouvrir la classe `com.fabernovel.codingdojo.app.main.DiscoverActivity` et utiliser `setContentView` pour inflate le layout `activity_discover`.



Pour accéder aux ressources de l'application depuis une classe, il faut utiliser `R.<type de la ressource>.nom de la ressource`.

Ici ce sera donc `R.layout.activity_discover`



Pour chercher une classe utiliser le raccourci `CTRL+N` / `CMD + O`

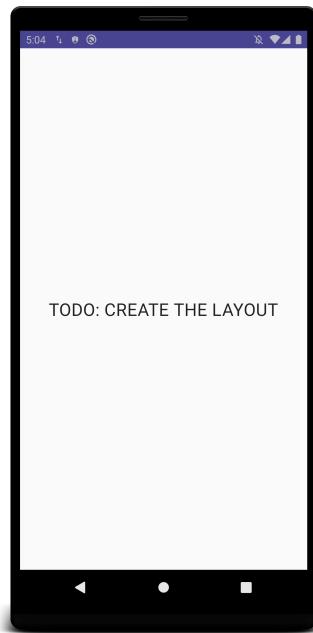
Pour chercher une fichier, utiliser `CTRL+SHIFT+N` / `CMD+SHIFT+N`



Relancer l'application



▼ Vous devriez avoir l'écran ci-dessus:

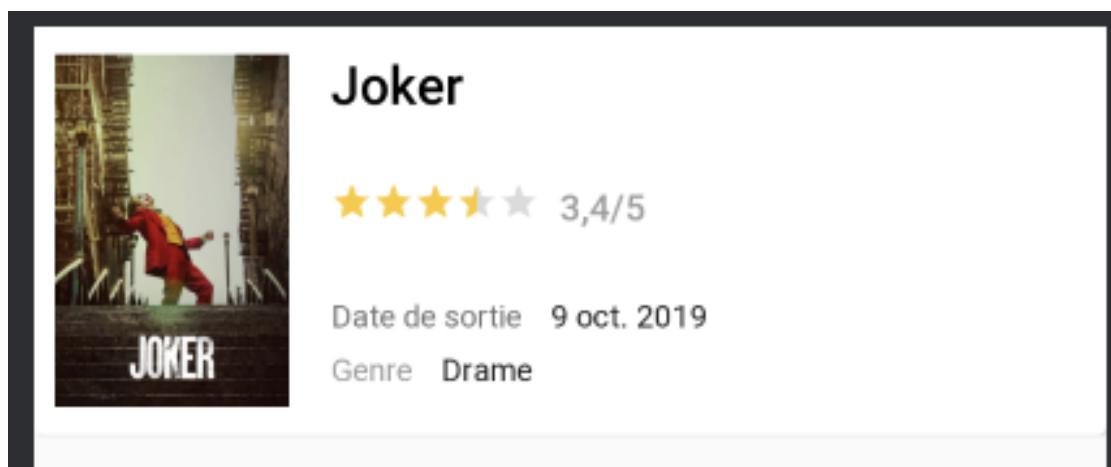


Création de la carte d'un film



Si tu as pris du retard: `git stash -a && git checkout step1.2`

Le but maintenant va être d'afficher la carte ci-dessous:



Layout d'une carte de film



Blueprint d'une carte de film



Les vues d'un layout ont deux attributs obligatoires: `layout_width` et `layout_height`.

La valeur de ces attributs peut-être:

- Un dimension (en `dp` de préférences exemple: `layout_width="56dp"`)
- `wrap_content` : Android calcule la taille de vue minimale possible.
- `match_parent` : la vue prend tout l'espace dans le parents.

Pour faire la carte en elle-même, nous allons utiliser une `CardView`



`CardView` vient de la librairie de material design de google qui a déjà été ajoutée dans le projet.

(<https://material.io/develop/android/docs/getting-started/>)

Afin d'organiser les éléments au sein de cette `CardView`, nous allons utiliser des `LinearLayout`. Grâce à ce layout, il est possible d'aligner des vues de façon horizontales ou bien vertical.

```
<!-- VERTICAL -->
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    >
<!-- VIEWS -->
</LinearLayout>

<!-- HORIZONTAL -->
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    >
<!-- VIEWS -->
</LinearLayout>
```

Pour les textes, nous allons utiliser des `TextView`. Le titre du film utilisera `android:textAppearance="@style/Headline6"`, les label (date de sortie et genre) `android:textAppearance="@style/Label"` et les valeurs de ces labels: `android:textAppearance="@style/Caption"`.

Le texte affiché par les labels se trouve dans le fichier de resource `res/values/string.xml`. Pour les utiliser dans le layout, ajouter un attribut: `android:text="@string/<nom de la ressource>"`.

Pour les étoiles de la note du film, une `RatingBar`, sur laquelle on utilisera l'attribut `isIndicator="true"` pour empêcher les utilisateurs de modifier le nombre d'étoiles.

Vues à ajouter dans le layout

Aa Vue	≡ ID	≡ Type de vue	≡ Commentaire	≡ Variable
<u>Poster</u>	movie_card_poster	ImageView		moviePoster
<u>Titre</u>	movie_card_title	TextView		movieTitle
<u>Étoiles</u>	movie_card_stars	RatingBar	- Ajouter un style: <code>style="@style/Widget.AppCompat.RatingBar.Small";</code> - Ajouter <code>android:isIndicator="true"</code>	movieRatingBar
<u>Note</u>	movie_card_rating	TextView		movieRating
<u>Label de date de sortie</u>	movie_card_release_date_title	TextView		
<u>Date de sortie</u>	movie_card_release_date_value	TextView		movieReleaseDateValue
<u>Label du genre</u>	movie_card_genre_title	TextView		
<u>Genre</u>	movie_card_genre_value	TextView		movieGenreValue
<u>Message de loading</u>	discover_message	TextView		discoverMessage
<u>Carte</u>	movie_card	CardView		movieCard



Ouvrir le layout `item_movie` dans le dossier `app/res/layout`.

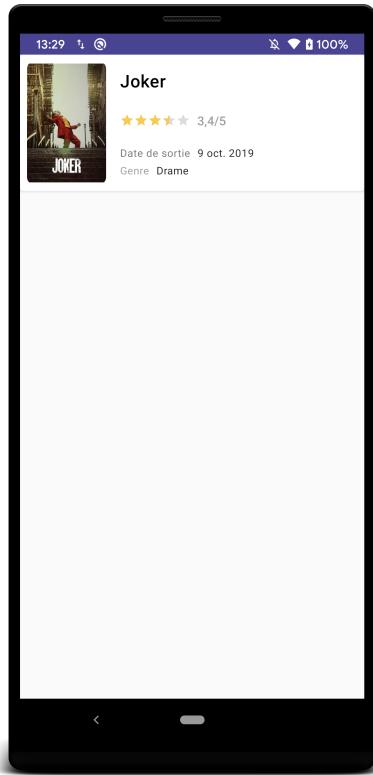
La vue sera composée d'une `CardView` parente dans laquelle se trouvera un premier `LinearLayout` horizontal qui contiendra le poster et un second `LinearLayout` vertical qui contiendra les autres vues. Pour le poster, il est possible d'utiliser l'image `joker.png` mise à disposition.

Dans le layout `activity_discover`, ajouter `<include layout="@layout/item_movie" />` au dessus de la vue `discoverMessage` et ajouter `android:visibility="gone"` sur la vue `discoverMessage`.



Lancer l'application.

▼ Résultat



2. Afficher les données des films dans la carte créée



Si tu as pris du retard: `git stash -a && git checkout step2.0`

Pour l'instant, les données affichées sont en dur dans le layout.

Le but de cette partie sera de récupérer les vues de la carte de film dans l'activité et d'y mettre les valeurs venant d'un objet `Movie`.

Récupérer les vues d'un layout



Pour récupérer une vue venant d'un layout dans une activité, il faut utiliser la méthode `findViewById` qui permet de rechercher une vue avec un certain id dans toute la hiérarchie de la vue utilisée par notre activité avec `setContentView`.
[https://developer.android.com/reference/android/app/Activity#findViewById\(int\)](https://developer.android.com/reference/android/app/Activity#findViewById(int))
mentionner R.id. ?

dGV



Dans la classe `DiscoverActivity`, créer des variables permettant de récupérer les différentes vues de notre carte film (poster, note, étoiles, titre, genre, etc).
Initialiser ces variables après l'appel à `setContentView`.

Pour rappel sur les types de vue:

Vues à ajouter dans le layout

Aa Vue	≡ ID	≡ Type de vue	≡ Variable	≡ Commentaire
<u>Genre</u>	movie_card_genre_value	TextView	movieGenreValue	
<u>Label du genre</u>	movie_card_genre_title	TextView		
<u>Date de sortie</u>	movie_card_release_date_value	TextView	movieReleaseDateValue	
<u>Label de date de sortie</u>	movie_card_release_date_title	TextView		
<u>Note</u>	movie_card_rating	TextView	movieRating	
<u>Étoiles</u>	movie_card_stars	RatingBar	movieRatingBar	- Ajouter un style: style="@style/Widget.AppCompat.RatingBar.Small"; - Ajouter android:isIndicator="true"
<u>Titre</u>	movie_card_title	TextView	movieTitle	
<u>Poster</u>	movie_card_poster	ImageView	moviePoster	
<u>Message de loading</u>	discover_message	TextView	discoverMessage	
<u>Carte</u>	movie_card	CardView	movieCard	

Afficher du contenus dans des vues



Si tu as pris du retard: `git stash -a && git checkout step2.2`

Maintenant que les vues ont bien été récupérées, il reste à récupérer un object `Movie` à afficher. Pour cela utiliser la classe `DiscoverPresenter`.



Tout d'abord, il faut implémenter l'interface `DiscoverViewContract` et override ses méthodes dans `DiscoverActivity`:

1. `showLoading` : afficher le message `R.string.loading` dans la vue `R.id.discoverMessage`, puis changer sa visibilité (`setVisibility`) à `View.VISIBLE` et celle de la carte film à `View.GONE`.
2. `showContent` : afficher les différentes propriétés du paramètre `Movie` dans les vues correspondantes puis changer sa visibilité à `View.VISIBLE` et celle du message à `View.GONE`
3. `showError` : afficher le message en paramètre dans la vue `R.id.discoverMessage`, puis changer sa visibilité (`setVisibility`) à `View.VISIBLE` et celle de la carte film à `View.GONE`.



- Pour afficher l'image dans le poster, utiliser `Picasso.get().load(model.getPosterPath()).into(moviePoster);`. Picasso permet de charger des images dans des `ImageView`.
- Pour mettre du texte dans une `TextView`, utiliser la méthode `setText(<text>)`.
- Pour mettre en forme la date de sortie, utiliser `DateUtils.formatDate(<date>)` qui formate la date en chaîne de caractères
- Pour mettre le nombre d'étoile de la `RatingBar`, utiliser le méthode `setRating(<note>)`

Il faut maintenant récupérer et initialiser le `DiscoverPresenter`.

-  1. Dans la `DiscoverActivity`, créer une variable de type `DiscoverPresenter` initialisée dans le `onCreate` en utilisant la méthode statique `getInstance(this)`. (cela permet de ne pas exposer le repository à la vue)
 2. Dans la méthode `onStart` appeler la méthode `start` du presenter en lui passant le view contract (**ici this puisque notre activité implémente ce dernier**).
 3. Dans la méthode `onStop`, juste avant l'appel à `super.onStop()`, appeler la méthode `stop` du présentateur. Cela permet, à la vue de ne pas être mise à jour lorsqu'elle n'est plus visible.

Récupérer un film dans le repository



Si tu as pris du retard: `git stash -a && git checkout step2.3`

Le `MovieRepository` fourni permet de récupérer un film. Pour l'instant, le film est déclaré en dur dans le repository.



Pour récupérer un film à l'aide du `MovieRepository`, l'utilisation d'un callback est nécessaire. En effet, la récupération d'un film est asynchrone et il faut attendre d'obtenir le résultat pour l'afficher.
 Le callback est ici déclaré dans l'interface `GetMovieCallback` qui comporte deux méthodes `onGetMovie` lorsque le film a été récupéré et `onError` en cas d'erreur.
 Le repository va récupérer les films en utilisant le thread Network I/O puis appeler le callback sur le main thread.



Dans la méthode `start` du presenter:

1. Afficher l'écran de chargement du view contract
2. Dans l'attribut `movieCallback` de type `GetMovieCallback` et appeler respectivement `showContent` et `showError` en cas de succès (`onGetMovie`) et d'erreur (`onError`).
3. Récupérer le film avec le repository: `movieRepository.getMovie(movieCallback)`



Lancer l'application.

3. Afficher une liste de films



Si tu as pris du retard: `git stash -a && git checkout step3.0`

Pour l'instant, le layout qu'on a mis en place ne supporte que l'affichage d'une seule carte de film.



Pour pouvoir afficher une liste de films de n'importe quelle taille, on va utiliser l'incontournable `RecyclerView`
<https://developer.android.com/guide/topics/ui/layout/recyclerview>



1. Dans le fichier `app/res/layout/activity_discover.xml`, remplacer le bloc qui permettait d'afficher la carte de film (`<include layout="@layout/item_movie">`) par une `RecyclerView` avec l'id `movie_recycler`.
2. Faites les changements nécessaires dans la classe `DiscoverActivity` pour récupérer la `RecyclerView` dans l'activité, et ensuite gérer sa visibilité en fonction de l'état de la vue : loading, content, etc.
 Si besoin, se référer à la section [Add RecyclerView to your layout](#).



Commenter le code permettant de mettre les données dans la carte de film afin de pouvoir le réutiliser après dans le `ViewHolder`

Passer la liste de films à la RecyclerView



Si tu as pris du retard: `git stash -a && git checkout step3.1`

Pour pouvoir fonctionner, une RecyclerView s'utilise toujours avec un `Adapter` qui gère la liste des éléments à afficher. L'`Adapter` a à son tour besoin d'un `ViewHolder` qui lui gère l'affichage d'un élément de la liste. Par souci d'efficacité, ces 2 classes vous ont été fournies et sont prêtes à être utilisées. Quand on implémente une recycler view pour la première fois, on peut avoir un peu de mal à comprendre comment tout cela fonctionne, voici donc quelques explications :

Le ViewHolder

Un view holder a simplement pour but de gérer l'affichage d'un élément de la liste. Si vous jetez un coup d'oeil sur la classe `MovieViewHolder`, vous verrez que son constructeur prend en paramètre une `View`. Cette vue correspond à une carte de film, et en appelant la méthode `findViewById` du paramètre `itemView` avec les bons ids, on récupère séparément des références sur les vues contenues dans la carte : la `TextView` qui affiche le titre, l'`ImageView` qui contient le poster, etc. Ces références sont utilisées plus tard dans la méthode `bind()` pour alimenter chaque carte de film avec les données d'un élément de la liste. C'est là qu'entre en jeu l'`Adapter`.

L'Adapter

Un adapter a pour rôle de faire le lien entre une liste d'objets de type `Movie` et les ViewHolders qui vont afficher chacun des éléments de la liste. Si vous jetez un coup d'oeil sur la classe `MoviesAdapter`, vous verrez qu'il contient les éléments suivants :

- un attribut de type `List<Movie>` `dataSet` qui permet de stocker la liste de films.
- une méthode `onCreateViewHolder(@NonNull ViewGroup parent, int viewType)` qui sert, comme son nom l'indique, à créer un ViewHolder, et donc à inflate le bon layout pour un élément de la liste en particulier. La vue à afficher dans le `MovieViewHolder` est celle contenue dans le layout `item_movie`.
- une méthode `onBindViewHolder(@NonNull MovieViewHolder holder, int position)` qui permet de passer un objet de type `Movie` à un `MovieViewHolder`.



Un recycler view peut supporter une liste composée d'éléments construits via des layouts différents (par exemple, si on veut afficher les films par catégorie on peut ajouter des cartes "catégorie" au début de chaque section), d'où le paramètre `viewType` qui nous aurait permis de choisir le layout et la classe du ViewHolder en fonction de l'élément à afficher.

Récupérer une liste de films dans le repository

Pour tester notre recycler view, utiliser la méthode `getMovies()` de `MovieRepository` (qui retourne une liste de 3 films) à la place de `getMovie()` dans le presenter.



Faites les changements nécessaires sur la méthode `showContent()` (dans `DiscoverViewContract` et `DiscoverActivity`).

Dans `DiscoverPresenter` remplacez le `GetMovieCallback` par `GetMoviesCallback` pour récupérer une liste de film.

Passer la liste à l'Adapter

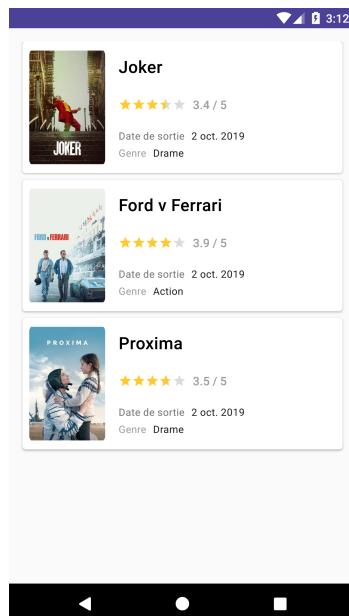


1. Dans la classe `DiscoverActivity`, déclarer un attribut de type `MoviesAdapter` et le relier à la recycler view en utilisant la méthode `setAdapter(Adapter adapter)` de la classe `RecyclerView`.
2. Dans la méthode `showContent` de `DiscoverActivity`, transmettre la liste reçue en paramètre à l'Adapter via `submitList()`.



Lancer l'application.

▼ Résultat



4. Utiliser une API pour chercher les films à afficher

Comme vous vous en doutez, écrire nous même la liste des films à afficher n'a pas trop d'intérêt. Pour que notre application devienne vraiment utile, on va se servir de la base de données (non-officielle) disponible ici [The Movie Database \(TMDb\)](#). Plus précisément, on va utiliser l'API REST qui expose différents endpoints qui permettent d'interroger la base de données de différentes manières. La documentation exhaustive de cette API est disponible ici <https://developers.themoviedb.org/3/getting-started>

Découvrir l'API

Avant de l'intégrer dans le code de notre application, il est pertinent de prendre quelques minutes pour interagir avec L'API et comprendre son fonctionnement.

Pour ce dojo, le endpoint `/discover/movie` suffit à notre besoin. Ce endpoint renvoie une liste de films qui correspond à la combinaison de filtres (paramètres) spécifiée dans la requête qu'on lui envoie.

L'accès à l'API se fait à l'aide une clé d'API qu'il faut insérer dans chaque requête à l'aide du paramètre `api_key`.



Pour plus d'informations sur les moyens d'authentification à l'API, consulter la page <https://developers.themoviedb.org/3/getting-started/authentication>



1. Consulter cette page de la doc <https://www.themoviedb.org/documentation/api/discover> qui illustre certains paramètres de requête qu'on peut utiliser avec le endpoint `/discover/movie` pour restreindre la liste de films qu'on récupère
2. Cette page <https://developers.themoviedb.org/3/discover/movie-discover> (onglet : **Try it out**) permet d'interagir avec le endpoint simplement, en insérant des paramètres dans un formulaire. Envoyer une ou deux requêtes et vérifier que la réponse (en format **json**) est cohérente (ne pas oublier de renseigner la clé d'api)
3. Quelles sont les 3 combinaisons de paramètres qui nous permettront de récupérer les 3 listes de films à afficher dans les 3 onglets de l'app ?

Compléter la définition de l'API

Pour faciliter l'envoi de requêtes et la réception des réponses, la librairie **Retrofit** est très pratique. Vous pouvez jeter un coup d'oeil sur la la page web de [Retrofit](#) pour voir comment ça marche (principalement les sections *Introduction*, *REQUEST METHOD*, et *URL MANIPULATION*)

Retrofit est un client REST qui facilite l'échange avec une API REST. Pour pouvoir s'en servir, il faut déclarer une interface qui décrit les requêtes qu'on va faire auprès de l'API. Chaque requête se traduit par un prototype de méthode, annotée du type de requête (`@GET`, `@POST`, etc). L'annotation prend en paramètre le "path" de la requête **sans la base url**. On vous a fourni l'interface ([MovieApi](#)) avec la méthode suivante que nous utiliserons pour chercher les films qui vont sortir bientôt :

```
@GET("/3/discover/movie")
Call<RestDiscoverResponse> discoverMovies(
    @Query("api_key") String apiKey,
    @Nullable @Query("primary_release_date.gte") String fromDate,
    @Nullable @Query("primary_release_date.gte") String toDate,
    @Nullable @Query("primary_release_year") Integer year,
    @Query("language") String language
);
```



- * la fameuse **base url** qu'on a évoquée ci-haut est définie dans la classe [Network.java](#). Comme son nom l'indique, c'est le commencement du path de toutes les requêtes de l'API.
- * le path de la requête `"/3/discover/movie"` commence par `/3` pour choisir la version n°3 de l'API qui permet de s'authentifier via une clé d'API
- * Il y a différentes façons de communiquer ses paramètres de requête à l'API. Ici on envoie la clé d'API, la date minimum de sortie, et la langue du film sous forme de `@Query` params comme indiqué par la doc du endpoint.
- * Comme type de retour nous avons indiqué `Call<RestDiscoverResponse>`. RestDiscoverResponse est le **modèle** qu'on reçoit du serveur. On a besoin de définir les modèles qu'on échange avec l'API parce que structurellement parlant, ils ne correspondent pas forcément aux entités qu'on manipule dans le code, d'où la définition d'un modèle [RestMovie](#) par exemple. Dans cette classe, nous avons fait appel à l'annotation `@SerializedName(...)` pour indiquer l'appellation du champ dans le Json renvoyé par l'API, et qui peut différer de la façon dont on a appelé notre champ dans le code.

Compléter le repository



Si tu as pris du retard: `git stash -a && git checkout step4.1`

Le [MovieRepository](#) qu'on a actuellement nous renvoie des "mocks", c'est à dire des données stockées actuellement. Dans cette partie, on va remplacer ces mocks par des films récupérés via l'API. Pour ce faire, le repository va

maintenant appeler les méthodes qu'on vient de définir dans l'API. Comme vous pouvez le constater, ces méthodes retournent une variable de type `Call<type du résultat>`. C'est seulement en appelant la méthode `execute()` de cette variable de type `Call` que la requête est réellement envoyée et si aucune exception I/O n'est levée, on récupère un objet de type `Response<type du résultat>`. Cet objet a une méthode `isSuccessful()` qui nous permet de savoir si le serveur a réussi à bien interpréter notre requête et nous a renvoyé une réponse 200. Dans ce cas on peut récupérer le contenu de la réponse via la méthode `body()` de l'objet `Response` qu'on a reçu.



1. Dans la classe `MovieRepository`, utiliser la méthode `discoverMovies` de `MovieApi` avec les bons paramètres dans les méthodes `getUpcomingMovies`, `getTopMoviesOfTheYear` et `getMoviesInTheatre`.
2. Décommenter le `dispatchSuccess` dans la méthode `fetchMovieList`
3. Adapter le code de `DiscoverPresenter` pour qu'au lancement il demande au repository de chercher les films qui sont dans les salles de cinéma en ce moment.



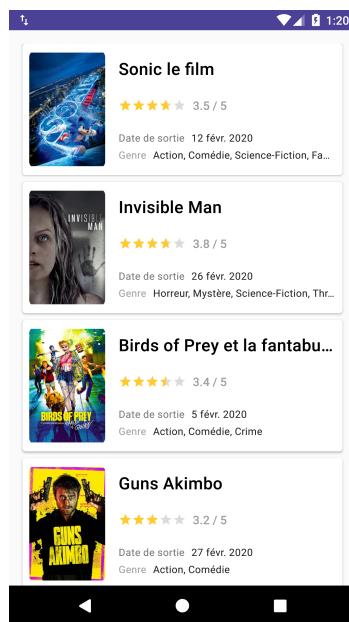
Ici la clef d'API est écrite en claire dans le code de l'application. Ce n'est pas une pratique recommander car il est facile pour un acteur malveillant de décompiler l'application et d'utiliser cette clef.



Lancer l'application.

▼ Résultat

Si tout s'est bien passé, vous aurez une longue liste de films renvoyée par le serveur ! S'il y a un problème quelconque, c'est probablement dû à une erreur serveur ou à une requête mal formatée. Pour vous faciliter l'identification de la cause des éventuels problèmes, l'intercepteur graphique de requêtes HTTP **Chuck** a été attaché à Retrofit (d'où les 2 petites flèches en haut à gauche de l'écran). Cet outil vous permettra de voir ce que vous envoyez à l'API et ce qu'elle vous renvoie.

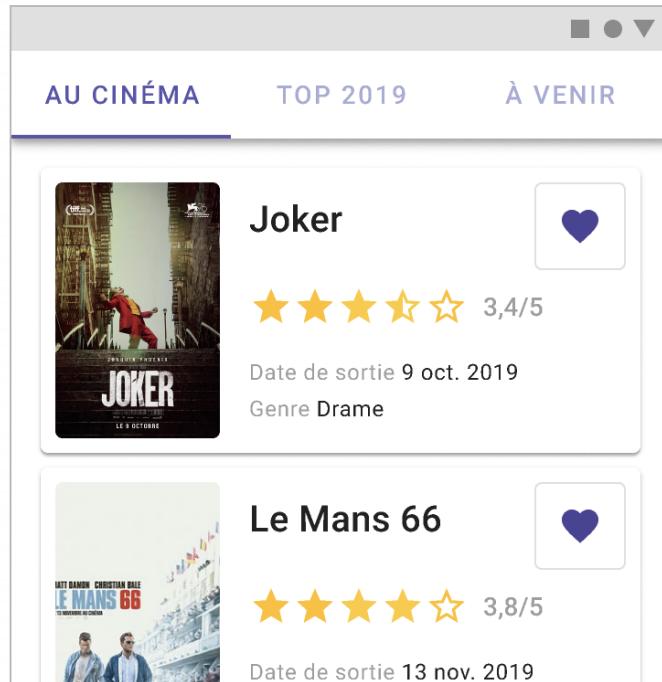


Compléter le layout avec les 3 onglets



Si tu as pris du retard: `git stash -a && git checkout step4.2`

Le but de cette partie est de rajouter ces 3 onglets pour pouvoir naviguer entre les différentes listes :



Pour cela, on va utiliser un **TabLayout**.



1. Dans `activity_discover.xml`, rajouter un `TabLayout` en ajustant bien les contraintes pour qu'il apparaisse en haut de l'écran.
(on pourra par exemple mettre les vues actuellement dans un frame layout puis mettre ce frame layout sous les tabs à l'aide d'un linear layout)
2. Rajouter 3 `TabItem` dans le `TabLayout` de manière à afficher 3 onglets. Pour attribuer à chaque onglet son titre, utiliser l'attribut `android:text`
3. Le titre de l'onglet du milieu est dynamique puisqu'il contient l'année en cours. Il faut donc lui donner une valeur "programmatiquement" (dans le code Java). Pour cela récupérer la vue correspondant au `TabLayout` dans `DiscoverActivity`, et récupérer ensuite l'onglet du milieu via la méthode `getTabAt` de la classe `TabLayout`. Enfin, utiliser une ressource de type `string` avec un argument. Pour ce faire, jeter un coup d'oeil à la section Formatting strings.



Dans un `LinearLayout`, `layout_weight` permet de donner un poids aux vues afin qu'elle prenne le plus de poids possible.

Gestion des clicks sur les onglets



Si tu as pris du retard: `git stash -a && git checkout step4.3`



1. Dans `DiscoverActivity`, utiliser la méthode `addOnTabSelectedListener` de la classe `TabLayout` pour définir un *listener* sur vos onglets. Ce listener est une interface que l'on doit implémenter. Laissez-vous guider par les indications et les raccourcis d'Android Studio pour générer le code minimum de l'implémentation de cette interface.
2. La méthode qui nous intéresse le plus est `onTabSelected`, on peut laisser les 2 autres méthodes vides. `onTabSelected` prend en paramètre l'onglet qui vient d'être cliqué. Récupérer l'indice de cet onglet via `getPosition()` et en fonction de cet indice, appeler la bonne méthode du presenter pour déclencher la récupération de la bonne liste de films.

▼ Bonus (TODO)

5. Ajout d'un film en favori



Si tu as pris du retard: `git stash -a && git checkout step5.0`

TODO

6. Affichage de la liste des favoris

TODO

