Documentation Technique
du projet : android-technical-test

par Kévin LENEYLE

Choix des librairies :

- Room : La librairie Room permet de créer une base de donnée hors ligne, que ce soit pour conserver les informations des albums en cas de coupure internet ou pour rajouter une gestion de favoris en local.

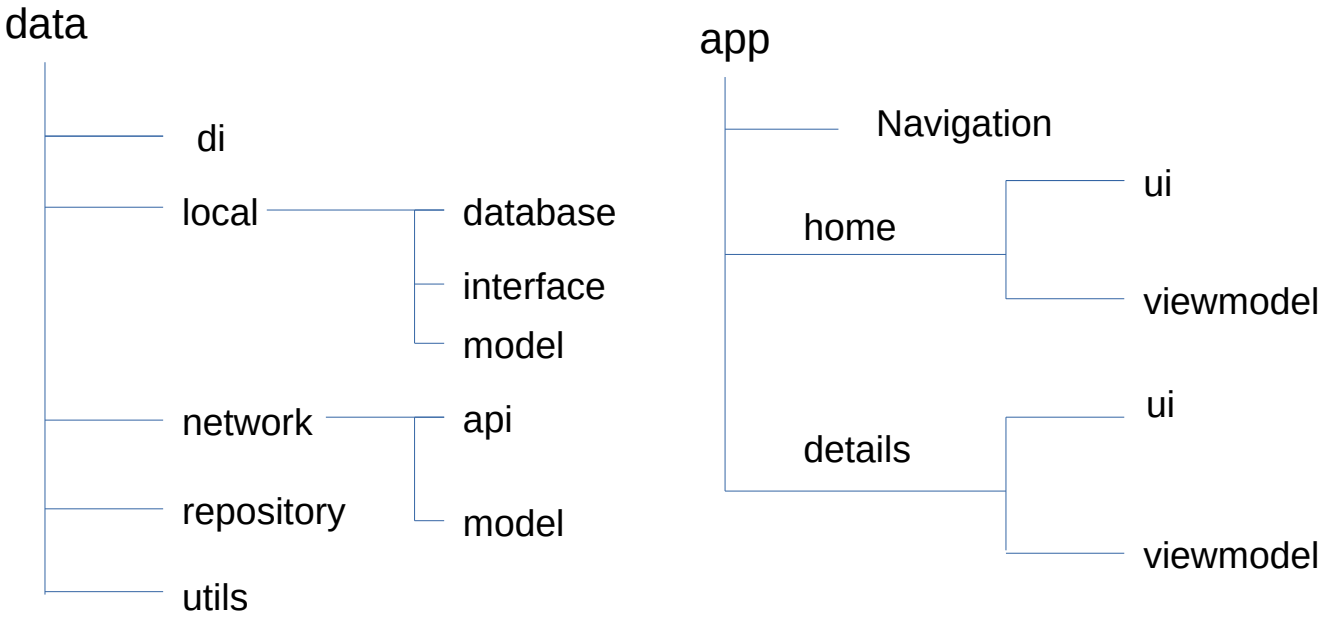
- Dagger : La puissance de Hilt permet une injection de dépendance plus facile et rapide. Cela réduit le code et facilite l'ajout de dépendances. Cela m'a permis de supprimer les Factory existantes.

- lifecycle-viewmodel-ktx : Cette librairie est utile pour retirer le GlobalScope existant, qui ne détruit pas la coroutine, et le remplacer par viewModelScope, qui permet une meilleure gestion des coroutines et évite les fuites de mémoire inutiles.

- navigation-compose : Cette librairie me permet de n'avoir qu'une seule Activity qui va jouer le rôle de conteneur et afficher les différents écrans de mon application.

- Truth et Turbine : Ces deux librairies ont pour but de faciliter les tests. La première permet une meilleure lisibilité des tests et la deuxième permet de tester les Flow.

Architecture utilisée :



J’ai utilisé deux modules, app et data.

Le premier module, app, est le module principal qui va contenir tout ce qui concerne l’UI et les viewmodels des différents écrans. Ce module est ensuite divisé par écran, pour permettre d’ouvrir le bon dossier en fonction de l’écran désiré, ainsi qu’un package destiné à la partie navigation. Les package de chaque écran sont ensuite redivisé pour séparer la partie ui et viewmodel.

Le second module, data, s’occupera de toute la logique de gestion des données. Il a été divisé en 5 packages. La partie ‘di’ où on mettra toutes les dépendances singletons et les repositories qui pourront être injectés avec Dagger. La partie ‘local’ où il y aura l’ensemble des fichiers utiles pour la gestion de données offline (database, interface, model). La partie ‘network’ sera utilisée pour les API (api, model). Nous avons ensuite un package ‘repository’ pour les regrouper et les retrouver plus rapidement. Et enfin un package ‘utils’ qui contiendra des fonctions d'aide, des classes d'assistance ou des fonctions d'extension qui sont génériques et réutilisables à travers toute l'application. Ce dernier, s’il vient à

grossir, devrait être retiré de ce module ‘data’ pour intégrer un module ‘common’ plus explicite. N’ayant pour le moment qu’un seul fichier, le module ‘common’ n’est pas indispensable mais il pourrait être une idée d’évolution.

Design Pattern :

Mon projet est basé sur l'architecture MVVM (Model-View-ViewModel), organisée en plusieurs couches distinctes, s'inspirant des principes de la Clean Architecture. L'objectif est de séparer les responsabilités, de rendre le code modulaire, facilement testable et maintenable.
La communication entre les couches est réactive et asynchrone, en utilisant principalement les Coroutines Kotlin et les Flows.

Pour connecter toutes ces couches de manière propre et découplée, j'utilise un framework d'injection de dépendances.

D'une manière générale, toutes les classes respecteront, au mieux, le principe du SOLID.

Axe d ‘amélioration possible:

- Création d’un module ‘Common’ pour y rajouter des fonctions d'aide, des classes d'assistance ou des fonctions d'extension qui seront génériques et réutilisables à travers toute l'application.

- Création d’un SplashScreen de démarrage pour permettre à l’application de charger les contenus en arrière-plan.

- N’afficher que les albums dans la liste des favoris avec un bouton à l’aide d’une BottomBar.

- La possibilité de rajouter un commentaire sur chaque album dans la page détails, ce commentaire sera ensuite visible dans la Card de la liste.