Moneytree Light App

Moneytree light app that displays accounts and transactions for Japanese users

Objectives

I mainly focused on following traits while designing and implementing this project.

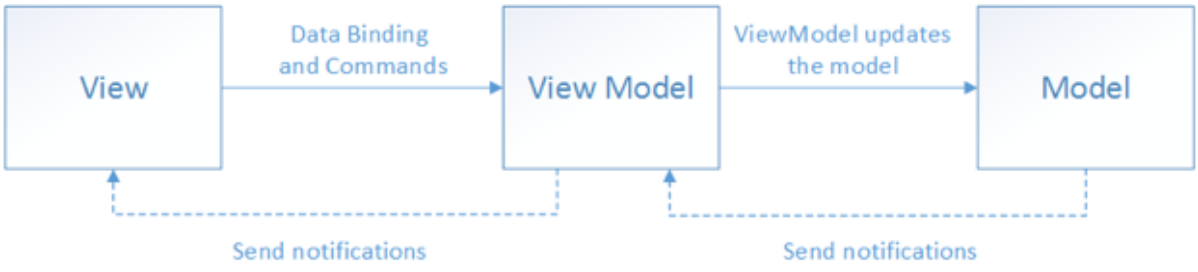
- High maintainability
- High testability (Low Dependency)
- High Performance
- Low memory usage
- Less boilerplate codes
- Can be easily integrated to use network response instead of using json file

Architecture and Libraries

To achieve the objectives above, I chose to use followings. Here's my reasoning for each features.

- MVVM - Separate View from Model to have better **maintainability and testability**. More details in Implementation Details below.
- Single Activity Concept / Navigation Architecture Components - To **reduce memory** usage especially when navigating to multiple pages (use replace instead of add when moving to other pages). Easy data sharing between views. Reduce boilerplate codes.
- ViewModel - To store UI-related data. Data does not get affected on **screen rotation** when using with LiveData
- LiveData - Lifecycle aware observable data holder, **no memory leaks**. No crashes due to stopped activities. Always up to date data
- ConstraintLayout - Allows to position and size widgets in a flexible way. Also can have **better performance** by having flatter view (with less hierarchy)
- ListAdapter - Computing diffs between Lists on a background thread when submitted list is updated. Animation is provided when there's change in the list. Nearly identical to PagedListAdapter, it can be easily converted to it when Paging is needed.
- DataBinding - 2-way dataBinding and easy access to child view when there's nested layouts
- Dagger2 - DI to have less dependency and **increase testability**
- Android KTX - reduce boilerplate Kotlin codes
- AndroidX - Fully replaces the Support Library by providing feature parity and new libraries. All new Support Library development will occur in the AndroidX library.
- Retrofit2 - Type-safe HTTP client for Android. (for future use)

Implementation Details



sources : https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

- **Fragment** will be used as UI Controller that displays data from **ViewModel**
- All UI related data is stored in **ViewModel**
- **Repository** being used to provide data to ViewModel. Testability and Maintainability can be increased by decoupling the application from the data sources.
- LiveData can be shared all across the app by using **Shared ViewModel** in Single Activity Concept design by setting lifecycle owner to **activity**

Implementation Guideline

- **View** will be used for displaying UI components by using data from **ViewModel** (not Model dependent)
- **ViewModel** classes will be used for storing data and requesting data to **Repository** only (not View dependent)
- Use **ConstraintView** for complicated views to have more flexible ui modification and flatter layout instead of nested layouts (LinearLayout can be used for simple views)

Folder Structure

- adapter/ - store view related adapters
- constants/ - store const vals
- di/ - store Dagger2 related files
- factory/ - store factory classes, such as VOListFactory that converts response list to VO list
- network/ - store network call related files, such as response model, Retrofit2 related files will be added in the future
- repository/ - store repository files
- util/ - store utils or helper files
- view/ - store activity, fragments and vo
- viewmodel/ - store viewmodels
- androidTest/ - store UI tests
- test/ - store unit tests

Lesson Learned

- I have decided to use **ListAdapter** for the benefits such as less boiler plate codes, DiffUtil support and Animation on change. However, while I had to implement the list according to the requirement, I have realized ListAdapter may not be a good choice for the list view that have header and its child items, because it is not possible to send list of different typed object when using submitList() and bind different layout using getItemViewType() (while it can be achieved by sending list of data when creating adapter and distinguish data by viewType without using ListAdapter).
  In order to satisfy the requirement by using ListAdapter, I have used single layout to have both header and item view and update visibility by setting isHeader Boolean value on View Object.

  Conclusion – ListAdapter might be more suitable for simpler lists, it could increase implementation complexity if we have to implement customized list with header and its child items, such as StickyHeader. (If there is solution to this, please let me know ☺)

Completed Requirements

- Data Story
- User Story – 1, 2, 3
- Accessibility on AccountListFragment and TransactionListFragment
- Unit Tests
    - CommonUtilTest
    - VOListFactoryTest
- UI Test
    - HomeActivityTest (**Espresso**)

Next Step

- Improve performance on Transaction list display logic in VOListFactoryTest – for loops need to be reduced
- Increase Unit Test coverage, such as Fragments, ViewModel, Repository and Models
- I deleted following test codes during development due to having problem on reading assets when using mock viewmodel/repository(NPE). I couldn't find the proper solution for this, but this can be fixed by using mock server as this example in the future (when there's JSON network response are provided)
    - MainRepository
    - AccountListViewModel
    - TransactionListViewModel
- Integrate network request to get JSON response from the server by using **Retrofit**
    - my github repository can be referenced
- **Paging** - I have implemented Pagination using JetPack's Paging library in my github repository. This can be implemented easily if api response supports it.
- Use **Coroutine** library to make heavy list display logics to run on the background thread, which also simplifies implementation without having to write callback codes.
- **Realm** or **Room** to store large user data to have better performance and access to old data without having network requests

References

- Architecture and Coding style - https://developer.android.com/arch