Candidate Code Exercise

Character View Android App

Comcast

Sean Najera
Android Developer

T 702 375 2921

najera.sean@gmail.com
https://github.com/flopshot/xfinity

Overview

The function of the app was to display a set of characters from different television shows. The style of the app was a single master detail flow of the list of characters. The app had two product flavors: The Wire & The Simpsons. The app also had two separate form factors: one for smaller mobile devices & one for tablets/phablets. The features of the app included the ability to bookmark/favorite characters as well as searching for characters via search bar.

App Architecture

The app UI architecture was designed with Model View ViewModel or observer pattern. I chose this because the MVVM observer pattern in Android is very quick to implement with api features such as LiveData/ViewModel/Room and Room that lend themselves to quicker development time.

The model for the app implements a repository pattern that handles where to get data for the UI. The repository can simply call the database for local storage or create an ApiResource that fetches data from the api.duckduckgo.com character api, stores it in the database, and returns the data as an RxJava Observable. This observable will listen for changes on the Database and update the model when changed, even after the network call.

The ViewModel Stores the UI's data from the model and handles configuration change persistence. The view model is where the observable's emissions are converted to a LiveData stream for the UI, as LiveData is lifecycle aware.

In the app's master detail flow style, the list and the detail Fragments are both separate views that contain their own ViewModel and only have a reference to their respective view models. The views update and receive updates from the view model by observing the data stream and feeding data back into the Stream.

Navigation

The list and detail fragments are controlled by the NavigationDrawerActivity, This single activity is not a view, but rather a controller class for the navigation logic of the app. However, most of the navigation and Toolbar-View logic is delegated to two other classes called the NavigationController and the NavActivityUIController. These delegates have a very loose reference to the activity that can be easily de-coupled if navigation implementation needs to change.

The view fragments have references to these delegate navigation classes via dependency injection with Dagger. This way, the fragment views can update the navigation activity and still be loosely coupled

Features

The app has two other features
1) Search:

The app's toolbar can be converted to a search bar when the magnifying glass icon is pressed. The user's input makes a call to the network to retrieve the most up to date character list and then searches the local database using a query. The result will update the list fragment with the result characters that the user can view in detail in the detail fragment

2) Favorite:

The user can mark a character as favorite in the detail page that will show as favored in any list item with a star next to the character's name. Since all views' connections to the model are observed via LiveData, any currently and subsequently active view will receive an update when a character's favorite status has changed and update its view accordingly.

Product Flavors

The app uses the different source sets and the gradle build system to dynamically switch the resource of different apps. We first define the different product flavors in the app level build.gradle file along with high level differences like api urls.

We then define different resource sets with different titles, strings, drawables, styles, and colors

We can then preform a grade task to assemble and install the different flavored apps

Form Factors

The app has two different form factors for mobile devices and phablet/tablets.

1) Tablets/Phablets:
Any device with a 7" screen or more will view the master detail style in two separate panes. In landscape, the left pane will be the list view and the right pane will the the detail view. In portrait, the top pain will be the detail view and the bottom will be the list view.

2)Mobile devices:
All other devices will show a list view or a detail view screen one at a time.

It should be noted that neither the fragments nor the views of the fragments ever have reference to the form factor logic. This is because all form factor logic is in the Navigation Controller and NavActivityUI controller. Those delegate classes handle which fragment transactions and title strings to use in the case of different form factors

Future Work

1) Large List Fragment State
By far the most serious concern is that the current list fragment holds quite a bit of state. Not too much, but not scalable either. This list fragment needs to know if the list is a FAVORITES

list or a SEARCH RESULT list, etc. Breaking this state up into a separate List Fragment delegate class will be necessary as the types of lists it can show grows

2) Testing, Testing, Testing…
While there are unit tests for the model and repository layers, there are no UI unit tests.

3) Will need to change to different architecture as app grows
While this architecture works for the simple case, a combination of MVI and Clean Architecture need to be implemented to handle the use cases and interactors of more Model Entity objects and more advanced ui states.

4) Minor nit picks
For the sake of time, these features were omitted, but really are an eye sore:

a) No empty-list data in the list fragment when the model returns nothing
b) No global network state listener. The network call logic will still work, but if device is not connected, we need to prompt the user
c) Shared element transition with the star in the detail page. Too jarring other wise
d) Pull down to refresh list. No brainer