ScotiaBank - GithubInfo Architecture and Decision Documentation

Core Architecture

- This app follows MVI with repository pattern.
- Compose is used for creating the UI and hence the decision to choose MVI
- By exposing onEvent method in the root composable and by updating state only in the view model we enforce Uni Directional Data flow which removes a lot of ambiguity and confusion.
- The state is defined and trickles down to individual composables and events are propagated up to the root composable which invokes view model.
- By following repository pattern we have clearly segregated data source repository and view model.
- The functionality was simple enough to avoid usage of UseCases which was thought of initially but later dropped as it would be over engineering.
- We can introduce use case layers easily at a later point if needed and we will conform to Clean Architecture when necessary.
- Hilt was used for dependency injection.

Main Screen

- Enter a github user id
- Search button
- Displays user avatar and user name followed by a list of repos
- If the user has more than 100 repos the network layer calls the paginated api until all repositories are fetched.
- A technical decision was taken to not implement pagination since for most users the number of repos will will be less than 100 and also due to another constraint of finding the total forks across all repositories to assign a star badge to the user in the detail screen and to display the total forks. So the approach taken was to request for 100 per page and if it exceeds 100 we hit the next paginated endpoint and continue this until all repos are populated.
- We didnt want the user to wait until all of the paginated api calls were completed. Instead once the first page is received we immediately emit the response and is consumed by the state and thereby starts showing in the composable. As and when more pages are received we keep emitting the values. Since we use id as a key in LazyColumn subsequent emits wont trigger recompositions also.

Main Screen Loading Indicator

- A basic circular indicator is used for now. We can implement a shimmer modifier for better UI/UX and to improve perceived delay in loading of items.

Details Screen

- 3 major approaches were thought while implementing this screen.
    - RoomDB for saving repo details which acts as a source for the details screen wherein we pass the repoid while navigating to the details screen and we fetch the repo details from room db.
    - SharedPrefs - Since offline support was not a requirement adding RoomDB was thought of as an overkill at the moment and instead relying on shared prefs to write to a key value and then retrieve it based on the repo id was also thought about.

    - Using a shared view model - Since the entire repo details are already loaded in memory, relying on the exact same view model in the detail screen was a valid approach. The main concern was that since the app is a single activity compose based application the view model will always be in memory if its shared on activity. Instead the viewmodelstore owner was defined as the navigation graph. This way the view model will be garbage collected when the navigation graph is popped out or replaced by another one. This will be a good approach for the current situation and in future even if we have other screens or features switching to a different navigation graph would release the view model from memory. Hence it was decided to proceed with this 3rd approach of shared view model.

Testing

Unit Tests

- MockK, Google Truth and Turbine were used extensively to cover the critical logic in code especially the main view model and network data source
- Code Coverage hasn't been checked and didn't aim for a specific code coverage percentage instead focused on getting the major functionalities under the unit test umbrella
- Added fixture library to randomly generate data classes and dates for creating Utils test.

UI Tests

- A wrapper root composable was used which holds the reference of the view model. No other composable down the semantic tree holds a reference to the view model. This refactoring made it easy to test UI by just mocking state.
- Similar approach was taken to test the Detail Screen and focus was given for testing the badge display scenario wherein total forks were less or above 5000.

Localization

- In order to support localization care have been given to not add any hardcoded strings inside code. Instead all strings are defined and referred from strings.xml

Multipane Support

- A technical decision to focus on phone screens was taken even though a list-detail approach for supporting devices with expanded width or foldable screens was initially thought of. This can be done in future by making use of WindowSize class and by using compose list - detail component. (ListDetailPaneScaffold)

Image Drawables

- An image placeholder png is used which ideally can be replaced with a svg or webp to reduce size further.

Design Anomalies to be addressed

- The design closely resembles Material2 components and the project uses Material3 library and some of the default UI appearance might seem a bit different eg. The space between the label and the bottom

line in text field is a bit more than the screenshot shared in the assignment pdf.

Future Improvements

- Add icon
- Add deeplink support
- DetailScreen is not a standalone screen since it is coupled with MainScreenViewModel. To address this either RoomDB implementation or a simpler Shared Pref approach can be seeked.
- Ideally the star badge should be displayed on the main screen instead of detail screen for better UI/UX and the total forks info is ideally part of the main screen itself and placing it inside repo details is not a good placement.
- We can write a github action to run the unit test cases on push to master or merging of pull request to master
- Dependencies in the build.gradle is not completely moved to libs.versions.toml
- Add Multipane support
- Add dark mode support
- Add code coverage report