**Multi-Module with Clean Architecture**

For the implementation of the project, I have followed clean architecture with MVVM pattern.

**Reason for implementing the clean architecture**

* **It makes it easy to unit test the business logic in each layer.**
* **Confirms to Separation of concern principle which makes it easy to understand the code structure.**
* **Makes it easy to modify or extend the code base.**
* **Faster feature development**
* **Makes the system highly cohesive with low coupling.**
* **Easier for the team to work on different features(e.g., by having multiple feature squads)**

**High Level Architecture**

**A diagram of data flow

Description automatically generated**

**Higher layers react to changes in lower layers, where event flows down and data flows up.**

**UI Layer**

**is responsible for implementing the UI elements in the form of jetpack composable, where UI state is maintained in view models using the reactive state flow which exposes the state to view(composable).**

**Domain Layer**

**In this layer, I have created 2 Gradle modules:**

1. **Repositorycontract:- contains the repository interfaces which data layer is dependant on and provides the implementation. All feature modules also depends on repository contract without knowing about the implementation which resides in data layer.**
2. **Domain:- I have created this module to host common use cases if required.**

**Domain layer don’t depend on any other layer except a common module which hosts commonly used utility classes and common model classes.**

**Data Layer**

1. **Hosts the implementation of repositories.**
2. **Repository implementations communicates with API service and is responsible for fetching/pushing data from/to API service and return the data/responses.**
3. **Hosts the network module which is required for making http calls to end points using retrofit.**

**Tech Stack used**

**Jetpack Compose**

* **is used to build the UI elements which observes the changes in UI state in view models and display the data accordingly by using lifecycle aware API.**

**Kotlin Flow - State Flow**

* **is used to manage the UI state of “All products” and Login data.**

**Use case(s) with Kotlin Flow**

* **are injected into view model constructor to fetch/post the request to repositories, after performing business logic in uses case(s), the data is then pushed into state flow to be then exposed to the composable.**
* **In every use case data is emitted as flow, which is then collected and pushed into state flows in the view model.**

**Dependency Injection**

Dagger hilt is used to provide the dependencies required by the dependent components.

**Http client – Retrofit**

**Retrofit is used to deal with http requests/responses**

Error Handling

* When error interceptor intercepts 401, it will load the login screen.
* Ideally we should have the token refresh API, to return the refreshed token in case the token is expired.

**Gradle and Version Catalog (libs.versions.toml)**

**Used version catalog which provides the central repository to keep the dependencies, plugins and their version. Bundles provides us the mechanism to group the dependencies and import as one liner in gradle scripts which makes it very easy in multi-module project.**

**App Theming**

Jetpack compose based app theming is implemented for light and dark themes which depends on systems theme settings.

**Unit Tests**

Are written using Junit.

Mocking

Mock is used for mocking the dependencies which makes it easy to test coroutine based code.

Notes:-

1. The money box test API cloud front returns the responses between 15-20 seconds. You will see the progress bar for a while. I had to use VPN as money box cloud front blocked my IP.