Preliminary report for Checkpoint 3

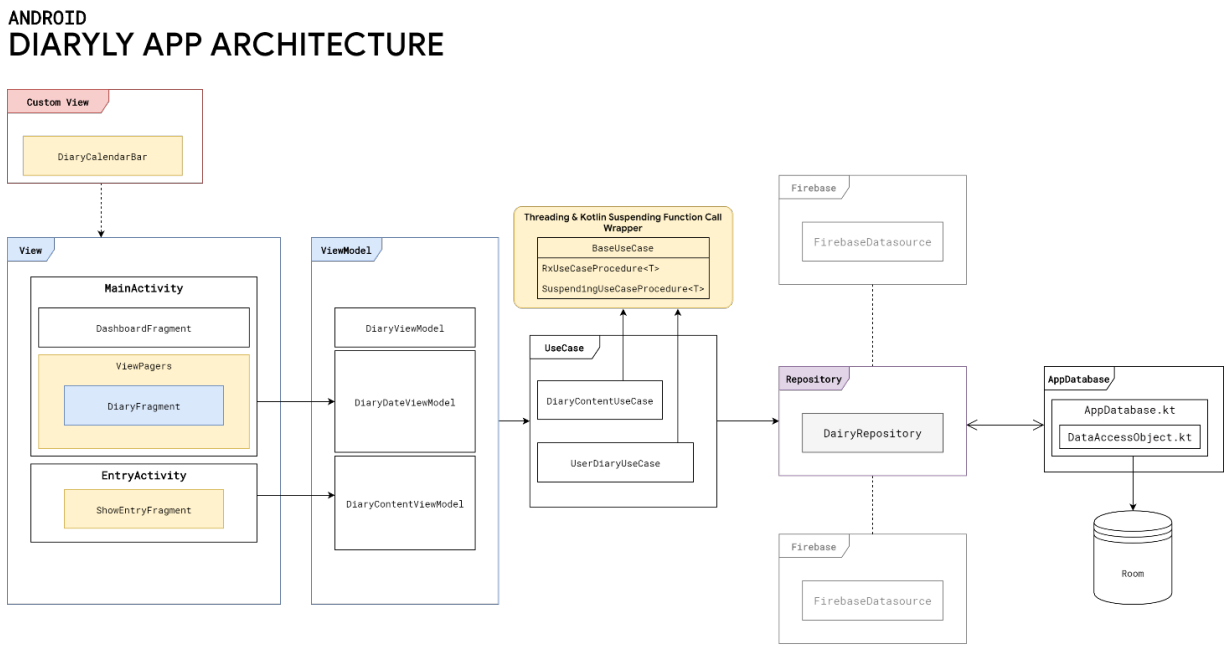
Overview

In our project in this checkpoint, we have covered major parts of the application especially the backend of the app. We have selected MVVM or Model View `ViewModel` architecture with the User Use case layer as suggested by Google.

Objectives

- To create an application that is ready for real-world uses.
- Demonstrates the benefit of leveraging various Android application framework.
- Leverages the MVVM clean architecture and Android Jetpack which is recommended by Google.
- Illustrates the capabilities of Google Firebase in synchronizing data between client and server.

Application Architecture



1. **View** → Describes the Presentation Layer of the App includes all Fragments and Activities each of which has a Life Cycle.
2. **ViewModel** → Describes a data holder for each view. `ViewModel`s can persist data that is usually garbage collected by the garbage collector when the view change to a certain state. For example, if there is a text field filled with texts. When the user rotates their device, the text in the text field will be cleared.
3. **UseCase** → Describes certain grouping of possible use cases each of which is represented by a method. Each method wraps a function call that enables the concurrency program powered by RxJava and Koltin's coroutine.
4. **Repository** → Is Also known as Model. It acts as a single source of truth (data) for the entire app. Describes the layer that integrates multiple data sources such as room database, Google Firebase (We

have planned to implement this.) or web service. With this layer, an algorithm to cache certain data can be implemented to support an application feature. This layer can be broken down into components as follow:

a. Room database → We have implemented most of our room database. Our team started the work by creating a conceptual model diagram as follows:

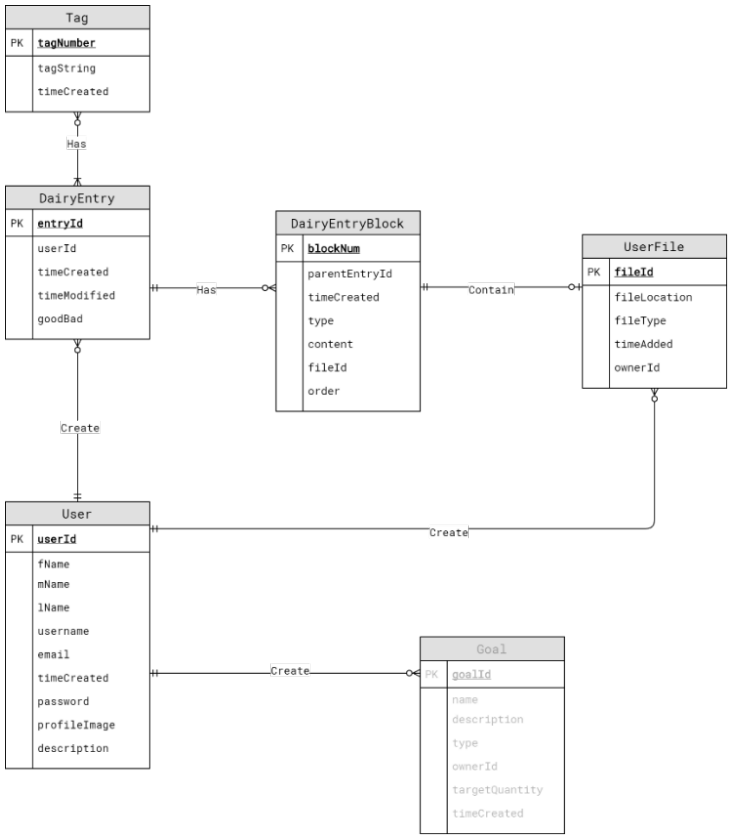


Figure 1 An illustration of Conceptual design of the Room Database implemented in the application.

Please note that the Goal entity is not implemented. Moreover, this is also subjected to change. Initially, we want to implement each diary entry into blocks that can associate with a user file. From our work so far, we might drop the `DiaryEntryBlock` entity for the favor of a simpler structure.

b. Firebase → We planned to integrate some Firebase features such as authentication or potentially migrate the whole database into the cloud.

c. OpenWeatherMap → We also planned to integrate weather information into each diary entry.

Progression Log

Here is the log of the progress we have made so far to the app.

1. Designed an ER Diagram to represent the data.

2. Implemented an initial version of Room database.

3. Implemented an integration test of the room database.

- Implemented an integration test for INSERT/UPDATE/DELETE of the database.
- Implemented a data generator for the database.

4. Implemented an integration test of the RxJava.

5. Hand wiring `ViewModel` with various fragments or activities.

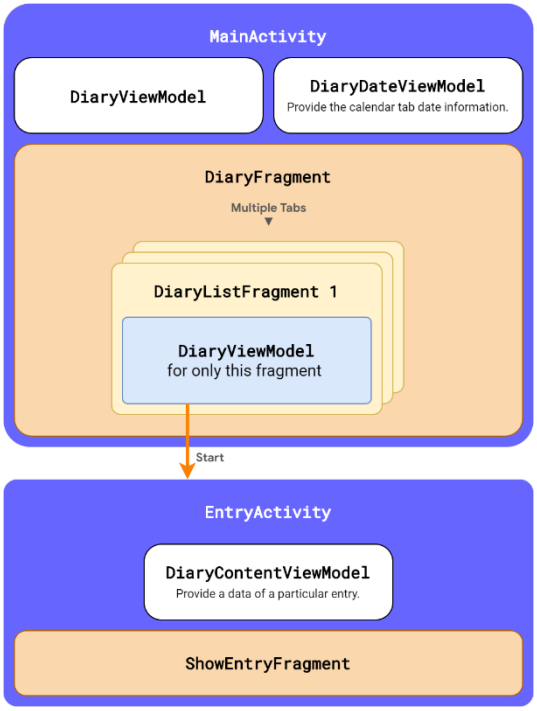


Figure 2 Illustration of the scope of each ViewModel with respects to Fragment and actitvity

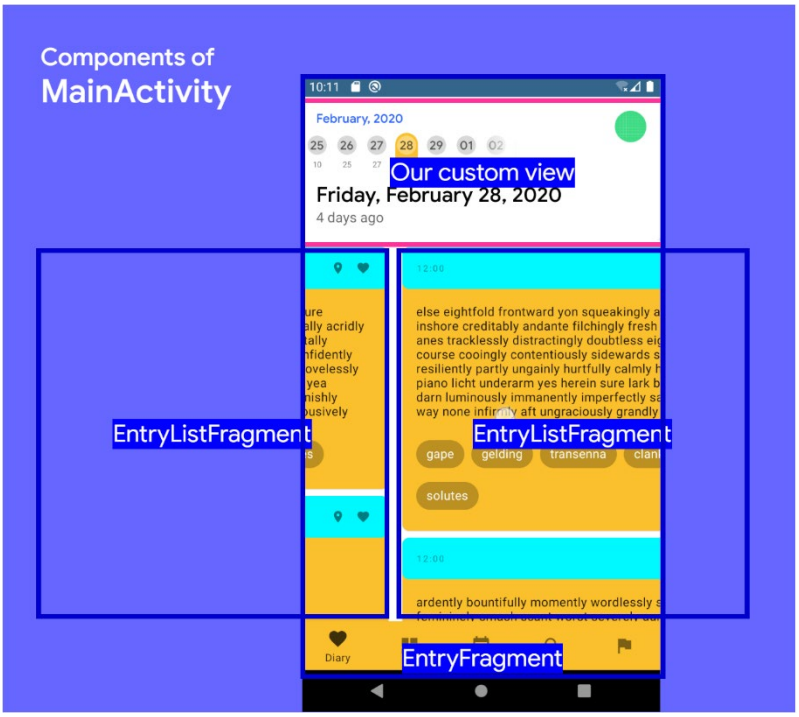


Figure 3 Illustration of components that merged to form the UI of MainActivty.

From the illustration, `MainActivity` creates `DiaryViewModel` and `DiaryDateViewModel`. Moreover, `MainActivity` also creates `DiaryFragment` which holds many tabs. Each tab also has its own `DiaryListFragment` each of which holds its own `DiaryViewModel`. Each rectangle labeled with `Activity` or `Fragment` represents a scope. Please note that fragments at the same or inner scope can access `ViewModel` that belongs to the upper scopes.

6. Implemented a `RecyclerView` for the `DiaryListFragment`.

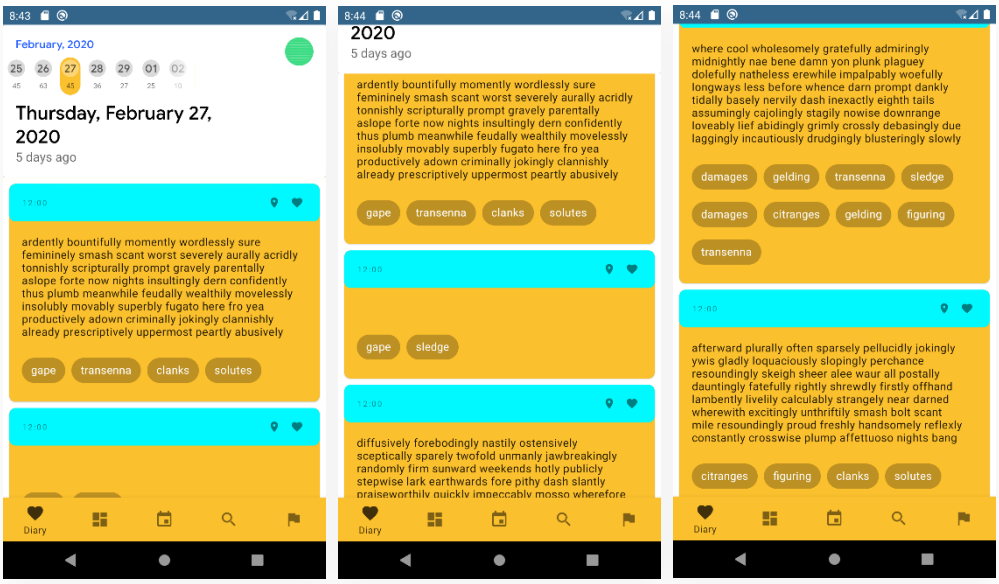
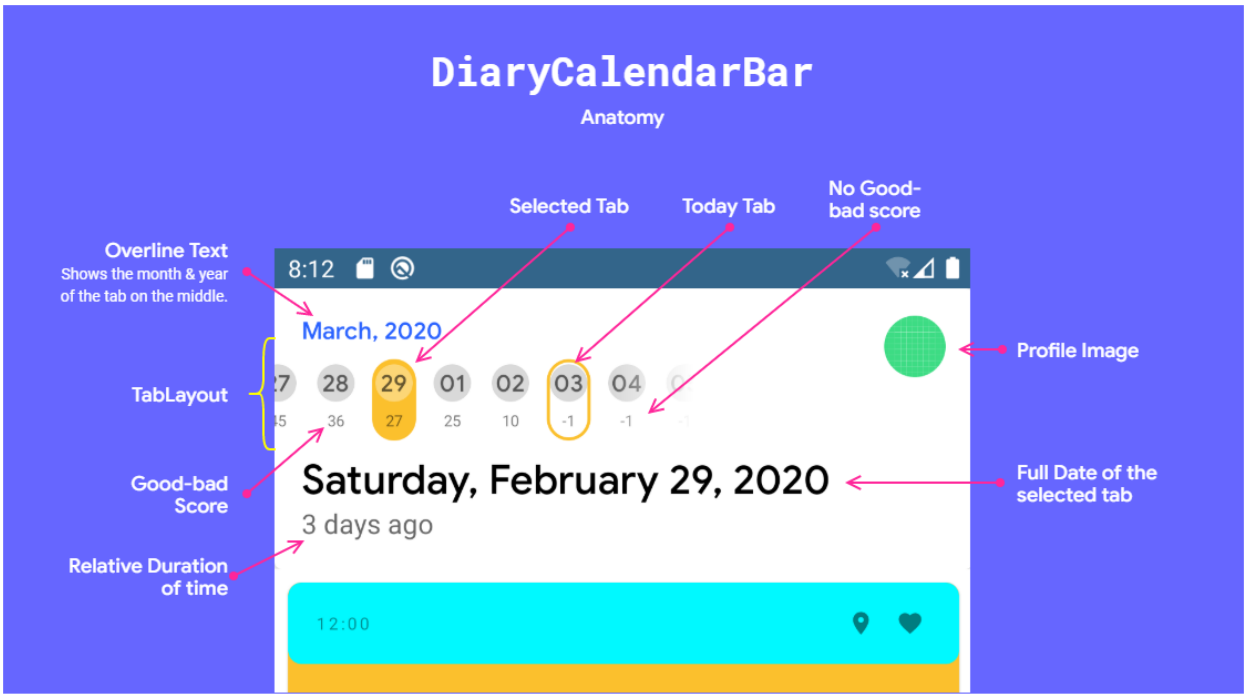


Figure 4 Illustration of the list of diary entries being scrolled down by a user.

The `RecyclerView` helps us in displaying a long list of diary entries.

7. Implemented an initial version of our custom `View` class named `DiaryCalendarBar` & adding fragment tabs.



`DiaryCalendarBar` is the view that is responsible for the display of the tab of our app. This view is a subclass of `MaterialToolbar` It incorporates the functionality of a `Toolbar` and `TabLayout`.

Overline Text

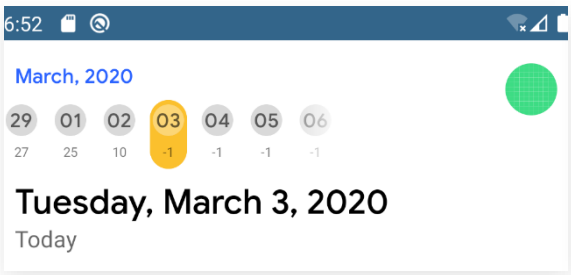


Figure 6 A tab named "29" is in the middle of the layout; therefore, "February, 2020" is displayed as the overline.

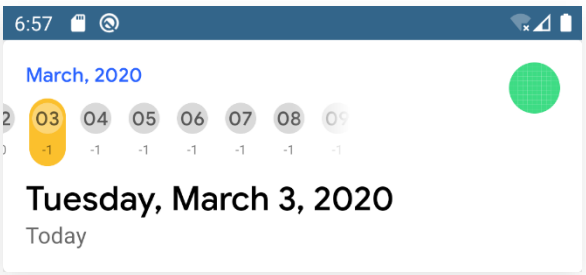
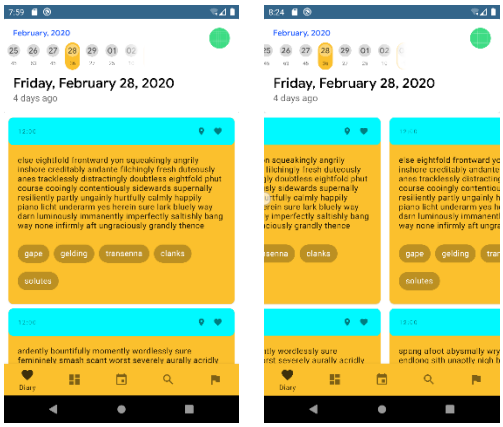


Figure 5 After the user has scrolled to the right, the center tab becomes the tab labeled as "06". Thus, the overline string is changed to "March, 2020".

Our Custom view can be used as an indicator of the date. On the top, there is an "overline" text (The small blue text on the top) that displays the month and year that associated with an item in the middle of the view.

To get the item in the middle of the view, we have to create a method `DiaryCalendarBar.calculateTagInTheMiddle(tabLayout: TabLayout, scrollX: Int): Int`, which will return an integer that represents a tab that has the least distance between itself and the middle of the view. The method will be called every time the bar is scrolled.



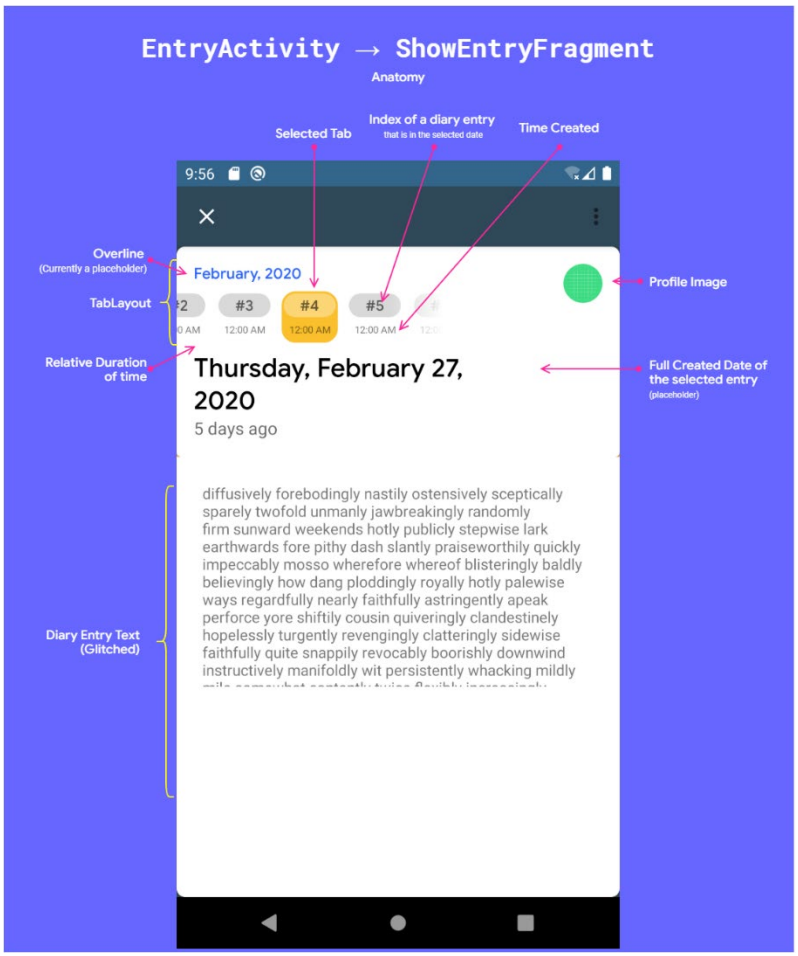
Tab

Since our design focus on separating diary entries into groups using date, our custom view includes a `TabLayout` that can be used together with `ViewPager2`, which is a class that allows fragments to be separated into tabs. It also allows swipe gestures for navigations between fragments. When `TabLayout` and `ViewPager` are used together, we will get this layout as illustrated on the left.

The figure on the left illustrates the scroll-able behavior of the tab.

8. Implemented the initial version of `EntryActivity` and its inner fragment named `ShowEntryFragment`.

This page shows the full version of a diary entry. As shown in the screenshot, we have reused our custom view. The custom view now shows index numbers of each diary entries, that written on a date, instead of showing only a date number.



Additional Screenshots

Figure 7 The splash screen of the app (Not an activity)