Document of SimpleTimer Project

1. View: xml vs **Compose**

Two ways we could use to create views, either through the traditional xml or the newly promoted Compose.

In the project Compose is applied. It could simplify and accelerate the UI with less code, powerful tools and intuitive Kotlin APIs. (RecyclerView can be replace by LazyColumn and no adapter is needed)

1. Screen arrangement: Two activities vs **One activity + two views**

The app has two screens, we can place each of them in a different activity. Or we could create two views that represent the two screens and put them into a single activity.

In the project, one activity + two views is applied.

With the help of Navigation, we could easily navigate between composables. Even more views are add, including the top or bottom bar widgets, they all are handled by only one NavHostController.

1. Pattern: MVVM

With Compose and Navigation applied, MVVM is the No. 1 choice

1. Communication between two screens:

Although navigation allows different screens to pass some primitive parameters forward or backward, it is found not a good choice for this project. When a new timer is created, we will navigate back to the show timer screen and show the timer and automatically start count down. If we use navigation to pass parameters back to refresh the show timer screen, when the app is brought to foreground from background, the show timer screen will refresh again. Because the parameters passed to show timer screen remain unchanged.

The solution is to create a separate viewmodel attached to activity. The two screens could pass parameters to each other and change their values through the viewmdoel.

1. Device storage: SharedPreference vs Database

The app requires to save some data to the device, considering only one timer is shown, so the SharedPreference would be enough. But this lacks extendibility.

Database could be a bit overkill here, but it could be easily extended to handle multiple timers. With the help of Room, operations of database could be easily implemented.

1. Access to database: Coroutine

Access to database can be time consuming and should not be invoked from UI thread. In the project, Coroutine is applied to access the database. Coroutine provides an easy way to do asynchronous tasks.

1. Performance improvement: limit access to database

Even with the help of Room and Coroutine, database operation is not a pain point, but access to database is time consuming and frequent access to database could lower the performance of the app. Also, race condition can happen when countdown timer and UI are both trying to modify the data in database.

To avoid excessive access to database, in the project, the data of the timer is first grabbed from database and then saved in memory. Then the countdown timer and the UI only communicate with the in-memory data. In this way, the access to database is very limited and the overall performance of the app won’t be much affected. There may be relatively more memory used, but it is total worthy compared with what we could gain.

1. Extendibility:

Although the app only shows one timer, the solution is expected to handle multiple ones. However, in the solution there is only one CountDownTimer to handle different timers (delete and add new time). If we really want to make it for multiple timers, we need to refactor this part by probably creating a different CountDownTimer for each timer. This has not been done in the solution and could take more time to implement and test.