COMP3018-MDP Report: Activity Tracker

TERENCE BURNS – 14309411

Design and architecture

General architecture

The task involved building an application that can log the user’s activity, enabling them to view and inspect the current activity they decided to record and any previous activities with it. In order to persist the user’s activities and exchange this data with view components as efficiently as possible, the application is built upon the MVVM structural pattern with the Room persistent library to build an abstraction layer on top of the underlying SQLite database.

Three fragments define the entire UI of the application, each with its own ViewModel to maintain its UI state in events such as a configuration change. Each ViewModel needs access to the database and as a consequence to avoid code duplication and enable reusability, a single instance of a repository is maintained within an application container. This container is instantiated and stored within a custom Application class, thus making the repository dependency available throughout the app. A custom ViewModel factory was also created in order to provide the ViewModels with the single instance of a repository during the ViewModel creation. The manual injection of the repository dependency reduces tight coupling between these components and maintains a separation of concerns. In addition, a pool of threads is also within this application container, and is used when accessing the database, and, for executing the service’s tasks to prevent overhead on the UI thread.

Components

Activities/Fragments:

The application was built with one main activity that maintains three fragments. The fragments include a Recording fragment that handles the interface for the user to start recording an activity, a Feed fragment that presents all the user’s activities using a RecyclerVew, and finally, a profile fragment that provides the user with statistics such as - what was their longest distance run, or what was their fastest min/mile pace.

As said previously, the repository is within each fragment’s ViewModel. Within the record fragment, LiveData fields are fetched from repository (where they are continually updated by the service) via the ViewModel and is observed in order to update the UI only in the event of change (Figure 1). Within the feed and profile fragments, the UI displays data resources as maintained by the database. Similarly, the data fetching process is delegated to the repository. In order to prevent the blocking of the UI thread, a pool of executor threads is utilised. In addition, data fetched from the backed is cached into the repository and then into the respective ViewModel. This use of lifecycle aware components and caching was implemented to avoid wasting cycles reloading data from the database, especially in the event of a configuration change.

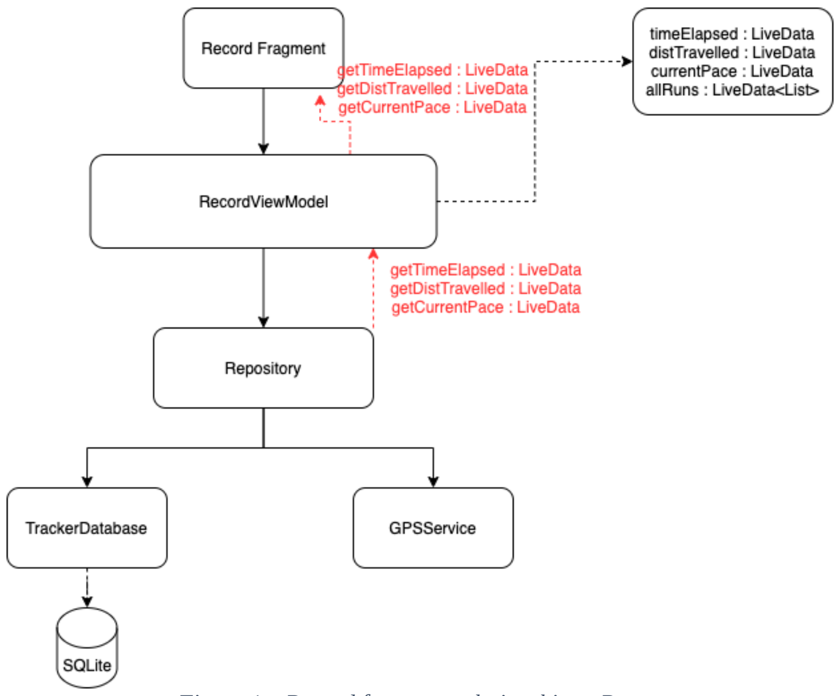

Figure 1 – Record fragment relationship to Room.

Service:

A service is used to handle location and time updates. The service is first started, to run indefinitely, and then binded immediately after to allow for IPC via a registered callback, which is instantiated in the repository. The service is started via a call from the Record fragment, relaying up to the repository, where the service connection is handled. The service is also foregrounded, and a notification is displayed to make the user aware that this application is currently consuming system resources. Foregrounding the service was done to reduce the likelihood of the service being killed, even if the user moves away from the application; we want the user to be able to continue to track their activity even if they are not directly interacting with it. The service is only stopped when the user initiates it by pressing the stop button in the UI. In the case where the user presses stop, the service is unbound and stopped explicitly via stopself() on the onUnBind() callback.

As previously mentioned, to avoid blocking the UI thread, the handling of the time increments is implemented via an inner class that extends a worker thread. Also, the executor threads are used when handling the location updates, as retrieved by the GPSLocationListener.

LiveData fields within the repository are updated by means of the callback. When the user ends the recording of their activity, the storing of the data into the database is delegated to the repository so as to continue to adhere to the single responsibility rule (keeping all data-handling logic within the repository). As illustrated in Figure 2, ending the recording will trigger the insert method, which collates all the data fields and then creates the appropriate data entities and inserts them into the database by means of the corresponding DAOs.

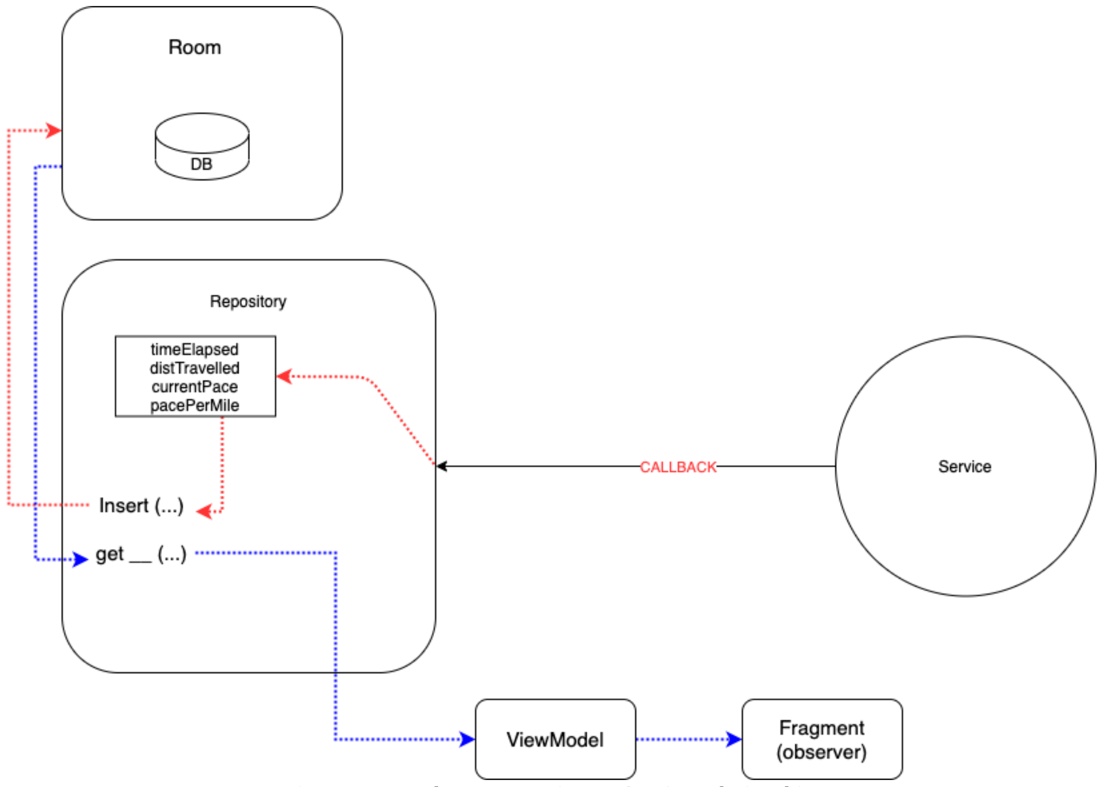

Figure 2 - Database - Repository - Service relationship

Content provider:

A content provider was implemented to provide access to external applications in a safe and structured manner. The provider only allows external applications to query results, rather than to modify or insert entries because there is no reason for other applications to edit the database outside the scope of the hosting application. In the result of a query, the provider accesses the database directly.

Evaluation

Functionality

The application is designed to be simple and user friendly. The UI is defined by three fragments with lateral navigation by means of a bottom navigation bar. All fragments are instantiated when the hosting Activity is created and doing so prevents the need to constantly create and free memory every time the user navigates throughout the app. From viewing competitors, data including distance travelled, duration, and pace were used, and as a result were integrated into the functionality of this app. It is assumed that a user would use the feed and achievement fragments briefly as a way to view their performance, and with this reason, multiple activities were not opted for; creating individual activities (and moving through each lifecycle event) produces greater performance expense that can otherwise be avoided.

The application uses location services to track the user's progress. Accessing this sensitive private user data results in the use of a runtime permission. A permission dialog is prompted to the user before recording is initiated and is done in order to fulfil Android's goals of allowing the user to *control* over their data, remain *transparent* on the data the app uses, and finally, ensuring the app uses only the *required* data to perform the task.

The app also provides a breakdown of each logged movement via a separate Activity component. The Activity is started via an explicit intent, therefore providing greater security as we can ensure what Activity responds to the request. An explicit intent is also used to start the Service for this reason.

The functionality is a foundation for users to start analysing their activities. A few improvements would be to add charts/graphs within the breakdown of each movement, therefore providing information that is both informative and accessible. These visualisations could provide an overview of how their pace varies across the movement or an analysis of how the terrain changes etc. Better customisation could also be added, allowing the user to change from metric to imperial measurements and vice versa and even allowing the user to set a given pace and get notified if they are going too fast or slow within a given margin.

Performance

When implementing the feed, a RecyclerVew was used in order to improve the performance and efficiency during the inflation of each view element. In addition, a separation between the UI and the backend/state logic was maintained by keeping all state handling implementations separate from the Activity/Fragment components. Also, as previously mentioned, threads were used as a way to keep long running tasks from blocking the main UI thread.

In hindsight, the content provider could have been used for both internal and external access to the database, thus providing a single point of access and therefore easier to maintain. Then, a CursorLoader could have been used in replacement to the executor thread to run the asynchronous calls. Overall, persisting data was the fundamental challenge in this task, therefore, special care was taken with regards to the lifecycle of the Activities/Fragments throughout and was handled mainly by the use of the MVVM architecture and lifecycle aware components that allowed to free memory as efficiently as possible.