Fitness Tracker Report

System Functionality

Main Activity
The user can set their weight and can view previous courses within a recycler view, where a dropdown menu can be used to sort them by their statistics. They can go on to either view a completed course, start a course, or view their statistics showing their course progression.

Permissions
When starting a course the user is prompted to enable permissions for location, storage, and the internet. This is required as the location tracker and map rely on the user's current location, while the image importer utilises external storage access and, the weather API[1] uses the internet to perform JSON requests.

Service
The service manages the location and timer updates as well as the notifications given to the user. Frequent location updates are requested in intervals retrieving the latest latitude, longitude points of the user. The timer utilises a background thread which updates a counter every second. The location updates and timer can be paused and resumed.

Notifications are created when the service starts, which update the user on their running statistics whether they are on the app or not. From the notification they are able to pause and un-pause the current course, in addition to being able to finish their run, saving the course and finishing the activity. The notification is foregrounded and when clicked will allow the user to return to the activity.

Broadcast Receiver
The notification button intents and the data generated through the service are broadcasted and obtained by the receiver within my New Course activity. The data is then passed to the ViewModel to be handled. I make use of Intent filters to limit the data being retrieved and so other applications can't make use of my broadcast calls, making the receiver act as a "Local Broadcast Receiver". The receiver was an efficient and easy to implement method for data transfer between the service and activity, which is why it was chosen over a Handler.

New Course
If they press start, the start date time, current weather from the API, and the user's weight is given. In addition, the service is started, tracking the time and location. A table displays their current statistics: distance, time, current pace, and calories burnt which is calculated utilising a formula linking their current pace and weight. A recycler view will showcase running splits for the user, where every 100 metres they receive their time change and pace during that split. A map is shown which plots the user's route in real-time using the Google Maps Android SDK[2]. Once a run is finished, they can alter their course name, associate an image, and give a rating. When they press finish, the course and it's statistics are saved to the database and the user is returned.

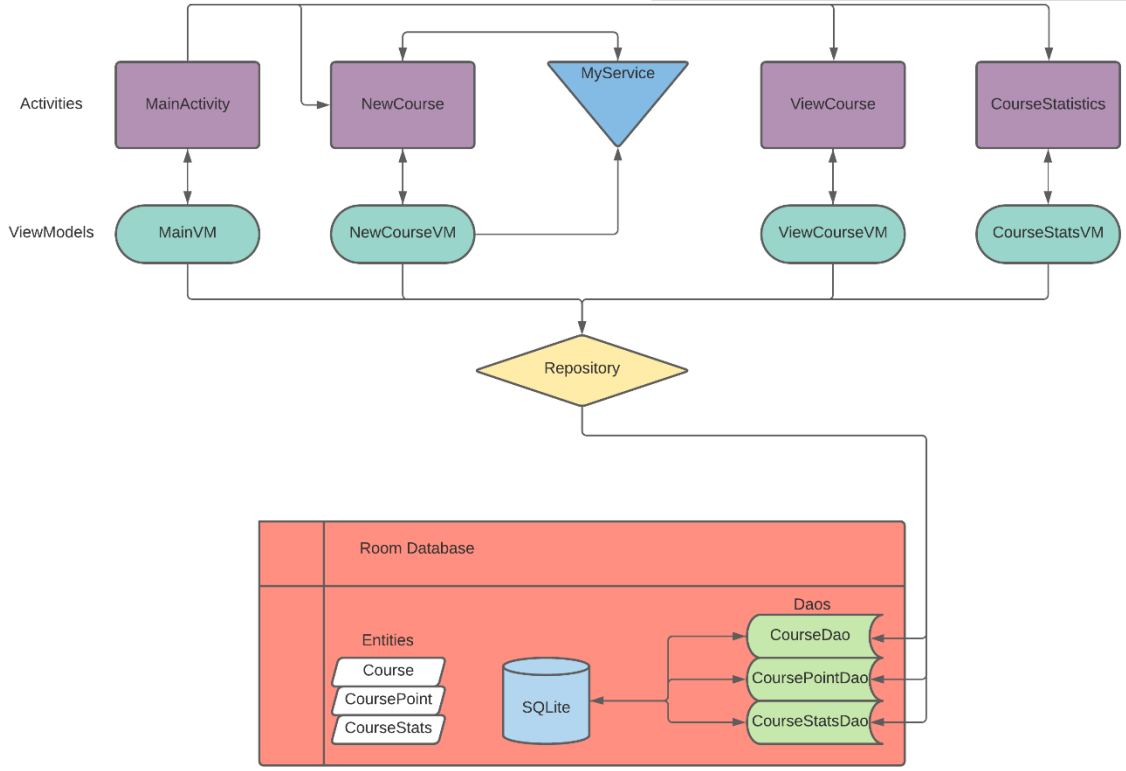
[1] https://openweathermap.org/current
[2] https://developers.google.com/maps/documentation/android-sdk/overview

View Course
The user is able to view all the statistics from the selected course. The route taken is plotted on a map and a recycler view of the 100m splits are shown. They can change the rating given and the name of the course. The user can either return, where any changes are saved, or delete the course, removing it from the database.

Course Statistics
They can view the overall statistics over all of their runs giving the total distance, time, calories burnt and average pace. A graph of their progression over time is shown where it can be filtered by a specific statistic with a dropdown menu.

System Architecture

Figure 1: System Overview Diagram



Room Database
Room is an implementation of SQLite where Room acts as a database layer on top of SQLite. Used as it allows for easier compatibility with SQLite, by reducing the amount of repetitive code used (boilerplate code), in addition to query validation at compilation time.

Three entities are created, one to hold all the courses ran with their given statistics and data. The second table held all the (longitude, latitude) points gathered throughout the run along with the time per point. Since there are multiple points per course, I assign two IDs, one a primary key for the individual points and the other being the course ID the point relates to. Finally, the third table follows the same ID assignment as course point table, however, storing the 100m split data, which includes distance, the time change and average pace. The latter two tables share many-to-one relationships with the main course table. A Dao is created for each entity and allows for querying such as inserting, updating, deleting, and fetching. I am able to fetch courses by their courseID or fetch all courses. I also query for ordered data to be used by my dropdown sort menus. The Database class acts as the main access point between SQLite and the application. It is declared as a

singleton as only one instance should be permitted during runtime. The Repository acts within the Room Model MVVM structure, abstracting access between the ViewModel and the multiple Daos.

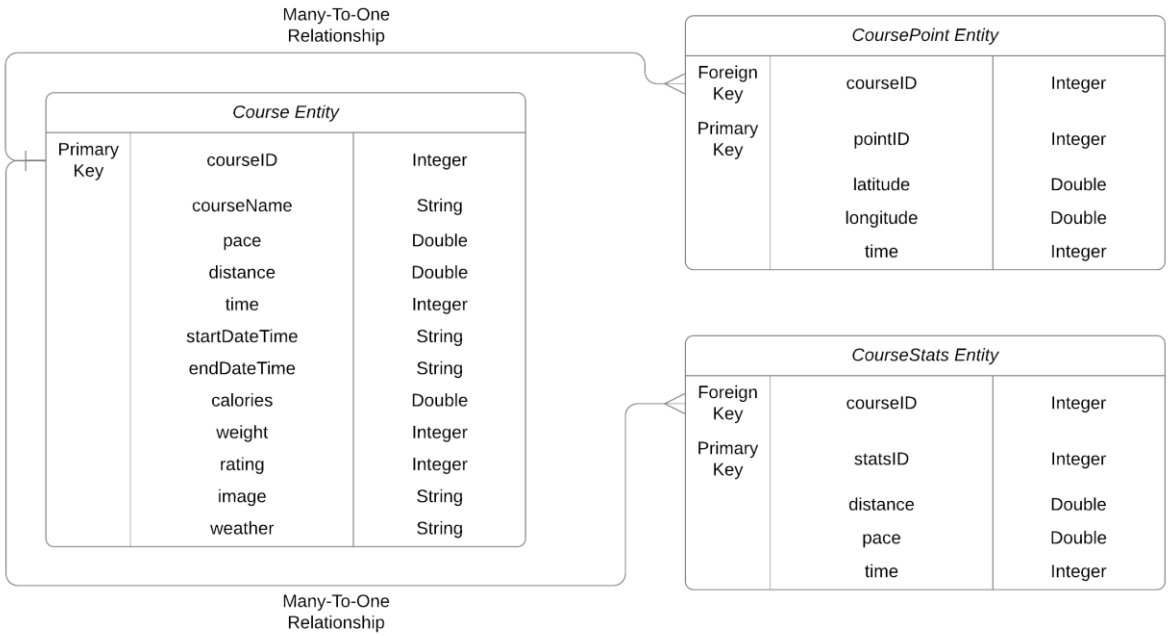


Figure 2: Entity Relationship Diagram

Model-ViewModel-View Design Pattern

MVVM separates the view from the back-end logic, allowing for a structured application which is more easily maintainable. As stated, the Model is the repository where it acts as the access point for data within the Room Database. The ViewModel provides data to the views, functioning as a communication layer between the repository and view. It hides the backend from the view and survives device configuration changes. Data logic is performed within the ViewModel, allowing calculations such as for my user's statistics, to be completed. The held data is then either passed onto the view or passed onto the database through the repository. Each activity has a ViewModel which extends an ObservableVM class to support databinding. The view is the visible elements in the application, it takes in user input and showcases received data from the ViewModel. Data received and/or any calculations needed to be done are handled by the ViewModel.

LiveData, Databinding and Viewbinding

LiveData is applied within the ViewModel, which follows the observer pattern and is lifecycle aware. It allows the view to observe the latest relevant data required so for instance after a configuration change, data is reinstated to the view as the observer has detected old data. It must be used when retrieving data from the database as queries can't be run on a main thread, where as LiveData employs the usage of background threads. This is why I utilised it to update my recycler views with data retrieved from the database.

LiveData with databinding allows the data to be directly displayed from the ViewModel to the view without the need of observers. I use the databinding in combination with MutableLiveData when calculating the statistics while on a run, allowing the updated data to instantly be shown to the UI. This also increases code clarity, however, in some cases it is preferable to use Viewbinding such as within the recycler view adapters and when utilising immutable data, due to better efficiency and computational speed.

Lifecycles

Log statements were used to track what lifecycles my activities and service went through. This allowed me to set up my Override and lifecycle aware elements. When ending an activity I would call "finish()" which would call the onDestroy() method, closing any connections and removing observers. The user would then return to the parent activity.

When changing the phone orientation, the activity is destroyed, so any data visible on the UI is erased. To fix this, LiveData is used to update the UI after an OnDestroy(), also the ViewModel is able to store all necessary data and prevent it from getting reset.

When a user decides to back out of the new track activity while the service is running, the service is not stopped. This is due to the service having to be stopped at the end of its runtime. To fix this, I implemented the Override method OnBackPressed() in the New Course activity, this allows me to stop the service before onDestroy() is called.

Furthermore, when a user exits through the system, the service must again be stopped. This is done by implementing an Override method inside the service. The function is called OnTaskRemoved() and within, I stop the service to end its lifecycle.

Finally, when linking my service I bind it to my activity, then when a track's started, I create the foregrounded service. In the service OnStartCommand(), I foreground my notification as it is a priority task and so should be prevented from being killed by the system. In my activity's onDestroy() I unbind my service's connection and nullify it. Now when the activity is destroyed the service's connection to the activity is unbinded, but when it is recreated the service is rebinded and a connection is re-established.

Content Provider

Allows the data to be used by other applications, where it is received as tables from the database with the use of Cursors. It let me centralise my data so it can easily be accessed if an application requests it's use. Content providers add another level of abstraction to an application, allowing changes to be made to the database without affecting other applications that are utilising the same data.