Pace - Final Report

Ang Wei Neng, Julius Sander, Tan Zheng Wei, Zhang Yuntong

Overview

Pace is a fitness application that lets users record running routes and share them with friends. With Pace, when a user records his run, his location and timing will be recorded repeatedly at a small time interval. This will produce an accurate mapping of a user's location in their run over time After the run, the user then can then share his running route and pace to his friends, or even share them publicly. Others can then choose to follow the same route, giving them a benchmark to follow during their run.

Terminologies

- Route: The path of a run that a user takes and records.
- Run: Defined as completion of a route. A route can have multiple runs created by different users.
- Checkpoints: Physical points in a run that a runner can reach and record. Checkpoints are stored with other informations, such as speed, time and cumulative distance.
- Follow: An action that a user takes, indicating that the user wants to trace the route that the creator of the route made. Note that "Follow" is a subset of "Run"

Features

To better understand the set of features for the app and the design of the application, the following is the list of features and its use case for the application.

Feature list

1. Run recording and route creation:

A user can choose to create a new route. When he does so, location and timing are recorded repeatedly at a small time interval when he starts a run. This is used to produce a new running route and a new pace.

2. Pace matching:

A user can choose to follow a route and pace with one of the recorded paces. When he does so, every time point he reaches a point in the route, his time would be recorded and matches against the time in the pace. In this case, the application would periodically inform the user through an audio notification whether he is ahead or behind the pace, and by how much.

3. Pace & Route publishing:

A user can choose to publish his runs publicly right after he finished his run. If the user's run creates a new route, it would be posted with his pace so that other users who choose this route can follow the same route. Otherwise, a new pace will be added to the followed route.

4. Run analysis on one or more runs:

After a run, a user can either choose to see its analysis on its own or in comparison with another pace. This pace can be one of the user's past paces, or from other users. When a user runs an analysis, he would be able to see the locations of both runners at every point in time. Only the fastest time by a user will be used as the comparison.

5. Save favourite routes

User can browse public routes and save the route as a favourite route for easy retrieval in the future

6. Track user statistics

User can access their overall run statistics from using the app to track their exercise regime.

Revised Specification

Our app will only be available for use on iPhones in vertical mode. It require users to provide the application with access to location services and internet to fetch location and routes information respectively. If location permission is not provided, the application will not be able to start a run/pace. If internet connection is not available, user will not be able to retrieve new routes; only cached ones. If there are cached information about previous runs of paces, the user will be able to view them but will not be able to make any changes or upload run statistics.

User Interface

We have revised our design of user interface and decided on the following views. There are 3 main views in the application, which are all be navigable through a single touch through the tab-bar at the bottom of the screen. Besides, there is a run analysis view which is included in multiple views.

1. Favourite Routes View

In this page, users will see a table with his/her bookmarked routes.

- A preview of the route is displayed, along with other useful information such as the start and end locations, distance and the number of runners who have previously run this route.
- The main use case of this page is to allow the user to bookmark and thus easily access their favourite routes with minimal effort.
- On selection of a route, the view will show the rendered route onto the map on the Activity View page.

2. Activity View

This view contains two functionality: Selecting a route and handling running activity.

In ‘selecting a route’ stage, a map and a start button are shown on the screen. The map shows markers which indicates the positions of different routes created by users. The start button lets the user start creating a new route while running. On this page, a user can either create a new route or follow other created route.

- When a marker is pressed, a pull-up controller will be displayed from the bottom edge of the screen, showing general statistics of the run.
    - Upon tapping the ‘follow’ button on the top edge of the pull-up controller, the user will be able to start following the first created run from the route shown
    - User can press the left and right button on the pull-up controller to view other routes represented by this marker.
    - User can choose to ‘swipe’ the pull-up controller down, in order to choose routes represented by other markers.
    - Upon expanding the pull-up controller, the user will be able to view the timings of the runs ran by other users.

- Upon starting a run (by pressing the start button), a new ‘route’ will be built by the user. The path taken by the user will be recorded, saved and uploaded to the cloud for other users to ‘follow’.
- Upon starting a run (by tapping pressing the follow button on the pull-up controller), a ‘follow’ action will be triggered by the user. The user will be able to get real time updates of his pace compared to the user that he is following.

Upon completion of the run, the user will be able to view the summary of this run, and view an interactive graph showing the statistics at every point of the run.

3. Profile

The user profile displays useful performance statistics based on the data collected from their runs, which gives them a good idea on their overall performance. Statistics include:

- Total distance clocked
- Average pace (min/km)
- Number of runs completed
- Average distance per run

This page also displays a list of the past runs of the user (sorted in order of the most recent).

Clicking on any of these runs will bring the user to a run analysis page. This page will also include the ‘Settings’ button for users to log out or change app settings & account details.

Reusable: Run Analysis

This page allows the user to analyze a particular run, showing a graphical display of certain statistics of the run during the progress of the run. There would be a Statistics include (represented as the y-axis):

- Run speed
- Elevation

Users can slide (pan) horizontally on the graph to see the exact x-y values at any point of the run. In addition to this, there is also a map which outlines the route of the run on it. Panning on the graph also visually displays the user’s location at that point in time on the map.

The user can also load another run (that follows the same route as the currently analyzed run) onto the map and graph, so that the user can make comparisons with their past runs (or with other people). Note that this run analysis can also be found after a user completes a run, in the summary page.

User guide

Logging in and out

- Logging in
    - Pace uses Facebook authentication to authenticate users. In order for a user to create routes or runs, or add routes to their favourites, they need to be logged in.
    - If the user is currently not logged into Pace, visit a view that requires logging in, such as Profile, or Favourite and tap ‘Log in With Facebook’. A Facebook pop-up should show up that allows the user to authenticate his Facebook credentials, and log him into Pace.
- Logging out
    - To log out, visit the profile tab, and press the “Log out” button.

Navigation & browsing

- Viewing nearby routes:
    - To find out about routes around a user’s current location, tap on the Activity tab. A map view would be presented and the user can pan the map around to explore more surroundings. The user should see a map of his current location’s surroundings.
    - The user can do a two-finger pinch to adjust the zoom level of the map to see more of the map.
- Viewing a specific route:
    - On the area map view, the user would see several indicators around the map, each containing an integer or an asterisk. This integer represents the number of routes available clustered around the area, and an asterisk is used if the number of routes exceeds 20.
    - To view the route, tap on one of these indicators, a drawer would appear from the bottom of the screen which would contain a summary of the route. This summary includes:
        - The total distance of the currently selected route
        - The average speed and time

        - A list of the top run for this route
        - An outline of the route would be drawn on the map
    - There would also be ‘LEFT’ and ‘RIGHT’ indicators on the drawer, which the user can tap to view the other routes in the cluster.

Running

- Starting a run (new route)
    - Click on the start (‘+’) button in the lower center of the map of the Activity page. Your running statistics will start being tracked. Any timings for the checkpoints would be relative to when this event is started.
- Starting a run (following another run)
    - Select a route from the map in the Activity page. To follow the route, press the “Follow” button when the drawer appears. When this button is pressed, the run is started similar to how a route is created.
- Muting the voice assistant
    - This application is provided with a voice assistant. To mute the assistant, simply tap on the mute button on the top right of the Activity screen.
- Locking the screen during the run
    - To lock the screen during a run, simply tap on the “Map unlocked” button to switch the mode to “Map locked”. This will result in the map camera tracking the user. The user will no longer be able to perform any gestures on the map, except for zooming out of the map.
- Ending a run
    - Click on the end button (a tick) in the lower center of the map of the Activity page. Your run statistics will stop being tracked and you will be show a summary of your run.

Post-Run

- Analyse run
    - Click on the statistics view in the summary page, which is the top half of the summary page, which contains the distance and speed of the run. A new view will appears that will be display a graph of your run.

    - The graph displayed depends on the user’s analysis settings - he can choose between altitude, speed and total duration. To choose between how you analyse this run, perform a long press on the graph (at least 1 second), and a menu would appear when the finger is lifted. Choose a mode from here.

- Save run
    - To save a run, click on the disk icon on the top right of the summary page.
        - If this run does not follow a route, the user will be prompted to create a new route. On clicking the button, a new route will be shared to the public.
        - If this run follows a route, the user can choose to either create a new route or add the run to the following route.
    - Note that you are required to trace the route taken by the route’s creator. Failure to do so will prevent you from adding your run statistics to the route’s leaderboard.
- Adding new run to route
    - Upon successful following of a route, you will be allowed to add your statistics to the route to compete in the leaderboard.

Other actions

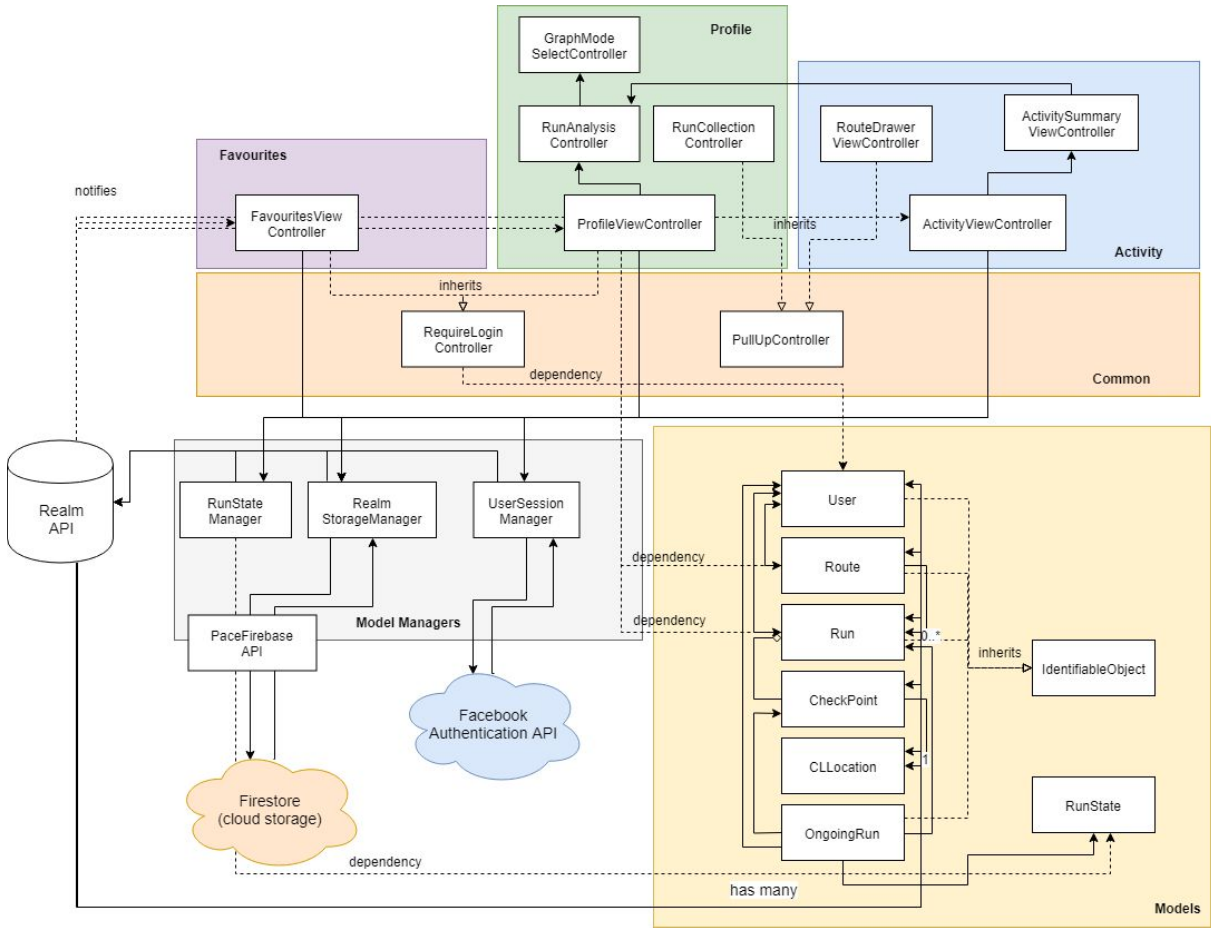
- Favoriting a route
    - Select a route on the map in the Activity page. To favourite a route, press on the “heart button” in the pop-up drawer. The route will be viewable in the Favourites tab.
- Accessing a favourite route
    - Tap on a route in the Favourites tab. You will be swapped to the Activity page with your selected route being rendered.
    - Note: You need to be logged in to access this page
- Viewing of cumulative statistics
    - To view all your past runs, go to the Profile tab. In that tab, you are provided with information on:
        1. Total distance ran while using Pace
        2. The average distance per run
        3. The total number of runs

        4. The average pace (in minute per kilometers).

- Viewing of past runs
    - To view all your past runs, go to the Profile tab. A list of runs will be shown. In there, you can select past runs to analyse the results. Simply tap on a run, and the run analysis graph will be shown. You can hold the graph to select the type of graph analysis to be shown. Options include altitude, speed and duration.
    - You can also compare your run with other runners who ran the same route as you. By dragging up the drawer, it reveals a list of other completed runs on that same route. By clicking, on one of the runs, you can view the graph of that run superimposed on the same graph view.
    - Note: You need to be logged in to view this page.

Design

Overview



Our application architecture was designed to be as modular as possible, which would allow us to easily conduct unit tests and make changes in the implementation. The application code can be split into the following components:

1. **Models** - Structures used to represent and store our data
    a. **User** - A user account created from their Facebook accounts via authentication
    b. **Route** - A running route (each route will have a run that it was created from). Additionally, a route can have multiple paces if other runners choose to follow that route.
    c. **Run** - A single run completed by a user (runner). Contains a list of **CheckPoints**.
    d. **CheckPoint** - A checkpoint that the runner has passed during the run, contains information such as the time, distance and location describing the progress of the run.

   e. **CLLocation** - Extended from Apple's in-build CoreLocation library, the representation of a location on the map. Contains information such as the longitude, latitude, speed, altitude and etc.
   f. **OngoingRun** - Similar data representation to a Run, but acts as its builder to build it the run as more CheckPoints are generated.
   g. **IdentifiableObject** - Superclass to inherit from to generate a unique identifier to distinguish Realm objects
   h. **RunState** - Used to persist the state of a run in the event that the app crashes. Stores an OngoingRun and the timings recorded by the stopwatch.
2. **Model Managers** - Used to handle persistent storage on both the local cache (Realm) and the cloud (Firebase)
   a. **PaceFirebaseAPI** - API interface that makes the Firebase queries to Firestore, and directly handles responses or errors from the query.
   b. **RealmStorageManager** - Interface used by view controllers to carry out updates directly to Realm or by passing callbacks to PaceFirebaseAPI (to call upon completion of request). Mainly handles requests pertaining to Runs and Routes
   c. **UserSessionManager** - Similar to RealmStorageManager but handles requests pertaining to Users. Additionally, interfaces with Facebook Authentication APIs which is later used to the user.
   d. **RunStateManager** - Used to persist the state of an OngoingRun in the event of an application crash. Called by AppDelegate to save the RunState in the termination callback.
3. **Favourites** - View Controllers (VC) / views pertaining to the Favourites page (tab)
   a. **FavouritesViewController** - main VC used for the Favourites tag
4. **Activity** - View Controllers / views pertaining to the Activities page (tab), that handles the runs.
   a. **ActivityViewController** - main VC used for the Activities tab
   b. **ActivitySummaryController** - VC used to display the summary and handle saving of a completed run.
   c. **RouteDrawerViewController** - auxiliary VC used to display the available routes at each route marker on the map.
5. **Profile** - View Controllers / views pertaining to the Profiles page (tab), that displays user information / statistics.
   a. **ProfileViewController** - main VC used for the Profiles tab
   b. **RunAnalysisController** - VC used to display an analysis of a completed run by the user.
   c. **RunCollectionController** - auxiliary VC used to display other Runs completed on the same Route being analysed.
   d. **GraphModeSelectController** - auxiliary VC used to select the graph mode displayed in the run analysis.
6. **Common** - Common modules used across the application, such as helpers, controllers, etc.
   a. **RequiresLoginController** - superclass VC for controllers that require user login in order to display their information: FavouritesViewController & ProfileViewController

b. **PullupController** - superclass VC for containers that can be dragged up from the bottom of the screen

Libraries and third-party modules

1. **Google Maps & Places** - for plotting a map so users can see their vicinity and route. This is also used for users to find routes around specified areas or places.
2. **Firebase** - for authenticating users and allowing them to record and share their runs and routes
3. **Realm** - as a local database

Authentication

On opening the application the first time, the user would be prompted to log-in to his Facebook account. The application will search Firebase (our own server) for the user's account, and login if found. If the user is a first time user of the application, the server will create an account for him/her on our server and log them in automatically.

The user may also choose to skip this authentication and continue using the app without an account (or logging in). In this case, the user would still be able to browse available routes as well as to follow the route/run of other runners. However, the user would not be able to save their completed run under the route (for others to follow) or to bookmark a particular route until they have logged in. Authentication would allow users to login to their account on other devices and access/modify any data related to that account.

Client-Server interactions

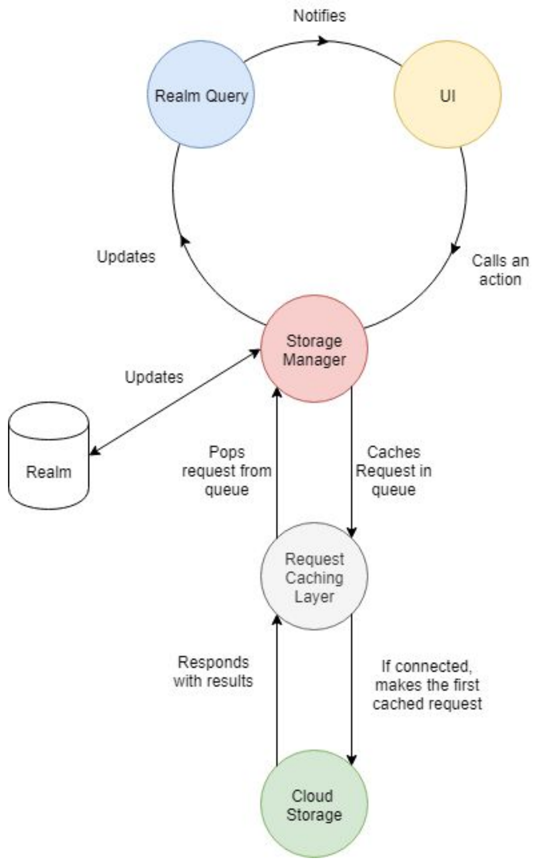
Goals

One of the features we want to achieve in our application is offline capability. If the user is unable to connect to our server to retrieve requested information, they will simply just fall back on their local cache for previously downloaded information and try to retrieve it there.

Implementation

In order to achieve this, we decided to set up a local database that would live on the client's device to store requested information for display.

However, when we first tried this we found that defining the caching mechanism and UI updating within the request callbacks made for unwieldy nested callbacks (For instance, imagine the view controller passing a callback to the storage manager, which would pass another callback, which is then passed to the storage API). So instead, we decided to use a reactive solution, which was enabled through using Realm. Realm provides a way to persist data and observe them from the UI. With this in place, the typical behaviour of our app looks like such:



As an example, a typical view works as follows:

1. The view controller first observes a Realm query for the objects it is meant to display. On viewDidLoad, this controller would first call the StorageManager for an initial fetch of the information.
2. StorageManager would then make a request to the cloud storage for the information, and on receiving it would store them into the Realm database.
3. Since the view controller is observing a Realm query for these objects, on addition of data to cloud, the view controllers would update views accordingly.
4. When the user or view controllers make subsequent requests, StorageManager handles them similarly, making the request for them and adding them to Realm.

There are several benefits to this way of structuring our code. Firstly, in this solution, there is a unidirectional flow of information, ensuring the code is kept clean and well separated. Secondly, with this solution, there would always be a single source of truth for each UI - the Realm query it observes to represent its view. Hence, the view can be simply written declaratively and it should be a true representation of the state at all times.

The typical alternative would be for the view to imperatively manage its own state. This may result in inaccurate representations of the state of the application due to race conditions in UI update callbacks. This also results in duplication of data which is another potential source of errors.

Offline actions & additional caching mechanisms

Goals

There are some other goals in designing and implementing a caching mechanism for our application. Firstly, we would like a fail-safe way to perform POST requests to the cloud - failed attempts to add or update objects in the cloud should be marked and reattempted at a later time. Moreover, these edits should happen in order to ensure correctness of transactions. Secondly, we would like to only fetch what we need - it does not make sense for the user to fetch items that he has fetched before.

As for the first issue, the operations would need to be stored in a queue of pending operations in the database before being attempted. This queue would then be dequeued one by one and the post requests are attempted again, one by one, until one fails. Once it fails again, the rest of the operations in the queue should not be attempted to ensure the order of the operations.

As for the second issue, it can be resolved by ensuring that all updated items are marked with an “last updated at” field, and anytime a successful query is made this field is updated. The next time the same query is made, we only need to fetch for items that happen after that time. This is a possible future extension to our current caching system, which despite lacking this extra measure of efficiency still performs correctly and responsively enough.

Implementation

This mechanism to cache POST requests is implemented with a special data structure known as UploadAttempts. They are created to capture these POST request attempts, and are stored in the persistent database in case failed attempts are not successfully reattempted before app crashes. UploadAttempts, in our implementation, store specific actions that are encoded into storable data in a fail-safe way. When retrieved, they would be decoded to produce idempotent actions that would be performed in their order of addition, in order to ensure that the correct end result is achieved in the cloud. This order is ensured by chaining these actions with their asynchronous callbacks, so an action would not be performed unless its preceding one is performed successfully.

These operations only refer to those made by the client but need to be sent to the cloud storage. As such, the user would see the changes happen locally on their clients, even as the attempts have failed and retries are ongoing. This should reassure users that their changes have been recorded. The idempotent nature of the actions ensure that duplicate actions are resolved easily on the cloud.

Pacing Algorithm

Issues Faced

Designing the models to be able to handle different possible use cases was challenging at the start. For instance, to be able to accurately match a user's pace with an existing pace in the route, they must both be following the same route. However, this is not always the case for the following user due to different reasons. As such, we needed to consider how we can best record runs, especially when a user chooses to follow a route. Initially, we thought about achieving this by recording a user's position at a regular time interval (e.g. 5 seconds) and recording these locations.

However, this posed to be an issue for someone following a pace. If the initial runner decided to stop at one location for a long period and multiple checkpoints are recorded for that location, a following runner would not be able to recognise which checkpoint to check its timing against. In addition, there might be issues when the route taken is up and down a single stretch of road, causing checkpoints to be in close proximity to one another. Both cases result in potentially noisy data that not only makes it hard for our algorithm to keep track of the runner's relative pace, but also a very messy route when drawn on the map.

Solution

As a result, the solution we arrived on was to first normalize a route that someone takes to a set of points in a route that are a significant distance apart. In other words, for one to record a new route, his route taken would be stored as a series of points in the route that are 20 metres from each other. With this, a following user would try to find a checkpoint that is not only the nearest but also with the closest cumulative distance. This would allow our matching algorithms to more deterministically filter and select the best candidate checkpoint to match a following runner's timing against.

The normalization approach keeps our data in a consistent format, which is important in using the data for other features such as run analysis and also generating run pacing stats. However, some edge cases appeared and we needed to make decisions on how to deal with them. For example, an edge case is that when normalizing a longer run based on a shorter run (in terms of distance). We would like to save the longer run in the consistent format (based on equal distance interval checkpoints) as the shorter run, but the longer run inevitably has some checkpoints which cannot be mapped in the baseline shorter run. Our solution to this issue is to normalize the remaining part of the longer run purely based on the distance interval and concatenate this part to the earlier part of the run which is normalized based on the shorter run.

Running Session and Run classification

As discussed in the section above, we decided to normalize the checkpoints in a Run based on fixed distance interval before it is saved. This section continues to discuss the management the running session when the user is in a Run and the new locations are made available from the GPS service.

There are two modes of running activity - new run and follow run. New run refers to an independently created run which is not intended to be saved under any existing route, while follow run refers to a run following an existing run. A follow run will be automatically considered to be saved under the route for the followed run, if it is eligible. (The eligibility criteria will be discussed later.) The mode of running is selected by user when he/she wants to start a run.

New Run

After a new run is started, checkpoints are created whenever the a new location is delivered from the GPS service. The checkpoints are not separate by fixed location, as when the location is available is not deterministic.

When the runner decides to end the run, the checkpoints will be normalized to checkpoints with fixed distance interval (i.e. 20m) in between them. After that, a new route is created for this new run.

Follow Run

(The run being followed is named as 'pace run' here)

Dealing with follow run is more complicated and challenging compared to new run. Since the runner now is following some other run, there are a few issues which need to be taken care of:

1. The follow run can be deviating from the pace run. In this case, the deviating status need to be detected, and the deviated run are not suitable to be saved under the route of pace run anymore.
2. The pacing statistics (how fast the runner is compared to the pace run runner) needs to be reflected to the user in real time. However, the existing runs have normalized checkpoints, but the follow run has checkpoints purely based on availability of GPS service.

After analyzing the issues above, we have decided to implement the following solutions to manage a follow run session:

1. Deviating status and run classification:
    a. Whenever a new location is delivered, there will be a check to determine whether there are not yet passed checkpoints (including the last passed checkpoint) in the pace run which is near the new location. If no pace run checkpoints are nearby, this

run is considered as deviated; if one such checkpoint can be found, this run is not considered as deviated for now.

   b. During the checking of nearby pace run checkpoints, if a checkpoint is near a new location, the checkpoint is marked as 'covered' by this follow run. At the end of the follow run, a certain percentage of checkpoints 'covered' is required for this follow run to be eligible for saving under the pace run route. Note that the percentage is calculated against the union of follow run checkpoints and pace run checkpoints. The current formula for calculating the percentage for eligibility is as follow:

$$\frac{(follow\ run\ points) \cap (pace\ run\ points)}{(follow\ run\ points) \cup (pace\ run\ points)}$$

2. Real-time pacing statistics:

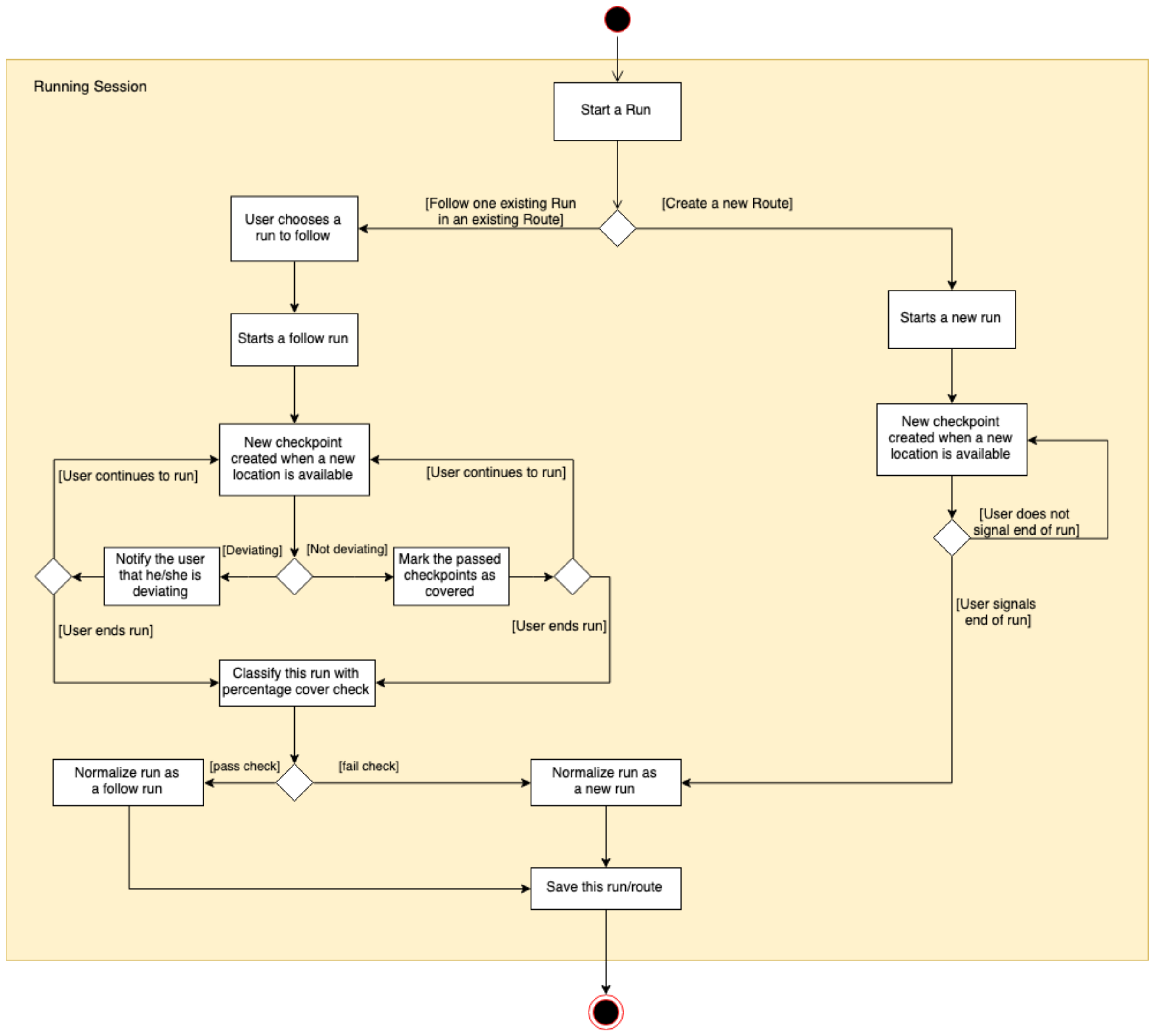
   a. It is not feasible to return the pacing stats immediately after the runner passes the pace run checkpoint due to the GPS location availability issue. The new location delivered most likely does not fall exactly on the checkpoint location. Therefore, the pacing stats are reflected to user in a fixed time interval.

   b. When it is time to reflect the pacing stats, the last passed pace run checkpoint and its successor is retrieved. These two checkpoints, together with the last obtained GPS location of the runner, are used to interpolate a temporary checkpoint of the pace run at the last obtained GPS location. The interpolated information in this temporary checkpoint is then used as pacing stats.

When a user decides to end a follow run, there will be a check on the percentage of checkpoints covered. If the percentage is at least 80%, this run is classified as a 'follow run' and is stored under the pace run route. Otherwise, this run is classified as a 'new run' and can only be saved as a new run under a new route. Normalization is subsequently performed to this run after it is classified, and it is then saved accordingly.

The management of running session is summarized in the diagram below:

(Pacing stats querying is not included, as the querying can be carried out any time during the follow run session.)


Running Session
Start a Run
[Follow one existing Run in an existing Route]
[Create a new Route]
User chooses a run to follow
Starts a follow run
Starts a new run
New checkpoint created when a new location is available
[User continues to run]
[User continues to run]
[Deviating]
[Not deviating]
Notify the user that he/she is deviating
Mark the passed checkpoints as covered
New checkpoint created when a new location is available
[User does not signal end of run]
[User signals end of run]
[User ends run]
[User ends run]
Classify this run with percentage cover check
[pass check]
[fail check]
Normalize run as a follow run
Normalize run as a new run
Save this run/route

Persistent States for Ongoing Runs

Issues

During the course of the run, it is possible for unexpected things to occur which could cause the termination of the application. As such, it is crucial to handle such situations so that data collected over the course of half the run does not get lost due to the application terminating.

Solution

To address this issue, we implemented a caching mechanism for the state of the ActivityViewController, which caches a RunState object into memory. This is mechanism is set up by setting the ActivityViewController as a PersistRunStateDelegate within AppDelegate. Upon termination of the application, AppDelegate receives a call to applicationWillTerminate, which then calls the PersistRunStateDelegate to generate a RunState from its current data. The RunState object will then be cached into Realm. Upon loading of the ActivityViewController again, the controller checks Realm for any existing RunState, and loads it into the controller which resumes the run.

Mocking Location for Testing

Due to the nature of a running app, our app requires changing the device location extensively for integration testing.

Issues Faced

It is not feasible to change the device location manually every time, and the Xcode simulator only allows to change the device location according to a route around the Apple Headquarter. However, our application requires the testing of running around different route at different locations, and also requires to simulate the running deviation. More importantly, we need to demo the application on a real device without physically moving, but the 'simulate run' functionality on the simulator cannot be used on a real iPhone device.

Solution

We implemented a custom *CLLocationManager* which implements a few methods which are used in requesting and updating locations in the *ActivityViewController*. These methods are mainly used to provide locations during a run so that the run can be recorded. A gpx file was exported from Strava and parsed into *CLLocation*s, which are subsequently being consumed by the custom *CLLocationManager* and used to update new fake locations as if the GPS service is being used.

Method swizzling was used to replace the built in *CLLocationManager* with our custom *CLLocationManager* (i.e. *MockCLLocationManager*). This is done through inject swizzling method invocation for specified class types during the runtime setup of the application. If a class conforms to the *SelfAware* protocol, its *awake()* method will be injected when setting up the *UIApplication*. By making *CLLocationManager* conform to *SelfAware* protocol and swapping some of its methods with the methods implemented in *MockCLLocationManager* in the *awake()* function call, we are able to override the behavior of *CLLocationManager*. Other classes can make use of *CLLocationManager* as if nothing has happened, while the implementation of the method used by other classes has been changed to our own.

The switching between using mock location and real location is through flipping the *MockLocationConfiguration.isMocking* boolean value. Switching mock and real location mode is only for development and testing purposes and should not be used in production.

The implementation of the location mocking used references from servel articles and blogs specified in the the header comments of the source code.

Known Issues

Since the location mocking is used for testing runs, it is only guaranteed to be working as expected under the *ActivityViewController*. Opening app to the activity page, switching to another page and switching back may cause the location service to behave in unexpected ways and may cause the app to crash. There can be other cases in which the app crashes because the location mocking is turned on and is not used in the activity page. With some investigation into this issue, a possible cause to the issue might be that the *CLLocationManagerDelegate* is not set properly in other VCs. However, we decided not to set them up because this will only be beneficial to the mocking but the mocking is not useful in those VCs.

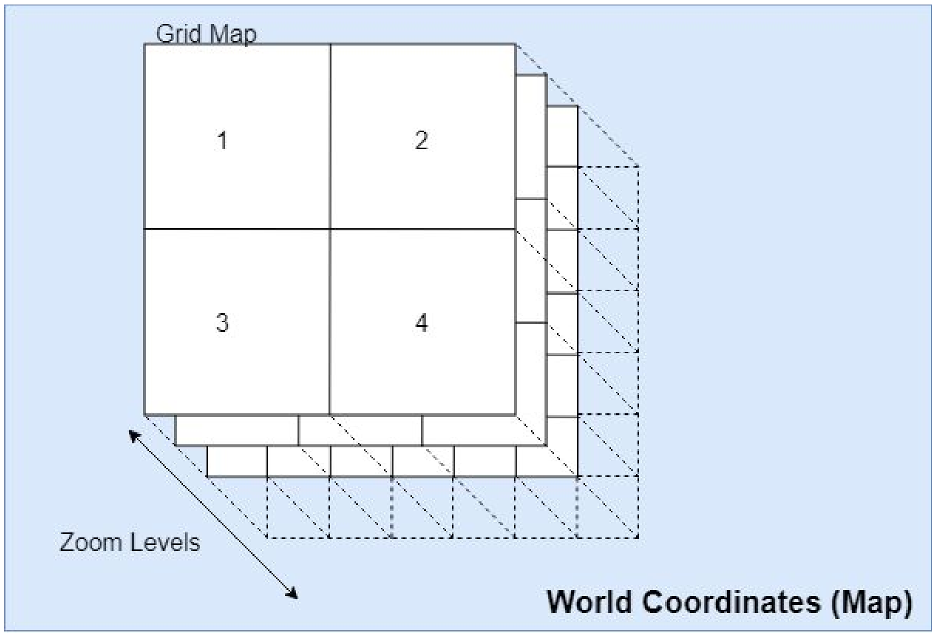
Another known issue is that under the location mocking mode, the blue dot indication the current position in the map does not move after starting a running session. This could be due to the way that the *MapView* was setup as well as the underlying implementation of the *MapView* which may make use of some other methods from *CLLocationManager* and those methods are not implemented by us. Since this issue does not affect the testing in any way, we decided not to investigate further.

Fixing the known issues are considered as a very-low priority task, as the current state already suffices its intended use cases, and the location mocking is not going to be used in production. Hence, feature-related tasks are prioritized over this task.

Browsing of Routes

For our application, we might reach a stage where there are a million routes created. As such, we cannot immediately fetch all ‘routes’ stored in the cloud as that will be extremely inefficient. Hence, we created a mechanism to only request relevant routes.

The general idea is to partition the world map into grids, and only fetch the ‘routes’ that the user pans the map to (i.e. we only render ‘routes’ that the user request).



Issues faced:

- We need to design our own grid system of the world map. As there is no grid system publicly available, we have to implement our own. Our grid system needs to be able to work for grids of different size for “layering” (see figure above). The grid system also needs to be able to retrieve the grids in a projected map given any tilt and rotation of the map. We circumvent this by drawing the bounds of the map (i.e. a rectangle that encompass the map) and determine all grids that intersect with that rectangle. We then populate all ‘routes’ in those grids. Difficulties faced by this includes getting the “*CLLocationDegree*” of a given width of a grid. As there isn’t any provided API to retrieve this, we have to implement our own. This is done by creating a *GridMap* object to handle the grid system. While alternative ways of retrieving “*CLLocationDegree*” such as using trigonometry, they require prior and precise knowledge of the Earth’s radius, which we do not have.

- Caching of markers: When we pan the map, we have to create markers to indicate routes. Creating objects are expensive, hence we want to create as little as possible. How we resolve this is by caching the markers that we generated, and reuse them when we are rendering a grid that was loaded before. This is more efficient for two reasons:

    - We only request the database for routes of a certain grid once. When we pan to a previously rendered grid, previously rendered routes will be cached and shown. It is more efficient in terms of time taken to load a marker and memory.

    - In the event that the user disconnects from the internet, the user will still be able to view previously rendered markers and interact with the markers accordingly.

- Rendering routes based on the map's zoom level: We need to collapse the markers when there are too many in the map's projection.
    - What is layering:
        - We have a different grid system for each "layer". With more layers, the collapsing of the routes will look better as each layer is better catered. However, this comes at a cost of memory and time taken to fetch the data from firebase. We hence design our application in a way that allows us to conveniently add "layers" to the application. Currently, we have three layers in our map of increasing grid size, but the layers can be increased and better tuned to collapse the markers better.
    - How it works:
        - Layering is the rendering of markers at different level. On higher levels of the map, we only render the number of counts in the grid, while at the lowest level, we render the marker representing the route. By having an increasing grid size as we zoom out, we can ensure that the map is not bloated with markers, which might hinder user experience or memory bloat.
    - For layering to work efficiently, we need to store the data to the database so that we don't have to load all routes the map's projection; we only need to query the database for the grid of that particular layer and show that value to the user. We currently store the gridID and the zoomLevel of the grid at that layer, together with the count of routes created in that grid. When user request for the number of routes at a certain layer, we will fetch this data from the database, and render the count as a marker on the map. Unfortunately, as our backend isn't able to perform calculations, we are not able to determine the centroid of the routes to accurate render the marker. Hence, we simply render the marker in the center of the grid. While this isn't as accurate, we had to make do with the limitations of Firebase.
- Collapsing of a route. If we render all route onto the projected map, there can be a case where there are too many markers on the map, causing the user to be unable to click a particular marker. Hence, we chose to collapse nearby routes into a single marker representing multiple routes. This result in some complexity as we will also have to collapse markers at every layer. The code is required to be very general to accommodate this.

Testing

Testing Strategy

Black-box testing

The black-box testing includes testing of high level behavior of the three main view controllers. This section can be regarded as a type of system testing, where the compliance of the product with the functional specification is tested.

The black-box testing is divided to three sections based on the main views: favorite page, activity page, and profile page. The navigation between these three main views are tested under each section, when there is a navigation button or a event triggering navigation in that main view. The functional specifications are thus thoroughly tested as it covers all the user views of the application and all the navigation options between them.

Glass-box testing

The glass-box testing includes unit testing of each module and integration testing for the application. We are using a bottom-up approach in this section. The unit testing is done first, and the integration testing is carried out when the individual components are ready and well-tested. Note that in our bottom-up approach, test drivers are not used for early integration testing, because the task of implementing view controllers and implementing models and reusable views are distributed to different people and are done parallelly. By the time the models and views are ready, the view controllers should be ready for testing as well based on our schedule. Therefore, building extra test drivers is not necessary and is not efficient given that we only have limited time to deliver the product.

As we are adopting MVC architecture, the unit testing is carried out of each individual models and reusable views. For the models, usual unit testing covering each execution path in the methods will be used. For the reusable views, snapshot testing can be used for each cells in the table view and the table view itself. The snapshot testing helps handle the edge cases in *UIView* instances and makes sure that the view stays in the correct place. We will likely be using ios-snapshot-test-case (https://github.com/uber/ios-snapshot-test-case/) here.

The integration testing will mainly be carried out based on each individual view controllers. Integration testing ensures that the view components display the correct information based on the current model state. The test cases in this section simulates user event such as button click, and checks whether the model state and displayed view are consistent with the event.

The description of glass-box test cases are not included in the appendix due to time constraints, and we feel that the other test cases are more important to serve as testing guidelines at the current stage.

Stress testing

Since we are using firebase as our backend, the stress testing focuses on testing the frontend workflow with extreme usage patterns. Some of these extreme usage cases can be simulated with automated unit test on a specific component, while others require manual testing. The manual testing includes manually switch between different views for many times and inspect many routes back and forth. The automated testing can be done on the vulnerable methods with extreme inputs.

The stressing testing test cases are not included in the appendix. Reasons being that the manual stress testing here is trivial, and the automated stress testing involves a subset of methods in the glass-box testing with extreme inputs.

Performance testing

The performance testing focuses on the aspect of the application where certain responsiveness is required and not easily achievable. This includes the real-time tasks handled in the app such as running status feedback when a user is following a route. Such feedbacks should be given to the user within a certain time period when the command is issued by the app.

Regression testing

The regression testing here is done on a selected group of critical interface methods tested with glass-box testing. To ensure that the core functionality of our application is not broken after making new changes, some important interface methods will be tested after any changes made to the codebase. Some examples of these methods are the normalization method for run and the method to compute current timing at checkpoints.

Testing Result

We have performed black-box acceptance testing, stress testing and performance testing according to the testing plans. Some bugs were discovered initially, such as locking of camera position when running etc.

However, due to time constraints, we were not able to carry out throughout glass-box unit testing on each components. Besides, we did not have a regression testing test suite setup because of the fast pace of development (the regression test cases could become outdated fast).

We have fixed the bugs that have been discovered, and are fairly confident that our product works as expected according to the user manual. However, bugs on edge cases in our implementation may remain, as we did not have time to perform unit testing on each individual component.

Reflection

Evaluation: What you regard as the successes and failures of the development: unresolved design problems, performance problems, etc. Identify which features of your design are the important ones. Point out design or implementation techniques that you are particularly proud of. Discuss what mistakes you made in your design, and the problems that they caused.

- The Pacing Algorithm is the core feature of our application that allows the user to track their relative pace against another runner, which is what our application is touted to be able to do. Considering the unpredictable nature of the data that is collected, we feel that our implementation handles this well, through sanitization of data as well as the classification of runs.
    - In addition to this, we have considered various edge cases and augmented our data with auxiliary fields (such as the cumulative distance) so that we can account for them.
- The browsing feature of our application is another important feature as it allows the user to easily search for a route to follow. This is only made possible (efficient) due to the underlying implementation of the grid system (layered grid maps).
    - Given no existing library or implementation to do this, this was designed by our team to efficiently fetch route counts within each grid, whilst keeping updating of our data structure efficient (constant in the number of layers).
- Offline capability is another core feature. Given that runs could bring users to areas without / poor reception, offline capability is definitely a good to have.
    - Similarly, our team designed the implementation to handle the caching of requests to keep the cloud in sync with the local device. We also took into account the order in which the cached requests should be handled by implementing an asynchronous queue.
- Persisting active states (that of an ongoing run) was not something we originally thought we had to do, but implemented towards the end. This feature is extremely important due to the fact that runs could be potentially long, and it is unpredictable termination of the application should be handled to ensure that the user can continue.

Lessons: What lessons you learned from the experience: how you might do it differently a second time round, and how the faults of the design and implementation may be corrected. Describe factors that caused problems such as missed milestones or to the known bugs and limitations. Known Bugs and Limitations In what ways does your implementation fall short of the specification? Be precise. Although you will lose points for bugs and missing features, you will receive partial credit for accurately identifying those errors, and the source of the problem.

- It would have been good if we had prior experience with the frameworks that we were using (Realm and Firebase). We initially had many problems with Firebase due to the limited capabilities it provided as a cloud server (in terms of querying). This slowed us down significantly at the start when designing our APIs.

    - Prior knowledge would also have allowed us to better design the modular structure of our application at the start.
- Due to the nature of our application, our dependence on location data and the sensitivity of said data, it would have been good for us to start out by collecting existing routes (in the form of gpx files) and add in the mocking capabilities from the start. This would have facilitated the debugging / testing of our methods.

Appendix

Test Cases

Black-box testing

Test ‘favourite’ page:

- Test login:
    - If user is not logged in
        - There is a button for user to login.
    - If user is logged in
        - Should be able to view his/her favourite routes.
- Test viewing of routes:
    - For each route displayed on the screen:
        - The route was marked as ‘favourite’ before.
    - Scrolling down while there are still routes not displayed before:
        - Should display more routes marked as ‘favourite’.
- Test navigation:
    - Click on one of the routes:
        - Should jump to the ‘activity’ page and display this route on the map
    - Click on any button on the bottom nav bar:
        - Should redirect to the corresponding page

Test ‘activity’ page:

- Test starting run:
    - When this page is first open:
        - The map on this page should be adjusted such that the current location is the center of the map
        - The WiFi symbol should show the current WiFi strength
        - The GPS symbol should show the current GPS strength
    - Click on the start button:
        - A new run should be started
        - The time stats should start updating
        - The other stats should start updating once a location is delivered by the GPS service
        - Click on the ‘Map unlocked’ button at the top right of the map should make the camera position follow the user location. (Click on ‘Map locked’ to toggle back this behaviour.)
    - Click on one of the markers shown on the map:
        - If the zoom level is not at maximum:

            - Should zoom in more to the position of the marker
        - If the zoom level is at maximum:
            - A draw should appear and display one of the route represented by the marker clicked
                - Information of the route should be displayed on the drawer
            - If the number on the marker is not ‘1’
                - There should be clickable left/right button at the top edge of the drawer
                - Clicking on the left/right buttons should be able to navigate between the different routes represented by this marker
            - Click on the ‘heart’ button should add/remove this route from the user’s list of favourite routes
            - Click on the follow button
                - If user’s current location is not near the starting point of this route
                    - An alert will be displayed, notifying the user that he/she is too far away from the starting point to start following.
                - If user’s current location is near the starting point of this route
                    - A follow run should be started, the pace run is the creator run of the current route
                    - The time stats should start updating
                    - The other stats should start updating once a location is delivered by the GPS service
                    - If the sound is turned on according to the button at top right corner of the screen
                        - The pacing info should be periodically reported to user through sound
                    - Click on the ‘Map unlocked’ button at the top right of the map should make the camera position follow the user location. (Click on ‘Map locked’ to toggle back this behaviour.)

- Test end run:
    - Click on the round ‘tick’ button at the bottom of the map
        - Should complete the run and redirect to run summary page
        - Should display the stats for the entire run
        - Click on the stats region
            - Should display the analysis page
            - Pan left and right on the analysis graph should show the info at that position, and the location symbol on the map above should move accordingly

      - Long press on the analysis graph should show an menu to change analysis mode. The three modes available are: Speed, Altitude and Duration.
      - Click on ‘back’ button at top left of the screen should go back to run summary page
    - Click on the ‘disk’ icon at top right of the screen
      - If user is not logged in
        - An alert should appear to ask user to log in
      - If user is logged in
        - If the distance ran was less than 20 meters
          - An alert should appear to tell user that the distance covered is insufficient to be saved
        - If the distance ran was more than 20 meters
          - If this is a new run
            - An alert should appear to ask user to confirm creating a new route for this new run
            - Click on ‘Create new route’ should save this run to a new route
          - If this is a follow run
            - If the run passes the percentage check to be considered as another run in the same route
              - An alert should appear to ask user whether to add this run to route or create a new route
              - Click on the corresponding button should save the run accordingly
            - If the run fails the percentage check
              - An alert should appear to inform user that this run cannot be added to the following route, and to provide a ‘Create new route’ button
              - Click on ‘Create new route’ button should save this run to a new route
    - Click on the ‘back’ button at top left of the screen should go back to activity page
- Test navigation:
  - Click on any button on the bottom nav bar:
    - Should redirect to the corresponding page

Test ‘profile’ page:

- If the user is not logged in:

    - There would not be anything visible on the Profile page, except for a Facebook Log-in button.
    - When the login button is pressed, there should be a Facebook login page that appears that lets the user authenticate their Facebook account. On pressing this button, the user would be logged in and the profile page for the user would be generated.
- Log-in
    - When the user logs in with the user profile page, within 1 second the profile page should be populated with the user's profile information.
- View the stats of the user
    - This would be only available when the user is logged in.
    - When the user opens the profile page, the top half should display the cumulative statistics of the user
        - The center top should display the cumulative distance run by the current user
        - The second row left should show the average distance run by this user
        - The second row center should show the total number of runs run by this user
        - The second row right should show the average pace of the user
    - These statistics should be verifiable by calculating these statistics for all of the user's past runs.
- Test log out:
    - This should only be available if the user is logged in.
    - When the user is logged in, there would be a button labelled 'Log Out' on the top right of the profile page
        - Click on the 'logout' button:
            - The user should be logged out and the profile page would revert to the default logged out, empty page.
        - If the same user attempts to log back in, the information from the last log in should be presented before a refresh happens.

Performance testing

Test fetch data from cloud:

- Navigate to 'Favourites' page:
    - If the user is logged in, his favourite routes should be displayed within a reasonable time provided that network connectivity is available.
    - If network connectivity is not available, the data should be fetched as soon as it is available.
- Navigate to 'Activities' page:
    - If there is network connectivity, the numbers should appear as soon as the data is fetched.
    - The user should be able to pan and zoom the map at any point as long as he is viewing the activity screen.

    - No data should be visible, until the map is stationary (user stops panning and zooming, releases finger from the map), after which the data would then be loaded.

Test storing data to server:

- When a user attempts to save a run, or add or remove a favourite:
    - If the user is connected to the internet:
        - Should display success message and redirect back to the original page after a few seconds, depending on the connectivity.
    - If the user is disconnected from the internet:
        - The user should only be able to see that these changes have been made on the client's database, i.e. the changes should be visible.
        - The requests should be cached, and when connection is restored and another upload is attempted, the requests should be sent

Test storing background session:

- If the user is currently in the middle of a run, if the app is accidentally shut down or interrupted (e.g. with a phone call), the timing and distances should be correctly reflected on app reopen, based on the last successful save. The time reflected should still be amount of time passed since the run has started.

Test running feedback:

- When a user is in a follow run session:
    - If the sound is not turned off:
        - There should be sound feedback on the pacing information (e.g. how many seconds the current user is behind or faster than the pace runner) for every 5-6 seconds (the update interval is currently set to 5s, and 1s latency is allowed here).