EdinFit - A Public Transport Application That Keeps You Fit

Manas Bajaj

4th Year Project Report
Software Engineering
School of Informatics
University of Edinburgh

2016

Abstract

In the last few decades, the proportion of people who perform some form of physical exercise on a regular basis has been declining steadily throughout the world. The major factor behind this decline is that nowadays people have a more stressful work life due to longer working hours and a faster work pace, and therefore they are not able to fit exercising into their daily routine. An easy solution to this problem is to simply make physical exercise a part of the journey to work. This dissertation describes the complete development life cycle of a mobile application that assists people in doing so.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Manas Bajaj*)

Table of Contents

Chapter 1

Introduction

The goal of this project was to make use of the newly-released Open Data API [1] made available by Transport for Edinburgh, to create a public transport mobile application that takes users from one place to another, while keeping them fit.

1.1 Motivation

A study [2] suggests that people who commute to their work place via public transport have a lower body mass index (BMI) and body fat percentage than people who only commute via cars, implying that even the incidental physical activity involved in public transport journeys plays a crucial role in keeping a person fit. Another study [3] suggests that performing a non-vigorous physical activity like walking for just 30 minutes a day can increase cardiovascular fitness, strengthen bones, and increase muscle strength and endurance. On top of these health benefits, walking is also the simplest form of exercise, that requires minimal equipment, and can be done at any time of the day. However, the proportion of people that walk regularly has been declining steadily throughout the world in the last few decades [3]. The major factor behind this decline is that nowadays people have a more stressful work life due to longer working hours and a faster work pace, and therefore they are not able to fit exercising into their daily routine. An easy solution to this problem is to simply make physical exercise a part of the journey to work. For example, people could walk to the next bus stop rather than just stand waiting for their bus, or perhaps they could leave the bus one or two stops earlier and walk to their destination from there. Doing something as simple as this can significantly improve their overall health. However, no public transport application available today is designed to assist people in doing so, as they only seem to focus on getting the user to their destination in the fastest way possible. Therefore, I wanted to create a mobile application that would not only take its users from one place to another, but at the same time, help them stay fit.

1.2 Achievements

I was able to develop and publish a fully functional public transport mobile application for Android [4] devices that had the following key features:

- **Basic transport information:** Show live departure times for stops and up-to-date weekly timetables for all bus services served by these stops. Users can also view bus stops near their current location and can also add them to a favourites list for quick navigation. Detailed walking directions along with accurate duration and distance estimates to reach all bus stops are displayed. Most of the transport information is available even when the user is offline.
- **Wait-or-walk:** Accurately estimate whether a user who is waiting for a bus, has enough time to walk to one of the next stops and still have a high probability of catching the bus.
- **Journey planner:** Come up with the shortest possible route to the destination, taking into account the physical activity requirements input by the user.
- **Physical activity statistics:** Provide detailed activity statistics for journeys undertaken by users. This includes statistics like the number of steps taken, number of calories burned, total distance travelled and the average speed during the activity. The entire path travelled by the user is also displayed on the map.
- **Live bus tracking:** View live bus locations throughput the different views in the application.
- **Disruption notifications:** Alert the users by sending them notifications regarding planned and incidental bus service disruptions.
- **Search:** Allow users to quickly search for bus stops and services.

The application was designed keeping numerous non-functional requirements in mind. These are listed in Section 3.3.

1.3 Related Work

As mentioned in Section 1.1, at the time I started working on this project, there were no public transport applications available on any of the mobile application stores that tried to combine transportation with fitness. However, since the application that I was developing had to have the basic feature-set of a public transport application before fitness related features could be incorporated, I decided to review the two most popular public transport Android applications (with at least 100,000 downloads) for the city of Edinburgh: Edinburgh Bus Tracker

[5] and Transport for Edinburgh [6]. Both applications were evaluated using the following criteria:

- **Functionality:** Does the application provide the basic features that a public transport application must have?
- **Usability:** Does the user interface of the application conform to the design guidelines provided by Google? Does the user interface make use of existing design patterns so that new users can quickly and easily learn how to use the application?
- **Performance:** Is the application fast at performing the various advertised tasks? Is the application responsive at all times?
- **Availability:** To what extent does the application function when the user is offline?
- **Compatibility:** Does the application support a large range of devices with varying operating system versions and screen sizes?

The reviews are contained in the following subsections.

1.3.1 Edinburgh Bus Tracker

Screenshots are shown in Figure A.1.

Functionality

This application had the following key features:

- *Display all bus stops near the user's current location on a map.* The application was not able to detect my location. Although I kept getting a pop-up message saying "Fetching your location", my location was never actually displayed on the map. The map did display the locations of all bus stops correctly, along with the names of the services served by them and the direction in which the services would go after departure.
- *Bookmark bus stops for quick navigation.* I was able to bookmark stops and easily navigate to them via the "bookmarks" view. This feature would definitely save time if someone wants to urgently view the next departure for a certain stop.
- *Display all routes of the services served by a stop.* Clicking on a bus stop on the map allowed me to view the routes of all the services served by that stop. This feature seemed very useless to me as no one would ever want to view the routes of all services on a single map at the same time. It would've been better to allow the user to select which service they want to view the route of. Also, in order to distinguish between the different services on the map, the application just labelled each stop in a service's route with the name of the service. It would've been a lot more clear if different coloured

markers were used for each service. Lastly, the functionality to switch between inbound and outbound routes was also missing.

- *Allow the user to get a notification when they are near a certain bus stop.* This feature worked flawlessly. This would be extremely useful to someone who doesn't want to keep checking the departure times, but just wants to get notified by the application when a certain service is due.
- *View upcoming departure times for a bus stop.* I was able to view the upcoming departure times for bus stops by simply tapping on the required stop on the map. However, only the next 8 departure times were available, so the only way to know the departure times after the provided times was to just wait and reopen the application at a later point in time. Also, once the upcoming departure times were loaded, they were not updated live. So I had to navigate back to the home screen and then come back just to see the updated times.

Apart from the missing functionality mentioned above, the other essential functionality that was missing from the application was as follows:

- View live bus locations.
- View weekly timetables for bus stops.
- Journey planning.
- View service disruptions.

Usability

The user interface of the application was extremely outdated as it was making use of Android version 2.2 (Froyo) [7] API which was released in April 2010. Whereas, the Android operating system had a complete UI makeover with version 5.0 (Lollipop) [8], which was released in 2014. Apart from this, the application made use of an integrated browser window to display the map, instead of using the native Google Maps Android API. Due to this, all interactions with the map were quite laggy and unresponsive. Lastly, I noticed considerable stuttering when I tried to scroll the departure times list. This could be because the developer of the application was not aware of the view recycling performance improvement technique for the Android `ListView` component [9].

Performance

Besides the unresponsiveness of the user interface mentioned above, the application seemed to perform really well in terms of loading transport related data. Although, the application did not seem to be using a local database to cache HTTP responses, and therefore whenever I tried to go back to view the same information, it had to be downloaded again.

Availability

As mentioned above, there was no local database being used and therefore the application stopped functioning when I went into offline mode.

Compatibility

This application supported all Android operating system versions from 1.5 and above. According to the latest report on Android fragmentation [10], this meant that the application could be installed on 100% of all existing Android devices.

1.3.2 Transport for Edinburgh

Screenshots are shown in Figure A.2.

Functionality

This application had the following key features:

- *View live departure times and weekly timetables for every bus stop.* I had no problems in using this functionality. All departure times were clearly displayed, and the live departure times were also automatically updated every minute.
- *View nearby bus stops on a map.* I had no problems in using this functionality. My location was correctly identified and all nearby bus stops were displayed accurately. However, it would have been really useful if the application allowed me to view nearby stops for a location different than my actual location.
- *Discover the best route to anywhere in Edinburgh - search for a place or address to compare transport options and see detailed directions all the way there.* The problem here was that the application only allowed me select bus stops as my destination location. Instead, it should have allowed me to point on a map to select the exact location where I wanted to go. Also, the suggested journey/itinerary did not list all the stops in the bus' route, but only the stops where I was supposed to get on and off the bus. It would have been useful to also include the intermediate stops so that the user could know how far are they from their final stop.
- *Bookmark bus stops for quick navigation.* This functionality worked flawlessly as I was able to quickly navigate to my bookmarked stops directly from the home screen.
- *View service routes and live bus locations.* This functionality also worked flawlessly as I was able to view full routes of specific services (unlike Edinburgh Bus Tracker, which showed all routes at once), along with live locations of buses belonging to the selected service. Although it would have

been useful if it provided me with an option to switch between inbound and outbound routes as for some services these routes can be different.

- *View list of service disruptions.* This functionality also worked flawlessly as I was able to view a list of all service disruptions, along with a summary of the cause of disruption and a list of all services affected by it. Although it would have been useful if the application automatically notified the user of any disruptions in the services served by their bookmarked stops, rather than making the user manually scan through the entire list of disruptions.

Usability

The application did attempt to follow the latest Material Design guidelines [11] provided by Google, however did not fully conform to them. For example, the application makes use of dashboard navigation pattern (all main features displayed as separate buttons on the main screen), whereas Material Design guidelines suggest the use of navigation drawer pattern [12] for content navigation. Also, Material Design guidelines suggest that "colour should be unexpected and vibrant" and provide a list of colours that should be used to style the application, whereas this application makes use of very dull colours. Overall, the user interface seemed to be quite intuitive and the transitions between the different views were seamless.

Performance

The application performed really well in terms of loading times (loading of transport related data), however, it did not seem to be using a local database to cache HTTP responses, and therefore whenever I tried to go back to view the same information, it had to be downloaded again.

Availability

As mentioned above, there was no local database being used and therefore the application stopped functioning when I went into offline mode.

Compatibility

This application supported all Android operating system versions from 4.1 and above. According to the latest report on Android fragmentation [10], this meant that the application could be installed on 91% of the devices. This is in fact considered to be better than supporting really old devices since the application can make use of the latest APIs which are backwards incompatible.

1.4 Report Structure

This report is structured as follows:

- In Chapter 2, I describe the factors I considered while choosing a development methodology for my project.
- In Chapter 3, I describe how I elicited requirements from the stakeholders that had been identified. Then based on the development methodology chosen in Chapter 2, I break down the requirements into multiple increments and create a project timeline.
- In Chapter 4, I describe the major technology choices that I made, give a brief introduction to the development frameworks I used, and then describe the design and implementation process for each increment identified in Chapter 3.
- In Chapter 5, I describe the different kinds of tests that were carried out before the delivery of each increment identified in Chapter 3.
- In Chapter 6, I describe how the project was evaluated and discuss the results of evaluation.
- In Chapter 7, I summarize what I was able to achieve with my project and discuss future work.

Chapter 2

Development Methodology

A software development methodology refers to the framework that is used to structure, plan, and control the process of developing a software system [13]. Essentially, the development work is split into distinct, ordered phases with the intent of better planning and management. A myriad of such frameworks are currently available, and since a single methodology is not suitable for all projects, each available framework has been designed for specific kinds of projects. The most commonly used frameworks include Waterfall, Spiral and V. Choosing the right development framework is extremely critical to the success of a project, since the way the entire development process is carried out will depend on the selected framework.

I started out by considering Waterfall model [14] as the development methodology for this project. In a waterfall model, the project is divided into a sequence of phases, and each phases must be completed fully before going on to the next phase. The general overview of this model can be seen in Figure 2.1:

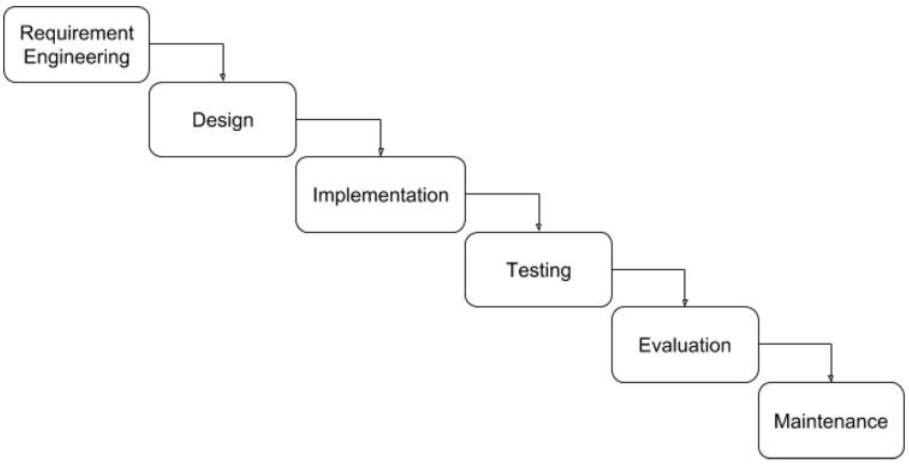

Figure 2.1: Overview of the Waterfall model

Waterfall model is very simple to use and understand, and is suitable for small projects where requirements are well-defined. Since this project's main objectives

and requirements were already defined as part of the proposal process (other requirements were collected during the requirement engineering phase - see Chapter 3), and the requirements were at a very low risk of changing, the Waterfall model seemed to be the clear choice of development methodology. However, the main issue with the Waterfall model is that working software is only produced near the end of the development life cycle, and therefore if the stakeholders do not approve of certain functionality later on, it can be quite costly to go back and make changes to the software. So I started to look for a different methodology what would allow me to generate working software quickly and early during the development life cycle. Then I came across a methodology called Incremental model [15], which essentially applies the Waterfall model incrementally. That is, the project is divided into multiple components based on the predefined overall requirements of the system, and then each component is developed and released using the Waterfall model. The series of releases that takes place is referred to as "increments", where each increment provides additional functionality to the end-user. In contrast to the Waterfall model, this model ensures that working software (builds) is generated early on in the development life cycle, and therefore the stakeholders can review each build and propose changes which can then be easily made as the developer would only have to deal with the code for one specific increment. This model also simplifies and expedites the testing process as it is easier to test and debug a single increment compared to the entire software system at once.

Here is an overview of the Incremental model, which I ended up using:

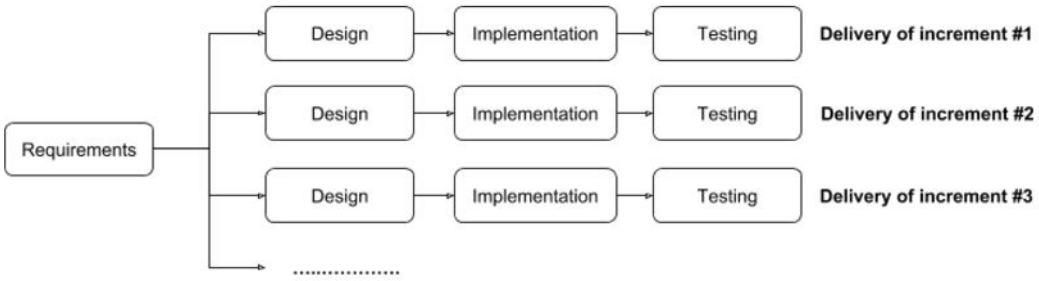

Figure 2.2: Overview of the Incremental model

As seen in Figure 2.2, the overall requirements were first collected during the requirement engineering phase and then divided into multiple increments (see Chapter 3 - Requirement Engineering and Planning).

Chapter 3

Requirement Engineering and Planning

In the context of software engineering, requirement engineering is a process of eliciting software requirements from the client, analysing them and then documenting them. It is divided into four stages:

1. **Stakeholder identification:** The developer identifies the people or organisations that have legitimate interest in the software being developed.

2. **Eliciting requirements:** The developer gathers requirements from the stakeholders using a myriad of different techniques such as questionnaires, surveys, interviews, etc.

3. **Analysing requirements:** The developer arranges the collected requirements in order of importance and development convenience.

4. **Documenting requirements:** The developer formally records the requirements for use as a reference point in later phases of development life cycle.

Requirement engineering is extremely critical to the success of a software project since requirements aid in the specification of project goals and in the planning of development cycles. It also helps in simplifying the evaluation phase of development life cycle as stakeholders would be less likely to come up with completely new requirements later on if the software they end up using was developed keeping all their expectations in mind.

Requirement engineering was performed and the resulting set of requirements was then divided into multiple increments. The results are detailed in the sections below:

3.1 Stakeholder Identification

There are four categories of stakeholders in any computer system: [16]

1. Those who are responsible for design and development.
2. Those with a financial interest, responsible for the system's sale or purchase.
3. Those who are responsible for the system's introduction and maintenance.
4. Those who have an interest in using the system.

Since I was the only person responsible for the design, development, testing, introduction and maintenance of the project, and that the final application was to be published for free, the first three categories pointed to me being the stakeholder. Then the only remaining category (4) identified the stakeholder to be the individuals interested in using the application, that is, the end users. Note that my supervisor, Professor Jane Hillston, played an integral role in evaluating each of the increments and providing me with detailed feedback based on them, and was therefore also considered to be a stakeholder.

3.2 Eliciting Requirements

Now that the stakeholders had been identified, it was time to gather requirements from them. I decided to use user stories as the requirement elicitation technique. A user story is basically a brief scenario-based description of a feature written by the future user of the system. My decision was based on the fact that they are very brief, quick to write and change, and relatively easy to estimate and prioritize compared to requirements gathered using other techniques such as surveys, questionnaires and interviews.

Here are the user stories I was able to gather from fellow students who used buses on a daily basis:

- *"I should be able to view stops near my location, along with the time it would take me to walk to them."*
- *"I would like the app to display upcoming departure times for all stops near me. It would be great if the entire week's timetable is also available."*
- *"I want to be able to view the live locations of buses belonging to a certain service. These should be displayed on a map and should be updated whenever the bus moves."*
- *"Since this app encourages fitness, it should show the number of calories I burned walking to the bus stop. It should also allow me to set my daily gaols in terms of the number calories burned."*
- *"I would like to add particular bus stops to my favourites list, so that I can quickly navigate to them and see when the next bus departs. Also, as*

I change my phones quite frequently, it would be great if my favourites list remains intact on every device I use."

- *"I want to be notified whenever there is a diversion in the regular route of a bus or when there are delays due to traffic jams."*

I, being one of the stakeholders and the project proposer, also added a few user stories myself. They are listed below:

- *"The application should have the ability to estimate whether a user who is waiting for a bus, has enough time to walk to the next stop and still have a high probability of catching the bus. Walking directions must be displayed to allow the user to successfully navigate to the next stop."*
- *"The application should be able to come up with the shortest possible route to the destination, taking into account the physical activity requirements input by the user. These requirements will include walk duration, total distance covered, average speed, step and calorie counts. Walking directions must be displayed to allow the user to successfully navigate to their destination."*
- *"The application should have the ability to display routes of a certain service on a map, indicating all the stops on the route. The application's basic functionality (displaying departure times) should work even when the user is offline."*

3.3 Analysing and Recording Requirements

As none of the stories were conflicting or redundant, I went on and decomposed the user stories into functional and non-functional requirements. Functional requirements describe what the system should do, while non-functional requirements place constraints on how the system will do so. I also indicated an estimate for the effort to meet each functional requirement, along with the level of priority. Instead of working out exactly how big each story is on its own, I decided to assign points (on a scale of one to five with five indicating the most amount of effort) to each of them as a relative indication of how long it will take me to develop the features implied by the functional requirement. This method of estimation ensured that I was able to accurately predict the implementation time for a certain feature based on the features implemented in the past. For prioritisation, I basically arranged the requirements in order of importance, with the most important requirement being at the top of the list. This helped me maintain a priority queue of tasks in the implementation phase, ensuring that within each iteration, I chose the task with the highest priority, and therefore implemented the most important features first. I also grouped the functional requirements into multiple increments (see Figure 2.2), with the first increment having enough functionality that the end-users can already start using the application.

Here is a list of functional requirements implied by the collected user stories:

Note that in the list below, the sequence number indicates the priority and the emboldened number of points in square brackets indicates the estimation of effort.

1. [**Increment #1**] Each user should be able to have their own account setup within the application. This will store information such as their favourite bus stops and physical activity statistics. This will also enable the users to restore their existing data if they decide to switch devices or use multiple devices and want to synchronize their data across all devices. [**2 pts**]

2. [**Increment #1**] Users should be able to view bus stops near them, along with the walking duration to each stop. [**4 pts**]

3. [**Increment #2**] Users should be able to view live departure times for stops and up-to-date weekly timetables for all services served by these stops. [**3 pts**]

4. [**Increment #3**] Users should be able to view detailed route information for each service. [**2 pts**]

5. [**Increment #3**] Users should be able to view real time information on the position of buses. [**1 pt**]

6. [**Increment #4**] Users should be able to add bus stops to their favourites list, and then be able to navigate to their favourites list which shows live departure times for each favourite bus stop. [**3 pts**]

7. [**Increment #5**] Users should be able to use the application in wait-or-walk mode, which estimates whether a user who is waiting for a bus, has enough time to walk to the next stop and still have a high probability of catching the bus. Walking directions must be displayed to allow the user to successfully navigate to the next stop. [**5 pts**]

8. [**Increment #6**] Users should be able to plan journeys from one location to another. The journey planner should come up with the shortest possible route to the destination, taking into account the physical activity requirements input by the user. These requirements can include walk duration, total distance covered, average speed, step and calorie counts. Walking directions must also be displayed to allow the user to successfully navigate to their destination. [**5 pts**]

9. [**Increment #7**] Users should be able to view physical activity statistics of journeys they previously undertook using the application. [**2 pts**]

10. [**Increment #8**] Users should be automatically notified whenever there is a disruption in any of the bus routes. [**2 pts**]

The above functional requirements will be further decomposed into a sequence of steps that need to be performed in order to implement the different components needed to meet the requirements in the design phase of development life cycle (see Chapter 4).

Here is a list of non-functional requirements that can be derived from the collected user stories:

- **Availability:** The basic functionality of the application must work even when the user is offline. This is extremely critical as it is not always possible for the user to be connected to the Internet when on the move.
- **Security:** All user data stored on the client-side or server-side must require authentication to access. Any sensitive information passed between the client and the server must be encrypted before it is sent.
- **Compatibility:** Since mobile devices come in a lot of different shapes and sizes, with vastly different performance levels and running different versions of the operation system, it must be ensured that the application supports a large range of devices, and therefore be able to reach a bigger audience.
- **Usability:** The application must provide a user experience that conforms to the guidelines provided for the target platform. Following the guidelines will ensure that new users are instantly familiar with the design patterns, making it easier for them to learn how to use the application.
- **Efficiency:** Tasks running in the background, such as tracking the user's activity or constantly polling the server for disruption notifications, must be optimized for energy consumption, ensuring that the device's battery lasts longer.
- **Performance:** Complicated computations such as journey planning and wait-or-walk decisions must be optimized for response time. Such long-running tasks must also be performed in the background so that they do not interrupt the operating system thread responsible for drawing and laying out user-interface elements, and therefore making it lag.
- **Scalability:** The server should be capable of handling high workload situations, for example, a large number of users trying to send requests to the server at the same time.
- **Documentation:** Due to a lot of features present in the application, it might be necessary to include tutorials/user documentation to make the user aware of all the functionality.

3.4 Planning

In order to estimate the completion time for the project, I considered the effort level estimates for each requirement and the workload of other university courses that I was undertaking, and grouped the increments into months to come up with the following timeline:

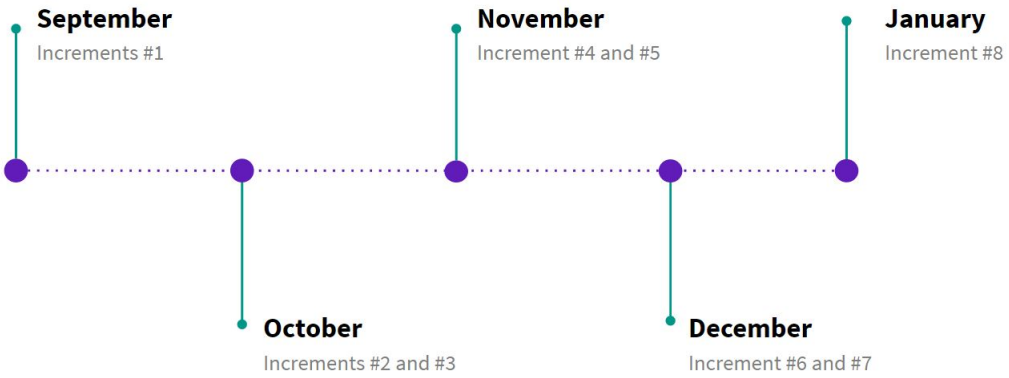

Figure 3.1: Project timeline

Chapter 4

Design and Implementation

Now that the stakeholders' requirements had been determined and documented, and I had a good understanding of them, I moved on to the design phase of the development life cycle. In this phase, the overall system architecture was defined in order to meet the functional and non-functional requirements of the system (see Section 3.3). This included details like what the final system would look like, how it would function and how the different components would interact with each other. The detailed system design specification that was output from this phase acted as an input to the next phase - the implementation phase. The implementation phase is also known as programming phase since this is where the work is divided into smaller units and the actual code is then written. This is the longest phase of the development life cycle.

As shown in Figure 2.2, the requirements were divided into multiple increments and then design, implementation and testing were performed on each increment separately. The closely related phases, design and implementation, are described in this chapter, while the testing phase will be described in Chapter 5.

Before delving into the details of each increment, in Section 4.1, I will describe the major technology choices that were made, give a brief introduction to the development frameworks used, and give a glimpse of how the code was structured within each framework. In Section 4.2, I will describe the design and implementation process for each increment. Each subsection of Section 4.2 corresponds to a different increment, and within each increment the functional requirements are converted into a sequence of steps (when possible) that need to be performed in order to implement the different components needed to meet the requirements.

4.1 Technologies

4.1.1 Server-side

4.1.1.1 Choosing a Development Framework

A web framework is a collection of libraries that facilitate common web development tasks such as connecting to a database or responding to HTTP requests. Given that there are a plethora of web frameworks in use today, the task of choosing one can be very cumbersome. Since I already had experience with Django Web Framework [17] and Express Web Framework [18], which are the two most used frameworks for implementing REST APIs (the API design model is chosen to be REST in Section: 4.2.1.1), I narrowed down my options to just these two frameworks. As both of them were extremely easy to set up, provided database abstraction via the use of object-relational mapping [19], had extensive documentation, were regularly updated, and provided easy-to-use libraries for testing, I could have chosen either one of them to implement my REST API without any problems. However, I discovered that Express is based on Node.js [20], a JavaScript runtime that uses the V8 [21] engine developed by Google for use in their Chrome browser, which makes use of the event loop (described in more detail in Section 4.1.1.2). Compared to Django, which makes use of traditional web-serving methods where each HTTP request spawns a new thread, in turn consuming a lot of memory, Node.js operates on a single thread using non blocking (asynchronous) calls, allowing it to support thousands of concurrent connections (addressing the scalability non-functional requirement identified in Section 3.3). This behaviour can be clearly seen in Figure 4.1.

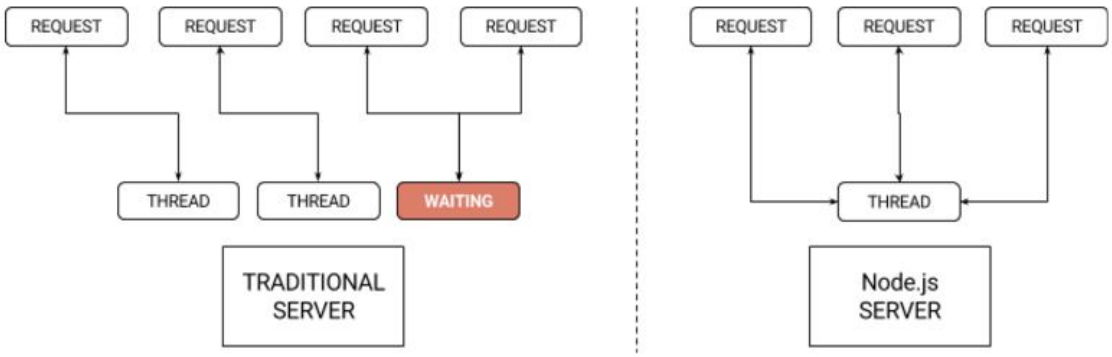

Figure 4.1: Traditional server creates new thread from limited pool, or waits for available thread. While Node.js server handles event-based callback on a single thread

The event loop was the primary reason that led me to choose Express over Django, but another Node.js feature that strengthened my choice further was the built-in support for package management using Node Package Manager (NPM) [22]. NPM makes a set of reusable components publicly available via an online repository, and

allows easy installation of them, along with version and dependency management of applications.

4.1.1.2 Introduction to Express Web Framework

Express is a minimal framework built on top of the Node.js platform, which provides the essential functionality to build web applications. Traditionally, the JavaScript [23] programming language has only been used for writing client-side web application code, however Node.js is a platform that allows JavaScript to be used to write server-side web applications. Although one can directly use the low-level APIs provided by Node.js to build web applications, Express makes this process a lot easier by abstracting away the complicated details of Node.js. The main components of Express that will be needed to understand the code structure and dependency choices are described in the sections below.

Modules

One of the major design flaws of JavaScript is that it is hard to write modular code with it since it lacks a standard for dividing the code base into multiple source files. This can easily become a problem for large-scale projects as having all the code in a single file can make it hard to navigate and debug. Node.js solves this problem by allowing the developer to create modules [24]. A module is essentially a single JavaScript file that encapsulates related code. A module in Node.js has three key features:

- A `require()` method which is used to import other modules into the code.
- An `exports` object which allows one to expose specific bits of a module to other modules.
- A `module` object used for providing metadata about a module (for example, name of the file containing the module). This object also contains a reference to the `exports` object as a property.

To understand them better, here is a code snippet showing a module named `hello_world.js` which exposes a method named `exposedMethod`:

```
var exposedMethod = function() {
        console.log("hello world");
}
module.exports.exposedMethod = exposedMethod;
```

Now a different module can access the method exposed by `hello_world.js` as follows:

```
var helloWorldModule = require("hello_world");
helloWorldModule.exposedMethod();
```

Routing

Routing essentially means to define communication endpoints (paths) [25] that a client-side application uses to communicate with the server-side application. The Express API provides routing methods corresponding to each of the HTTP methods (GET, POST, PUT and DELETE). The basic structure of a routing method is as follows:

```
// METHOD is an HTTP request method (get, post, put or delete)
// PATH is the URI that differentiates this route from other routes
// HANDLER_FUNCTION is a function that is executed when the client
// sends a request to this route
app.METHOD(PATH, HANDLER_FUNCTION)
```

To understand the routing methods better, here is a code snippet showing a sample endpoint:

```
var express = require('express'); // import express module
var app = express(); // initialize new express application

app.get("/hello", function(req, res) {
        res.send("hello from server");
}
```

The above code snippet defines an endpoint `/hello` which can be accessed using an HTTP GET request, and responds with the string "hello from server". Also, note that each handler method has access to the request object (`req`) and the response object (`res`). The request object represents an HTTP request and has properties for the HTTP headers, parameters, body, etc. The response object represents an HTTP response that the server-side application sends back to the client. In Express terminology, the handler method is also known as a middleware function [26].

Event-driven programming

As discussed in Section 4.1.1.1, Express follows an event-driven programming paradigm [27]. That is, one can register a function to an event and this function will only be executed when the corresponding event occurs. The way this works is that a single thread is responsible for running an infinite loop known as the event loop. Whenever an event occurs, it gets added to a queue of events. The event loop then removes events from this queue one by one, executing the method registered to it. To understand this paradigm better, here is a code snippet showing a method which only executes when the content of the file `hello.txt` has been read:

```
var fileSystem = require("fs"); // import module for reading files

fileSystem.readFile("hello.txt", function(err, data) {
    if (err) {
        console.log("an error occurred");
        return;
    }
    console.log(data.toString()); // print file contents to console
});
```

4.1.1.3 Code Structure

Since Express allowed me to divide my server-side code into different modules, I had to decide how to split my code into modules with different responsibilities, and then choose a folder structure for organizing them, in order to make the code base easy to maintain. I decided to use a slight variant of the Model-View-Controller architectural pattern (MVC) [28]. The MVC pattern separates application logic into three separate classes:

- **Model** is responsible for handling the application data. For example, inserting, updating and removing data from the database based on the instructions sent by the controller.
- **View** is responsible for rendering the different screens that will be seen by the users.
- **Controller** is responsible for controlling the interactions between the view and the model. For example, the controller can receive input from the user via the view, and then update the model based on the received input.

However, in my case, the client-side application was responsible for rendering the different screens for the user based on the data received from the different endpoints exposed by the server-side application. Therefore, I ignored the View part of the pattern and came up with the following structure:

- **controllers/:** This directory contains all modules which define routes served by the server-side application. Each logical part of the application has its own controller. For example, all requests concerning the `User` model will go in a controller module named `user_controller.js`.
- **models/:** This directory contains all modules which directly interact with the database. Each logical part of the application has its own model. For example, the schema defining the `User` model, along with the methods which modify the `User` model will go in a model module named `user_model.js`.
- **middlewares/:** This directory contains Express middlewares, i.e. all middleware functions that need to be regularly used within controllers. For example, a middleware function responsible for protecting an endpoint against unauthorized access will go in a module named `auth_middleware.js`.

- **utils/:** This directory contains utility code that is needed by various models, controllers and middlewares.
- **scripts/:** This directory contains all command-line scripts. For example, a script to launch the server-side application will go here.
- **tests/:** This directory contains all modules that contain code for testing the different components of the server-side application. For example, a module that tests database operations on the `User` model will go in a test module named `user_test.js`.
- **server.js:** This file is the main entry-point of the server-side application. It initializes the server-side application and connects all the other components (models, controllers and middlewares) together.
- **package.json:** This file tracks all the dependencies of the server-side application, and provides application metadata like the name and version of the application.

4.1.1.4 Base Dependencies

Although Express Web Framework provided me with a myriad of HTTP utility methods that could be used for creating APIs, it still lacked some essential modules which I describe below.

morgan

In order to effectively manage a web server, it is an absolute necessity to get feedback about the activity of the server, as well as any problems that might be occurring. In order to achieve this, I used an HTTP request logger called `morgan` to log each and every HTTP request to the command line. Here is some sample output from `morgan`:

```
POST /api/authenticate/google 200 323.515 ms
GET /api/users/56cb260316b82a8f118d7fa7 200 2.150 ms
POST /api/authenticate 401 229.704 ms
GET /api/authenticate 404 0.339 ms
```

If we consider the first request in the above output, we can clearly see that a `POST` request was made to <`base_url`>`/api/authenticate/google`, and a successful response with status code 200 (OK) was returned in 323.515 milliseconds. This kind of feedback helped me identify problems like:

- requests being sent to the wrong API endpoint;
- requests taking too long to process;
- requests using the wrong HTTP method, and
- requests returning the wrong HTTP status code

async.js

As discussed in Section 4.1.1.1 and Section 4.1.1.2, Express follows an event-driven programming paradigm. In this paradigm, if a function runs asynchronously, it does not stop subsequent function calls from executing before it finishes running itself. For example, if there are four tasks that need to be run synchronously with the nth task being dependent on the result of the (n-1)th task, the code structure would look like this:

```
task1(function(result1) {
    task2(result1, function(result2)) {
        task3(result2, function(result3)) {
            task4(result3, function(result4)) {
                console.log(result4);
            }
        }
    }
});
```

This way of organizing function calls is known as the callback pattern. In this pattern, a function is being passed as a parameter to another asynchronous function. If the above code snippet was a part of a larger program, it would have been extremely hard to understand the flow of control between the different function calls. Therefore, in order to prevent heavy nesting of function calls and manage asynchronous control flow better, I used a library called `async.js` [29]. `async.js` provides a control flow function called `waterfall()` which "runs the array of functions in series, each passing their results to the next function in the array" [30]. This function can be used to make the above code snippet a lot more legible as follows:

```
async.waterfall([
    task1,
    task2,
    task3,
    task4
], function(error, result) {
    // if there was no error, result = output of task4
});
```

Mongoose

Since I had decided to use a document-oriented database system for persisting server-side application data (see Section 4.2.1.2), and was also using the MVC architectural pattern to structure my server-side code (see Section 4.1.1.3), I wanted to find a tool that would let me represent my database collections as separate models. MongoDB (a document-oriented database system chosen in Section 4.2.1.2), on its own, did not provide a way to do so since document-oriented database systems are meant to be very unstructured. In order to solve

this problem, I decoded to use `Mongoose` [31], an object modelling tool for MongoDB. `Mongoose` essentially provided me with an abstraction over the raw data stored in the MongoDB database, so that I could work with objects belonging to the different models instead of working directly on raw data. This object-oriented way of interacting with the database made my code a lot more readable and easier to test and debug.

4.1.1.5 Database schema

In increment 1 (see Section 4.3), it was decided to make the client-side application responsible for retrieving and storing transport related data from the TFE API. Therefore, the server-side application ended up having a single `User` collection as it was only used for the purpose of authenticating users' credentials. The schema for this collection is shown in Figure 4.2.

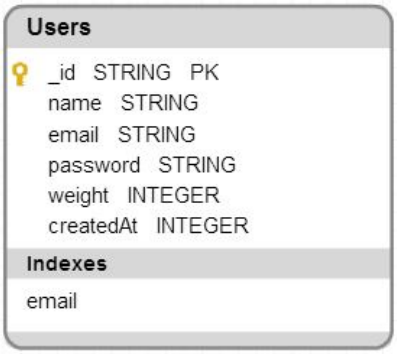

Figure 4.2: Database schema for Users collection

Here is a short description of each of the fields in Figure 4.2:

- **_id:** Unique identifier managed by MongoDB engine.
- **name:** Full name of the user.
- **email:** Email address of the user. Since most user related queries were performed using this field, it was also set as an index to improve query performance.
- **password:** Hashed version of the user's password (see Section 4.2.1.3 - Authentication strategy).
- **weight:** Weight of the user in kilograms. Used for computing the number of calories burned.
- **createdAt:** The time at which the user account was created.

4.1.2 Client-side

4.1.2.1 Choosing a Target Platform

Before I could start building the client-side application, I had to choose the target mobile operating system. According to a report by International Data Corporation (IDC) [32], the two most dominant mobile operating systems in terms of market share were Google's Android OS (82.8% in 2015 Q2) [4] and Apple's iOS (13.9% in 2015 Q2) [33]. Based on this report and the compatibility non-functional requirement which stated that the application must be able to reach a large audience (see Section 3.3), I narrowed down my options to only these two operating systems. Then I considered each functional requirement separately and read the relevant documentations to check whether they can be implemented on both the platforms. After finding that both platforms were equally suitable for the project, I ended up choosing the Android operating system because of its application store - Google Play Store. Google Play Store provides the perfect environment for applications in their early stages. Developers can quickly react to user feedback and have an update available to the users in a few hours. On the other hand, Apple's App Store's approval process frequently takes more than a week, and applications must be resubmitted if an issue is detected. Since I was required to refine my application through rapid feedback at the end of each increment, these delays and the added uncertainty of rejection would not have been suitable for me.

As Android devices run a lot of different versions of the Android operation system, I had to choose the minimum version that my application would support. This choice determined the oldest version of Android my application could be installed on. Based on the latest report on Android fragmentation by Open Signal [10], over 85% Android devices were running at least version 4.1 (Jelly Bean) when I started working on the project. Also, the fact that Project Butter [34] - an initiative to make the Android operating system quicker and more responsive for users, was only introduced in version 4.1 (Jelly Bean), I decided to choose version 4.1 as the minimum version that my application supported.

4.1.2.2 Introduction to Android Application Framework

Android is an operating system that is based on the Linux kernel. Android application framework essentially provides a high-level API to Android applications in the form of Java classes, and Java is the programming languages with which Android applications are developed. The main components of an Android application are described in the sections below.

Activity

An activity represents a single screen of an Android application. In terms of the MVC architectural pattern, an activity is analogous to a controller since it

lays out the user interface and handles user interactions. An application usually consists of multiple activities which are loosely coupled to each other. There's also a designated main or launcher activity which is the first activity to open when a user launches an application. Also, different activities communicate with each other using intents (described below).

Fragment

A fragment can be thought of as a sub-activity since it is usually used to represent a portion of a screen. They are mostly used in large-screen devices (tablets) where it can be convenient for the user to see two different views at the same time. For example, an email application could have an activity with two fragments in it, sharing the screen equally. The first fragment would display a list of emails (sender, subject and time stamp), and whenever the user clicks on an email, its content will be displayed within the second fragment. Even though this project was targeting small-screen devices, I still made use of fragments so that later on when there's a requirement to support large-screen devices, I can do so by simply adjusting the XML layouts (described below). Also, sometimes fragments can even be used to represent the entire screen, and then based on user interactions, they can simply be switched by other fragments within the same activity, since it is much faster to switch fragments compared to starting up new activities.

Intent

Intents are messages that allow an activity to start another activity to perform some functionality. For example, in an email application if the user clicks on the reply button for an email, the activity which contains the reply button will start up another activity, passing in information about the email that the user wants to reply to.

Service

By default, all components of an Android application run on the same thread, known as the main thread. Since the main responsibility of the main thread is to lay out the user interface and handle user interactions, any tasks that take longer than a few milliseconds to complete (like database queries or network operations) would end up blocking the main thread, leading to poor user experience. Therefore, the Android application framework provides a component called a service, which allows applications to perform long-running tasks in the background, without affecting the user experience. For example, a service might download a video file in the background while the user is on a completely different application. This is only possible because a service is not tied to the life cycle of the activity that started it.

Layout

Activities and fragments define their user interface via XML layout resource files. Although this can also be done using Java code, XML layout files are still the preferred way as they help to separate the programming logic from the layout definition (see MVC architectural pattern in Section 4.1.1.3). Use of XML layouts also allows the developer to reuse the same layout file for different activities or fragments.

4.1.2.3 Base Dependencies

EventBus

As discussed in section 4.1.2.2, I decided to use multiple fragments within a single activity as switching between fragments is faster than starting up new activities. I also made use of services to run long running tasks in the background. In both these use cases, data had to be passed between the different application components. For example, if an activity had two child fragments - fragment A and fragment B, and based on the user interaction in fragment A, some new content had to be displayed in fragment B. In order to achieve this, I could have made the parent activity responsible for passing data between the two fragments (as suggested by the official documentation [35]). However, this would have resulted in tight coupling between the activity and the fragments, making it difficult to modify one part without impacting the other in the future. In order to avoid this tight coupling, I decided to use the publish-subscribe pattern via a library called EventBus [36]. In this pattern, publishers are responsible for posting events in response to a change in state, while subscribers respond to the published events. So basically, these events transfer the required data from the publisher to the subscribers, minimizing the coupling between the two, and therefore making the application code easier to maintain. This behaviour can be clearly seen in Figure 4.3.

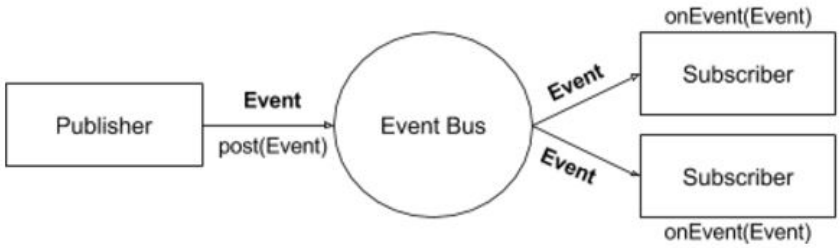

Figure 4.3: Overview of the publisher-subscriber pattern used by EventBus

GSON

When designing the server-side API, it was decided to use JSON as the data exchange language (see Section 4.2.1.3). So every time the client-side application received a response from the server, it had to convert the JSON representation

into Java objects. This can be a very tedious and error-prone task if done using the JSON serialization libraries provided as part of the Android Application Framework. To understand it better, let's consider the following JSON response that needs to be converted to a Java object:

```
"success": true,
"data": {
    "id": 1,
    "name": "Manas Bajaj",
    "email": "manas.bajaj94@gmail.com",
    "weight": 70
}
```

Now this is how the above JSON response can be converted into a `User` object using Android's JSON serialization library:

```
// Note that 'data' holds the JSON response as a String
JSONObject json = new JSONObject(data);
User user = new User();
user.setName(json.getString("name"));
user.setId(json.getInt());
user.setEmail(json.getString("email"));
user.setWeight(json.getInt("weight"));
```

As we can see in the above code snippet, it is extremely easy for the developer to get the name of a key wrong, or even forget to lookup a particular key. Furthermore, if the JSON response included nested structures, the amount of conversion code required would have increased considerably. In order to avoid such problems, I decided to use GSON [37], a serialization library developed by Google. Here is the GSON equivalent of the above code snippet:

```
// the only requirement here is that the field names of User class must
// match the key names in the JSON response
Gson gson = new Gson();
User user = gson.fromJson(data, User.class);
```

AsyncJob

As discussed in Section 4.1.2.2, networking and database operations should not be executed on the main thread, as they would end up blocking the main thread, and therefore making the application unresponsive. To address this issue, Android Application Framework provides a mechanism called `AsyncTask` [38], for executing operations in a background thread without having to handle thread creation and execution. An `AsyncTask` has the following structure:

```
private class SampleAsyncTask extends AsyncTask<String /*task input*/,
    Void, String /*task output*/> {
    protected Bitmap doInBackground(String... strings) {
    // This method executes the long-running task in a background
    // thread
    ...
    }

    protected void onPostExecute(Bitmap result) {
    // This method is executed in the main thread with access to the
    // result of doInBackground()
    ...
    }
}

// Execute the task from the activity or service where its needed
new SampleAsyncTask("task input").execute();
```

Although this solution works just fine, it forces the developer to create a new Java class every time they want to do something as trivial as running a network operation. Since the client-side application required a lot of tasks to run in the background, I did not want to pollute the code base with such classes. In order to avoid this issue, I decided to use a library called `AsyncJob` [39]. Using `AsyncJob`, I could simply define and run background tasks from the same place in the code, without needing to define local classes (as seen in the code snippet below).

```
new AsyncJob.AsyncJobBuilder<Boolean>()
    .doInBackground(new AsyncJob.AsyncAction<Boolean /*task output*/>() {
        @Override
        public Boolean doAsync() {
            // This method executes the long-running task in a background
            // thread
            return true;
        }
    })
    .doWhenFinished(new AsyncJob.AsyncResultAction<Boolean>() {
        @Override
        public void onResult(Boolean result) {
            // This method is executed in the main thread with access to the
            // result of doAsync()
        }
    }).create().start();
```

ActiveAndroid

In increment 1 (see Section 4.2.1.5), I decided to use a denormalized SQLite database on the client-side for storing frequently used data. So in order to convert my database tables to object-oriented models (and for reasons discussed in 4.1.1.4 - Mongoose), I made use of `ActiveAndroid` [40], an object relational mapper

for the Android operating system. It allowed me to save and retrieve SQLite database records without writing a single SQL statement. For example, here is a code snippet which defines a `User` model using Java annotations [41] to create a database table named `Users` with two columns `name` and `age`, and then adds and queries a new record:

```
// Create new table named Users
@Table(name = "Users")
public class User extends Model {
    @Column(name = "name")
    public String name;

    @Column(name = "age")
    public int age;

    public User() {
        super();
    }
}

// Add new user to database
User user = new User();
user.name = "Manas Bajaj";
user.age = 21;
user.save();

// Retrieve the newly added user from database
User user = new Select().from(User.class).where("name=?", "Manas Bajaj")
    .executeSingle();
```

4.1.2.4 Code Structure

Since an Android application consists of a lot of different components, they should always be neatly organized with a clear folder structure that makes the code easy to navigate and maintain. Organizing the code solely based on the MVC architectural pattern (as discussed in Section 4.1.1.3), would not have worked because that would have still led to a lot of files being in the same folder. So instead, I combined the MVC pattern with another structuring approach called package-by-feature. Using the package-by-feature approach, all items related to a single feature are placed into a single directory or package. That is, items that work closely together are placed next to each other, instead of being spread out all over the application. Using this combination of approaches, I ended up with the following folder structure:

```
models/
network/
ui/
 |-- screen1/
       |-- activities/
       |-- events/
```

```
       |-- fragments/
       |-- services/
 |-- screen2/
       |-- ...
 |-- ...
utils/
res/
App.java
```

The files that go into these folders are described below:

- **models/:** This folder is used in the same way as it was used for the server-side code. It contains model classes which interact with the database directly. Each logical part of the application has its own model. For example, the schema defining the `User` model, along with the methods which modify the `User` model will go in a Java class named `User.java`. Also, since models usually have application-wide use, it is not possible to package them by feature.

- **network/:** This folder contains Java classes for communicating with the server-side application. These Java classes enable the controllers (activities and fragments) to send HTTP requests to the server and retrieve responses in the form of Java objects instead of raw JSON data. For example, a class responsible for accessing the server-side API endpoints dealing with users will be named `UserService.java` and be placed in this folder. Note that these network services are not be confused with Android services (see Section 4.1.2.2).

- **ui/:** This folder contains all the user-interface related code. This includes the controllers (activities and fragments), `EventBus` events that enable inter-fragment communication, and Android services. The name of each of these classes ends with the the type of components they are. For example, a fragment representing the screen for logging in users is called `LoginFragment.java`.

- **utils/:** This folder again used in the same way as it was used for the server-side code. It contains utility code that is needed application-wide.

- **res/:** This folder contains the XML layout files which the activities and fragments use to define their user-interface.

- **App.java:** This class is the main entry-point of the Android application. It initializes all the network services, providing them with all the Android-specific resource they need to function properly, and also sets up the database connection for the model classes.

This way of organizing my code structure ensured that as the client-side application grew in size, the number of packages increased, while the number of classes within each package stayed roughly the same, allowing for easier code navigation. It also helped me stick to the "high cohesion, low coupling" software engineering principle, as the different feature packages had low coupling between them, while

each package had high cohesion within itself.

4.1.2.5 Database schema

In increment 1 (see Section 4.2.1.5), I decided to use a denormalized SQLite database on the client-side for storing frequently used data. This was done to address the performance and availability non-functional requirements (also discussed in Section 4.2.1.5) that were identified during the requirement engineering phase (see Section 3.3). The final database schema for the client-side application can be seen in Figure 4.4 below.

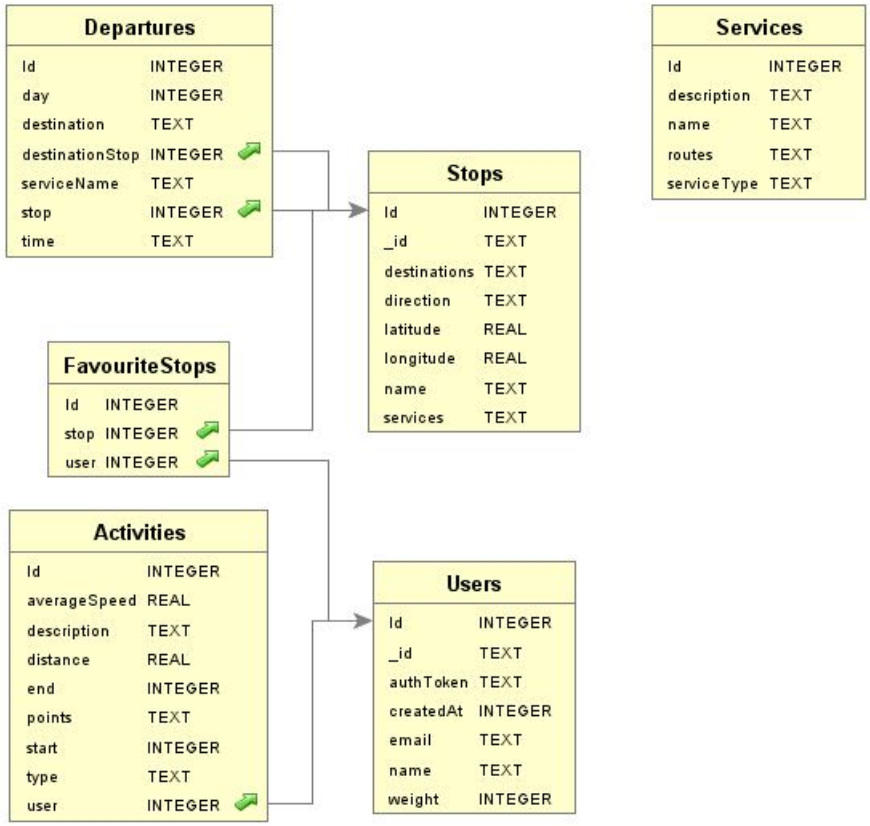

Figure 4.4: Client-side database schema

The sections below describe the fields in each of the tables shown in Figure 4.4.

Users

This table stores the details of authenticated users. It has the following fields:

- **id:** Auto incrementing identifier managed by SQLite engine.
- **_id:** Auto incrementing identifier for server-side MongoDB `user` collection. This field essentially connects the client-side user object with the server-side user object.

- **authToken:** JWT mapped to a user (see Section 4.2.1.3).
- **createdAt:** The time at which the user account was created.
- **email:** Email address of the user.
- **name:** Full name of the user.
- **weight:** Weight of the user in kilograms. Used for computing the number of calories burned.

Stops

This table stores the details of Edinburgh bus stops. It has the following fields:

- **id:** Auto incremented identifier managed by SQLite engine.
- **_id:** Stop identifier returned by TFE API.
- **destinations:** List of destinations of the services served by the stop.
- **direction:** The direction of the stop in compass points (NE, SE etc).
- **latitude:** The latitude of the stop.
- **longitude:** The longitude of the stop.
- **name:** The name shown on the stop flag.
- **services:** List of services served by the stop.

Services

This table stores the details of all bus services served by Edinburgh bus stops. It has the following fields:

- **id:** Auto incremented identifier managed by SQLite engine.
- **description:** Description of the service containing the main location names served by the service. For example, "Clermiston - Brougham Place - Easter Road".
- **name:** Name of the service.
- **routes:** A list of routes. Each item in this list is a complex JSON object containing a destination, a series of geographical coordinates and a list of stops on the route (in order).
- **serviceType:** Can be day, express, night.

Departures

This table stores the departure times for Edinburgh bus stops. It has the following fields:

- **id:** Auto incrementing identifier managed by SQLite engine.
- **day:** The day of operation for the departure. 1 is Mon-Fri, 5 is Saturday and 6 is Sunday.
- **destination:** The destination displayed on the bus' destination board.
- **destinationStop:** The `id` of the destination stop in `Stops` table.
- **serviceName:** The name of the bus service that will depart from the stop.
- **stop:** The `id` of the stop in `Stops` table the departure belongs to.
- **time:** The time of departure in 24-hour format.

FavouriteStops

This table stores users' favourite stops. It has the following fields:

- **id:** Auto incrementing identifier managed by SQLite engine.
- **stop:** The `id` of the favourite bus stop in `Stops` table.
- **user:** The `id` of the user in `Users` table that the favourite stop belongs to.

Activities

This table stores details of the activities performed by the user (see Section 4.2.7 to know about the different types of activities). It has the following fields:

- **id:** Auto incrementing identifier managed by SQLite engine.
- **averageSpeed:** Average speed of user in kilometres per hour.
- **distance:** Total distance covered by the user in kilometres.
- **end:** Unix time stamp indicating the end of activity.
- **points:** Ordered list of geographical coordinates representing the user's path.
- **start:** Unix time stamp indicating the start of activity.
- **type:** Can be "WAIT_OR_WALK" or "JOURNEY_PLANNER".
- **user:** The `id` of the user in `Users` table that the activity belongs to.

4.2 Increments

4.2.1 Increment 1

The functional requirements for this increment were as follows:

- Each user should be able to have their own account set up within the application. This will store information such as their favourite bus stops and physical activity statistics. This will also enable the users to restore their existing data if they decide to switch devices or use multiple devices and want to synchronize their data across all devices.
- Users should be able to view bus stops near them, along with the walking duration to each stop.

The above functional requirements were converted to the following sequence of steps:

1. Choose a design model for the server-side API.
2. Choose a database technology for persisting application data.
3. Set up an authentication server which provides a network service that the client-side application will use to authenticate the credentials of its users.
4. Make the authentication server accessible via the World Wide Web.
5. Set up a client-side application that is able to authenticate the credentials of its users.
6. Enable the client-side application to retrieve bus stop information from the TFE open data API, and then display the bus stops nearest to the user.

The following sections describe how each of the above steps were performed.

4.2.1.1 Step 1: Choose a design model for the server-side API.

An authentication server essentially exposes an interface which the client-side application uses to authenticate the credentials of its users. This interface is known as a server-side application programming interface (API). Before I could start developing a web API, I had to choose which design model to use. A design model specifies exactly how structured information will be exchanged between the server and the client. The most dominant design models are Simple Object Access Protocol (SOAP) [42] and Representational State Transfer (REST) [43]. The main difference between the two is that REST strictly works on top of Hypertext Transfer Protocol (HTTP) [44], while SOAP can work on top of any application layer protocol. Also, REST makes data available as resources while SOAP makes data available as services. For example, REST would have a resource named "user", while SOAP would have a service named "getUser".

The main reasons that confirmed my choice of design model to be REST are as follows:

- Since mobile applications usually require a lot of back-and-forth messaging, and that mobile devices can lose reception at any point, I wanted to make sure that when such an event occurs, the mobile application is able to retire the pending request and resend it when the device regains reception in as

little time as possible. Given that REST is stateless, i.e. no state is stored on the server-side and therefore requests are independent of each other, it allows requests to be retired independently. On the other hand, SOAP is much more verbose as each request needs to know and retain changes made in previous requests, thus making the resend process a lot more complicated and time-consuming.

- REST permits the usage of many formats for encoding the data exchanged between the client and server, while SOAP only permits Extensible Markup Language (XML) [45]. Given that XML is a lot more verbose than JavaScript Object Notation (JSON) [46], which is the most common data format used by REST APIs, REST is able to make efficient use of the bandwidth as there is less data to transfer. Moreover, JSON's simpler syntax makes it faster to parse than XML, further minimizing the client-server communication time.

Note that the above reasons also address the performance non-functional requirement (identified in Section 3.3) as faster client-server communication time implies that the user will be able to view the requested information faster.

4.2.1.2 Step 2: Chose a database technology for persisting application data.

The two most prevalent database systems that were in use when I started working on this project were relational database management systems (RDBMS) and document-oriented database systems. RDBMS provide a store of related data tables. For example, information about bus services can be stored in a table named `services`:

name	description	service_type
1	Clermiston - Easter Road	day
2	Brougham Street - Waverly	night

Table 4.1: Sample data stored in an RDBMS table named `service`. Every row is a different service record.

In RDBMS, a schema must be defined before adding records to the database. A schema is written in a formal language such as Structured Query Language (SQL) (which is the most dominant language for managing RDBMS), and it provides a blueprint of all the tables in the database, the relationships between these tables, and constraints on the type of data that can be entered into the tables. This design is very rigid, for example, one cannot insert a number where a string is expected.

On the other hand, document-oriented database systems consist of collections, which are analogous to RDBMS tables. These collections contain documents, which are analogous to RDBMS records. Also, since they do not require explicit schemas to be defined, the developer is free to store any kind of information in the

database. Here is how the above RDBMS table would look like in a document-oriented store:

```
{
    {
        name: "1",
        description: "Clermiston - Easter Road",
        service_type: "day"
    },
    {
        name: "2",
        description: "Brougham Street - Waverly",
        service_type: "night"
    }
}
```

In order to check if any data that I would end up storing in the future would be able to exploit the freedom offered by document-oriented database systems, I started to look at the structure of data returned by the different endpoints served by the TFE API. I found an endpoint that returned the route information for each bus service. As a single service could have more than one route, an RDBMS would require me to create a new table named `routes` and then link it to the `services` table using the service name as the foreign key constraint. This would minimize data redundancy as I would not be repeating route information for every service, but just the reference to it. This redundancy minimization technique is known as normalization. Similar normalization could also be performed on document-oriented database systems (service collection would simply reference a document in the route collection using a route identifier), however, based on the structure of route information returned, it did not seem practical to me. To understand the problem better, here is a part of the route information returned by the TFE API:

```
"routes": [
    {
        "destination": "Clermiston",
        "points": [
            {
                "latitude": 55.96799,
                "longitude": -3.168343
            },
            {...}
        ],
        "stops": [
            36234249,
            36234252,
            {...}
        ]
    },
    {...}
]
```

So looking at the above structure, storing points information in RDBMS would require me to create a separate `points` table and then link each point record to a route in the `routes` table. Also, since points are ordered, a new field would have to be added in order to keep track of the position of the point within the set of all points for a particular route. On the other hand, in a document-oriented database system, I could simply opt to denormalize the service document by storing the above nested structure of routes as it is. This would lead to extremely fast queries since I would only have to make a single query in order to obtain a particular service along with all its routes.

Based on the reasons outlined above, I decided to choose MongoDB [47], the most popular document-oriented database system, for persisting server-side application data.

4.2.1.3 Step 3: Setup an authentication server which provides a network service that the client-side application will use to authenticate the credentials of its users.

Now that the API design model had been chosen, I had to move on and implement a REST API. A REST API must adhere to three main rules:

- *It should have a stateless design.* No client context or state should be stored on the server. Each request from the client must contain all the information necessary to service the request.
- *It should have self-descriptive messages.* Each message should include enough information to describe how to process the message. For example, which parser to use may be specified by a MIME type [48].
- *It should use existing standards.* For example, the API should use existing features of HTTP like HTTP methods (GET, PUT, POST and DELETE) and HTTP status codes.

I started to design the API, keeping the above rules in mind. The design process is described in the sections below.

Output format and structure

The output format for the API was chosen to be JSON. Traditionally, XML has been used, however it is hard to read, and its data model isn't compatible with how most programming languages model data. Also, since JSON is more compact, it can be transmitted faster than XML (as discussed in Section 4.2.1.1).

If a request to the API fails, the reason for failure must be communicated to the API user. Although HTTP status codes can be used for general errors like "resource not found" and "unauthorized access", in some cases the reason for failure is a lot more specific than that. For example, a user might be trying to register with a username that already exists in the database. To communicate

such messages to the user, I decided to design my own specification for formatting JSON responses. Basically, the responses were separated into two types: "success" and "fail", which was indicated by the boolean field "success". For example, a successful response that complies with this specification would look like this:

```
"success": true,
"data": {
    "name": "Manas Bajaj",
    "email": "manas.bajaj94@gmail.com"
}
```

While an unsuccessful response would look like this:

```
"success": false,
"error": {
    "code": 404,
    "message": "The provided username is already in use."
}
```

URL structure

The key principles of REST involve separating the API into logical resources. The only resource that was identified for the purpose of authentication was `user`. After resource identification, I identified what actions applied to them. RESTful API design rules (see Section 4.2.1.3) suggest the use of HTTP methods to handle CRUD (Create, Read, Update and Delete) operations on resources. The resources are manipulated using HTTP requests where the method has a specific meaning. URLs should be short and descriptive and utilize the natural hierarchy of the path structure. Table 4.2 shows the mapping between HTTP methods and the different endpoints I created for the `user` resource.

HTTP Method	Description
GET	GET /api/users/123 will retrieve the user with id = 123
PUT	PUT /api/users/123 with body "name=newName" will update the name of the existing user with id = 123 to "newName"
POST	POST /api/users/ with body "name=newName" will create a new user with name = "newName" and id will be auto assigned
DELETE	DELETE /api/users/123 will delete the user with id = 123

Table 4.2: Accessing user resource using various HTTP methods

Authentication strategy

The security non-functional requirement that was identified during the requirement engineering phase (see Section 3.3) suggested that all user data stored on

the client-side and server-side must require authentication to access. In order to fulfill this requirement, I had to protect the endpoints listed in Table 4.2 in such a way that only authenticated users can access them.

Since the HTTP protocol is stateless, if a user authenticates with a username and password in one request, then on the next request, the server would not know who they are and they would have to authenticate again. The traditional way of making the application remember authenticated users is to store their information on the server (cookie-based authentication). A REST-ful API should be stateless, which means that authentication should not depend on HTTP cookies [49] or sessions [50]. Instead, each request should come with some sort of authentication credentials. Therefore, I decided to use a stateless authentication scheme known as token-based authentication. The sequence of events that takes place in order to authenticate a user is as follows (and can also be seen in Figure 4.5):

1. User attempts to authenticate by sending their username and password to the server.
2. Server validates the credentials and provides a token to the client.
3. Client stores the received token and sends it back to the server along with every future request (as part of HTTP Authorization header [51]).
4. Server verifies the token and responds with requested data.

Step 2 states that the server validates the credentials of the user before issuing a token to them. This validation process is essentially comprised of checking whether the provided email address and password match the ones stored in the database. However, storing passwords in plain text is considered to be a major security vulnerability because in case an attacker manages to get read-only access to the server data, they would easily be able to use other users' email-password combinations to sign in as them and get the corresponding administration powers. Also, the fact that people usually reuse passwords across several systems just makes matters even worse. In order to avoid this vulnerability, I decided to hash the user passwords before storing them. Hashing is the process of turning any amount of data into a fixed-length and irreversible string, that is, there is no key that would convert the hash back to the original text. I decided to use a hash function called bcrypt [52] to generate hashes for user passwords. The reason I chose bcrypt was that it was much slower (5 orders of magnitude [53]) than other common functions like MD5 and SHA1. This meant that even if an attacker got hold of a password hash, it would have taken them a long time to find the corresponding password by brute force. The final validation process was as follows:

1. Check if a user corresponding to the provided email address exists in the database. If not, return HTTP response with status code 401, indicating an authentication error. Else, proceed to next step.
2. Hash the provided password and compare it to the stored password hash. If

they match, return HTTP response with status code 200, indicating success. Else, return HTTP response with status code 401, indicating an authentication error.

Step 4 states that the server first verifies the token sent by the user and then responds with the requested data. This verification can be performed in two ways:

1. When a user authenticates successfully, the server provides them with a token and then stores the token in a server-side database. For all future requests that contain a token, the server verifies the token by simply checking if the token is stored in the database.
2. When a user authenticates successfully, the server provides them with a self-contained token which has been signed using a secret key held at the server. This way, all future requests that contain a token can be verified by decoding the signature part of the token using the same algorithm that was used to encode it, and then comparing the decoded secret key to the actual secret key. This kind of a token is known as JSON Web Token (JWT) [54]. It is a fairly new standard (proposed in May 2015), but is gaining popularity rapidly and is on the path to become an industry standard.

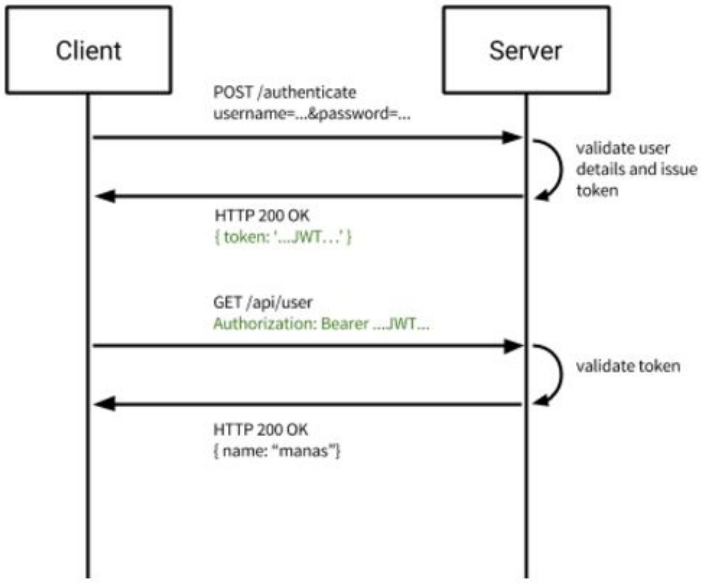

Figure 4.5: Client-server interaction in token-based authentication

Given that a REST-ful API must have a stateless design (see Section 4.2.1.3), I decided to use the JWT approach of token verification. The final implementation using the above design decisions helped me address the following non-functional requirements:

- **Performance:** In cookie-based authentication, the process of checking whether a session exists for a particular user in the database is more time consuming than simply parsing the decoded token to validate it, as done in token-based authentication.

- **Security:** Since cookies were no longer being used, I did not need to protect against cross site requests (Cross Site Request Forgery - CSRF). [1]
- **Scalability:** If cookie-based authentication is used along with multiple web servers (without a shared database) then there is an additional overhead in making sure that each user request gets directed to the same server every time. This is not an issue for stateless token-based authentication, and therefore horizontal scaling (adding more servers) can be performed with ease. Although scalability wasn't a problem for me during the evaluation phase of the development life cycle (see Chapter **??**), I just wanted to future-proof my application in case such a problem arises later on.

Authentication through external services

To enhance the usability of the application, I decided to give users the ability to use their existing accounts on other social websites to login to my application. Social login makes the account registration process extremely simple and fast as the users are no longer required to manually type in their credentials. Moreover, social login also obviates the need for them to remember another username/password combination. I chose to use Facebook and Google as the external authentication service providers because of their huge user base and easy-to-use software development kits (SDK) [56]. I then discovered that both, Facebook and Google, use an open standard for authorization, known as OAuth [57]. OAuth essentially works by delegating user authentication to the external service that hosts the user account, and authorizing third-party applications to access the user account without exposing their password. So in order to incorporate the external authentication service providers' authentication strategies with my own strategy (see Section 4.2.1.3), I came up with the following authentication flow:

1. User selects "Login with Facebook" or "Login with Google" button on the client-side application.
2. The client-side application uses Facebook/Google SDK to retrieve a short-lived access token for the user and sends it to the server.
3. Server verifies the integrity of the token by making a request to an endpoint on Facebook's/Google's authorization server. The response to this request is then checked to make sure that tokens issued to a malicious app cannot be used to access user data on my server (confused deputy problem [58]). Note that this also addresses the security non-functional requirement (see Section 3.3).
4. Server retrieves user's email address and name from Facebook's/Google's servers using the verified access token, and then creates a new account for the user using these details.

[1] "CSRF is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated." [55]

5. Server issues a JWT for the newly created user and sends it back to the client.

4.2.1.4 Step 4: Hosting the authentication server

Now that the authentication server-side application was ready, all I had do was to use a web hosting service to make my server-side application accessible via the World Wide Web. I decided to go for the most popular web hosting service - Amazon Elastic Compute Cloud (Amazon EC2 [59] for the following reasons:

- Using its simple web service interface, I was able to configure my server instance within minutes.
- The web interface allowed me to easily scale my server-side application both horizontally (by increasing the memory/disk capacity of any server instance) and vertically (by booting new server instances). Note that this addresses the scalability non-functional requirement (see Section 3.3).
- The web interface allowed me to easily control network access to my server instance by limiting the inbound and outbound traffic to specific ports and protocols.

In order to address the security non-functional requirement of encrypting all communications between the client and the server, I made sure that all inbound and outbound traffic was limited to port 443 and the secure version of Hyper Text Transfer Protocol (HTTPS) [60].

4.2.1.5 Step 5: Set up a client-side application that is able to authenticate the credentials of its users.

Choosing a data store

According to the authentication strategy discussed in Section 4.2.1.3, once the user's credentials have been authenticated, all future requests from the client-side application are supposed to contain the authentication token received from the server. Also, since some API endpoints can only be accessed by the user associated with the token, the details of the currently authenticated user (along with the assigned token) need to be persisted in some sort of a data store on the client's device. This would obviate the need to authenticate the user's credentials every time the application is started, and therefore improve application start-up time (note that this addresses the performance non-functional requirement identified in Section 3.3). In order to achieve this, I decided to use a SQLite [61] relational database, which is the only storage option provided by the Android Application Framework for storing structured data. The reason why Android only provides a SQLite database is because it is a self-contained database that does not require a server to be run on, and therefore is more suitable for mobile devices.

Also, since I wanted to allow multiple users to persist their data on the same device, I had to keep track of the currently authenticated user's ID in order to load the corresponding user's data from the database on application start-up. To achieve this, I decided to use `Shared Preferences`, which is essentially an API provided by Android to persist primitive data as key-value pairs in an XML file. I could have simply created a new table in the aforementioned SQLite database, however since that table would only always have a single record, it didn't make sense to use that approach.

Design brief

Now that I had decided where to store the details of the currently authenticated user, I moved on to design the user interface that would allow users to login and sign up for the application. But before writing any code, I decided to create a design brief listing the various UI elements that the user would see and interact with.

When the user views the "login and sign up" view, they would see:

- Two tabs at the top to switch between sign up and login sub-views.
- Input fields for email and password, and a button which can be clicked to login using the input from these fields [login sub-view].
- Social login buttons which can be clicked to login without typing in any details [login sub-view].
- Input fields for name, email and password, and a button which can be clicked to sign up using the input from these fields [sign up sub-view]

Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database model:** An `ActiveAndroid` model class named `User` was created to define the schema of the `Users` table in the database (see Section 4.1.2.5).
- **Utility:** A utility class named `PreferencesManager` was created to encapsulate the logic for storing and retrieving data to Android's `Shared Preferences` (see Section 4.2.1.5).
- **Network service:** A network service named `UserService` was created to access the various user and authentication related endpoints of the server-side API. This service was also responsible for persisting user data in the `User` table whenever a user authenticated successfully, and used `PreferencesManager` to keep track of the currently authenticated user.

- **Controllers:** A parent activity named `LoginActivity` was created, along with two child fragments `LoginFragment` and `SignupFragment`. These two fragments represented the login and sign up sub-views described in the above design brief. They used `UserService` to communicate with the server based on the user interactions, and based on the authentication result, either show an error message or start up a new activity to let the user access the restricted apart of the application.
- **Event:** An `EventBus` event named `OnAuthenticatedEvent` was used to let the `LoginActivity` know that the user has been successfully authenticated, so that `LoginActivity` could start up a new activity and let the user access the restricted part of the application.

These components are shown in the UML class diagram [62] in Figure 4.6.

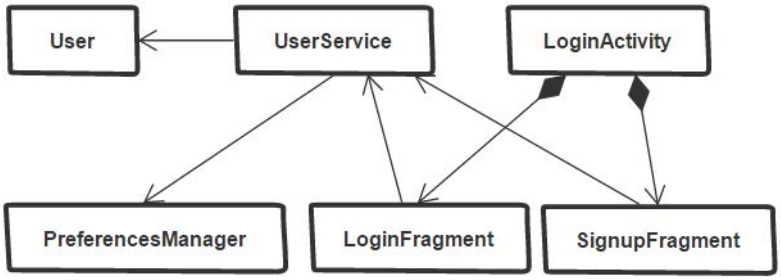

Figure 4.6: UML class diagram showing the key components needed for implementing client-side authentication

Screenshots

See Figure A.3.

4.2.1.6 Step 6: Enable the client-side application to retrieve bus stop information from the TFE open data API, and then display the bus stops nearest to the user.

Deciding where to store data retrieved from TFE API

Although I had decided to use a document-oriented database system on the server-side, to store the nested data structures returned by the TFE API (see Section 4.2.1.2), in this step I realized that it might have not been the best approach. The initial plan was to cache the frequently used data about stops, services and timetables on the server-side database in order to reduce the number of requests sent to TFE API. However, I realized that if the frequently used data was cached on the client-side application instead, it would save the user a lot of bandwidth and increase the response time of the application as in most cases, no HTTP requests would need be made and the required data would simply be retrieved from the local database. Furthermore, this would also fulfil the availability non-functional requirement that required the basic functionality of the application

to work even when the user is offline. In order the achieve this, I decided to use a denormalized SQLite database on the client-side (see Section 4.2.1.2 to understand why denormalization of data was required).

Design brief

When the user views the "nearby bus stops", they would see:

- A map displaying the bus stops that are nearest to the user's current location, which is also displayed on the map. Clicking on an empty area in the map changes the user's current location to be the point that was clicked, and the nearest bus stops are then updated. Clicking on a stop displays the name of the stop.
- A list showing the stops that are nearest to the user's current location (same stops as the ones displayed on the map). Each item of this list shows the name of the stop, the list of services served by the stop, the destinations of the services served by the stop, and the estimated walking duration to the stop. Clicking on an item of this list shows options to view departure times for the stop, add stop to favourite stops list, and to show the stop's location on the map. The first and second options are made functional in increments 2 and 4 respectively.
- User's current location is displayed, along with the time stamp indicating when this location was last updated.

Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database models:** `ActiveAndroid` model classes named `Stop` and `Service` were created to define the schema of the `Stops` and `Services` database tables respectively (see Section 4.1.2.5). `Stop` also provided a method to retrieve all stops nearest to the geographical coordinates provided as arguments. Instead of retrieving all stops and then filtering them based on the haversine distance [63] between their location and user's location, it first used SQL to only retrieve a small subset of all stops based on the ascending order of the difference between the geographical coordinates, and then filtered out the stops that had a haversine distance of more than 1 kilometre from the user's location. This method can be seen in the code snippet below:

```
public static List<Stop> getNearby(LatLng latLng, double maxDistance,
  int limit) {
    List<Stop> nearestStops = SQLiteUtils.rawQuery(Stop.class,
      "SELECT * FROM Stops S " + "ORDER BY ABS(ABS(S.latitude - ?) +
      ABS(S.longitude - ?)) ASC LIMIT ?",
      new String[]{String.valueOf(latLng.latitude),
      String.valueOf(latLng.longitude),
```

```
      String.valueOf(limit)});

    List<Stop> requiredStops = new ArrayList<>();
    for (Stop stop : nearestStops) {
        if (Helpers.getDistanceBetweenPoints(stop.latitude, stop.long,
          latLng.latitude, latLng.longitude) <= maxDistance * 1000) {
            requiredStops.add(stop);
        } else {
        break;
        }
    }
    return requiredStops;
}
```

Using this approach, the time needed to find the nearest bus stops was reduced considerably, making the application feel more responsive. Note that this addresses the performance non-functional requirement identified in Section 3.3.

- **Utility:** Two utility classes named `LocationProvider` and `ReverseGeocoder` were created. `LocationProvider` acted as a wrapper class, encapsulating the logic for retrieving user's last known location using various classes and interfaces provided by the Android Application Framework. `ReverseGeocoder` acted as a wrapper class, encapsulating the logic for reverse geocoding the geographical coordinates of user's current location. [2]
- **Network service:** Network services named `StopService` and `ServiceService` were created to retrieve all stop and service related data from the relevant TFE API endpoints and then populate the `Stops` and `Services` database tables respectively.
- **Controllers:** A parent activity named `HomeActivity` was created, along with a child fragment named `NearMeFragment`. `HomeActivity` was the first activity to start after a user had successfully authenticated. It had the responsibility of making sure that `Stop` and `Service` database tables were populated before the user was allowed to use the application, and it made use of the network service `StopService` to do so. `NearMeFragment` represented the "nearby bus stops" view, and handled all user interactions for this view. It made use of the utility classes `LocationProvider` and `ReverseGeocoder` to display the user's current current location in a human-readable form. It also made use of the model class `Stop` to query the stops nearest to the user's location.

These components are shown in the UML class diagram in Figure 4.7.

Screenshots

See Figure A.4.

[2] "Reverse geocoding is the process of back (reverse) coding of a point location (latitude, longitude) to a readable address or place name." [64].

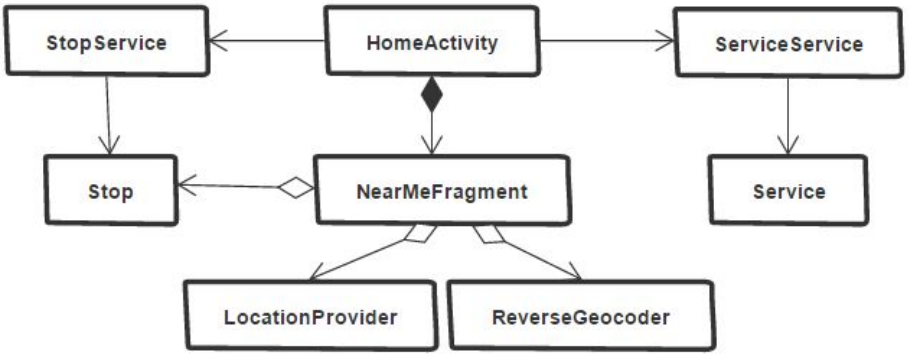

Figure 4.7: UML class diagram showing the key components needed for implementing "nearby bus stops" view

4.2.2 Increment 2

The functional requirement for this increment was as follows:

- Users should be able to view live departure times for stops and up-to-date weekly timetables for all services served by these stops.

Since this requirement was quite specific, and that the major technology choices had already been made in increment 1 (see Section 4.2.1), I did not need to break it down further into a sequence of steps.

As seen in Figure A.4, I provided the user with an option to view departure times for the stop they selected. So the task in this increment was to create a view for displaying the live departures and the weekly timetable ("departure times" view).

4.2.2.1 Design brief

When the user views the "departure times" view, they would see:

- A map displaying the selected stop, along with an estimate of the walking duration to the selected stop from the user's current location. Walking directions are also marked on the map.
- Some information about the stop they selected, for example, the name of the selected stop and the services served by it.
- A list of departure times that is updated live. Each item in this list displays the number of minutes remaining until the departure of the corresponding service.
- A list of non-live departure times that can be filtered by day of the week and time. Each item in this list displays the departure time in 24-hour format.
- A button to filter both of the above lists by service name.

- A button to add the selected stop to favourite stops list. This button is made functional in increment 4.

4.2.2.2 Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database models:** An `ActiveAndroid` model named `Departure` was created to define the schema of the `Departures` table in the database (see Section 4.1.2.5).
- **Network services:** A network service named `DirectionsService` was created. This service encapsulated the logic for retrieving walking directions and estimated walking duration between two geographical coordinates using Google's Directions API [65]. It was responsible for converting the JSON response returned by Google's directions API endpoint into a Java object (`Directions`) by extracting and restructuring the required information about the walking route. An existing service named `StopService` was updated to add the ability to retrieve timetable information for a specific stop and populate the `Departure` table accordingly. Unlike the `Stop` and `Service` tables, which were populated with all stop and service data at once, data was only added to the `Departure` table when the user selected a stop for which the timetable information did not already exist. As there was a lot more timetable data in total compared to all stop and service related data, this approach was taken to ensure that only the timetable data required by the user was persisted locally.
- **Controllers:** A parent activity named `StopActivity` was created, along with a child fragment named `StopFragment`. `StopActivity`'s received the identifier of the stop that was selected by the user from `NearMeFragment` (see Section 4.2.1.6), and passed it onto `StopFragment` as an intent message. This allowed `StopFragment` to know which stop it had to display data for. `StopFragment` represented the "departure times" view, and handled all user interactions for this view. It made use of the `LocationProvider` utility class to retrieve and display the user's last known location on the map. `StopService` was used to retrieve the departure times for the selected stop (the decision to retrieve it from the database directly, or to send an HTTP GET request to the relevant TFE API endpoint was made by `StopService` itself), while `DirectionsService` was used to retrieve the walking directions and duration from user's last known location to the stop. Also, in order to reduce the number of database queries and decrease response time, the the entire week's timetable was fetched at once and filtered using Java (instead of SQL) based on the time selected by the user.

These components are shown in the UML class diagram in Figure 4.8.

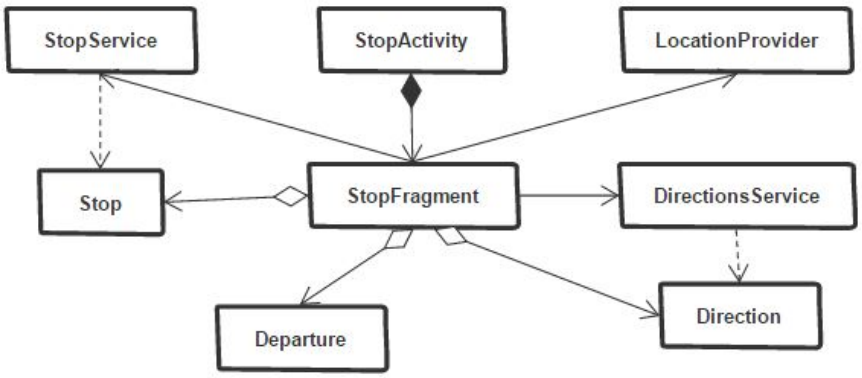

Figure 4.8: UML class diagram showing the key components needed for implementing "departure times" view

4.2.2.3 Screenshots

See Figures A.5 and A.6.

4.2.3 Increment 3

The functional requirements for this increment were as follows:

- Users should be able to view detailed route information for each service.
- Users should be able to view real time information on the position of buses.

The route information for this increment was to be displayed when the user selected an item from the list displayed in "departure times" view (see Figure A.5). Selecting an item from the "Live Departures" section opened the "live service view", while selecting an item from the "Timetable" section opened the "service view". The only difference between these two views was that "live departure view" showed the remaining route of the service that was about to departure from the selected stop, while "service view" showed the entire route for the selected service. Therefore, in the sections below, I only describe the design and implementation process for "live service view".

Design brief

When the user views the "live service view", they would see:

- A map displaying the route of the service that the user selected, along with all the bus stops that the service will stop at. Clicking on any such stop shows the time by which the selected service will reach that stop. Live positions of the buses that belong to the selected service are also shown on the map. Clicking on such a bus shows the number of the bus and the final destination of the bus.

- The same information as above, displayed in a list form. Clicking on an item on this list takes the user to the corresponding stop on the map.

Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database models:** No new database models were created. Only existing models for `Stops`, `Services` and `Departures` tables were used.
- **Network services:** A network service named `LiveBusService` was created to encapsulate the logic for retrieving live bus locations from the relevant TFE API endpoint, and then converting them into Java objects (`LiveBus`).
- **Controllers:** A parent activity named `ServiceTimetableActivity` was created, along with a child fragment named `ServiceTimetableFragment`. The activity was responsible for letting the fragment know which service the user selected and which bus stop to start displaying the route from. While the fragment's responsibility was to lookup the `Service` and `Departure` tables to find the exact route of the selected service, and then find the times at which the selected service will arrive at each of the stops in its route. The final route and arrival times were based on the date and time at which the user made their selection. `ServiceTimetableFragment` also made use of the network service `LiveBusService` in order to display live bus locations (by polling the TFE servers every 20 seconds) along with the route of the selected service.

These components are shown in the UML class diagram in Figure 4.9.

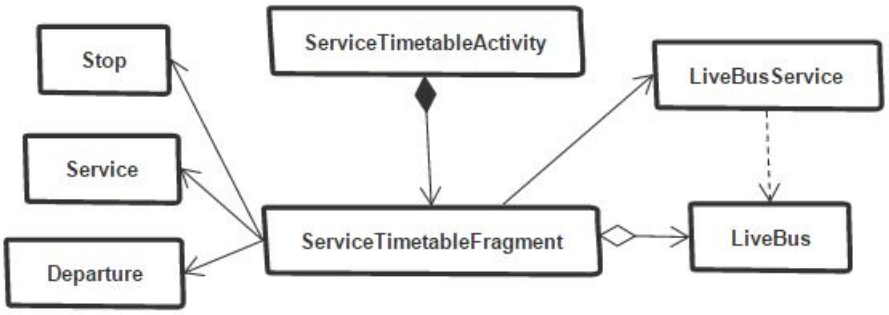

Figure 4.9: UML class diagram showing the key components needed for implementing "live service view" view

Screenshots

See Figures A.7 and A.8.

4.2.4 Increment 4

The functional requirement for this increment was as follows:

- Users should be able to add bus stops to their favourites list, and then be able to navigate to their favourites list which shows live departure times for each favourite bus stop.

As discussed in increments 1 (see Section 4.2.1) and 2 (see Section 4.2.2), the functionality to add bus stops to user's favourites list was left out for later a later increment. In this increment, the task was to design and implement the "favourites" view.

4.2.4.1 Design brief

When the user views the "favourites" view, they would see:

- A list of bus stops that they have added as a favourite. Each item in this list will have the name of the bus stop, along with some information about the upcoming buses. This information includes the name of the upcoming service, its final destination, and the amount of time remaining until departure. Clicking on an item on this list shows options to either view more departure times for the selected bus stop, or remove it from the favourites list.
- A time stamp indicating when the departure times were last updated.

4.2.4.2 Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database models:** A new `ActiveAndroid` model named `FavouriteStop` was created to define the schema of the `FavouriteStops` database table (see Section 4.1.2.5). Apart from this model, `Departure`, `Stop` and `User` models were also used.
- **Network services:** No new network service was created as the task of adding stops to the user's favourites list only concerned the local SQLite database. The existing network services `UserService` and `StopService` were used by the controller. `StopService` was updated to add the ability to retrieve upcoming departures for a list of stops provided as an argument.
- **Controllers:** A new fragment named `FavouriteStopsFragment` was created to represent the "favourites" view, and handle all the user interactions for this view. No new activity was created since I added this fragment to the existing `HomeActivity` that was created in increment 1 (see Section 4.2.1). I decided to do this in order to provide the user with single-click

navigation (via the navigation drawer pattern [12]) to all the main features of the application. Note that this also addresses the usability non-functional requirement that was identified in Section 3.3. Since a `FavouriteStop` object was nothing but a `Stop` object mapped to a `User` object, in order to retrieve all the favourite stops belonging to the user, the currently authenticated `User` instance was required. `FavouriteStopsFragment` made use of `UserService` to retrieve this instance. It also made use of `StopService` to retrieve upcoming `Departure` instances for each of the favourite stops. It also made sure that the departure times were updated every minute.

These components are shown in the UML class diagram in Figure 4.10.

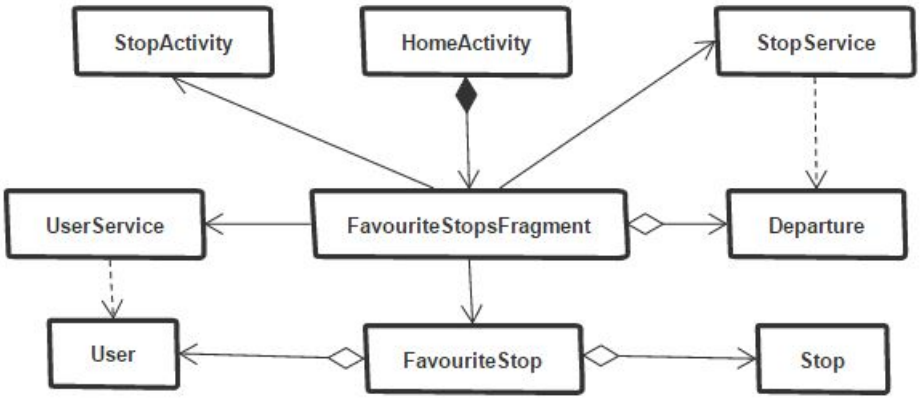

Figure 4.10: UML class diagram showing the key components needed for implementing "favourites" view

Screenshots

The navigation drawer can be seen in Figure A.26, while the "favourites" view can be seen in Figure A.9.

4.2.5 Increment 5

The functional requirement for this increment was as follows:

- Users should be able to use the application in wait-or-walk mode, which estimates whether a user who is waiting for a bus, has enough time to walk to the next stop and still have a high probability of catching the bus. Walking directions must be displayed to allow the user to successfully navigate to the next stop.

The above functional requirement was broken down into the following sequence of steps:

- Implement a "new activity" view that allows the user to choose where they are waiting, where they want to go, and which bus service they are going to use.

- Implement a "suggestions" view which, based on the choices made by the user via "new activity" view, provides them with suggestions to either wait at the current stop or walk to the next stops.

The following sections describe how each of the above steps were performed.

4.2.5.1 Step 1: Implement "new activity" view

Design brief

When the user starts up the "wait or walk" mode of the application, they would first be presented with a "new activity" view. In this view, they would be asked three questions in three different sub-views, each representing a step. The three sub-views are described below:

- **Sub-view 1 - Where are you waiting?:** The user would be presented with a list of stops near them, and would be asked to choose at which of them they are waiting for a bus. Each item in this list would have the name of the stop, the estimated walking duration from the user's current location to the stop, the services served by the stop, and the destinations of the services served by the stop. Once the selection is made, the user would also be able to see the selected stop on a map.
- **Sub-view 2 - Which service are you going to use?:** The user would be presented with a list of services served by the stop they selected in sub-view 1, and would be asked to choose the service they are waiting for. Each item in this list would have a service name and description.
- **Sub-view 3 - Where will you get off?:** The user would be presented with the all the stops in the route of the service they selected in sub-view 2, and would be asked to choose at which stop are they going to get off the bus. The first stop in the presented route would be the origin stop that was selected in sub-view 1. Once the selection is made, the user would also be able to see the selected stop on a map. Since a service can have multiple routes for the same direction, the user would also have an option to select the exact route which they are travelling.

Whenever a selection is made in any of the above sub-views, the selected item would be highlighted, and the user would be allowed to navigate to the next sub-view by selecting the "next" button. A "back" button would also be provided to navigate back to an earlier step.

Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database models:** No new database models were created. Only the existing models for `Stops` and `Services` tables were used.

- **Network services:** No network services were used since only stop and service related data was required, which was simply queried from the corresponding database tables.
- **Controllers:** A parent activity named `NewActivityActivity` was created, along with three child fragments named `SelectOriginFragment`, `SelectServiceFragment` and `SelectDestinationFragment`. These three fragments represented the three sub-views described in the design brief. `SelectOriginFragment` made use of `LocationProvider` utility class to retrieve the last known location location of the user, and then queried the `Stops` table using the `Stop` model class to retrieve the list of stops nearest to the user. `SelectServiceFragment` queried the `Services` using the `Service` model class to retrieve the list of services for the origin stop selected by the user. `SelectDestinationFragment` queried the `Services` to retrieve the route of the selected service. The parent activity had the responsibility of keeping track of the user selections in each of the sub-views, handling navigation between sub-views, and also passing the selected data from one sub-view to the next.

These components are shown in the UML class diagram in Figure 4.11

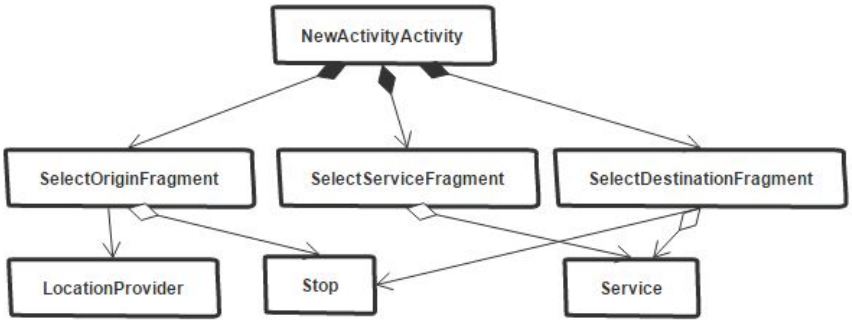

Figure 4.11: UML class diagram showing the key components needed for implementing "new activity" view

Screenshots

See Figures A.12, A.13 and A.14.

4.2.5.2 Step 2: Implement "suggestions" view

Design brief

When the user selects the "next" button from the last sub-view of "new activity" view, they would be presented with the "suggestions" view. This view would be composed of two sub-views:

- **Sub-view 1 - Suggestions:** The user would be presented with a list of suggestions. Each suggestion would indicate whether the user must wait

at the current stop, or walk to another stop by skipping a certain number of stops in between. Once a suggestion is selected, it would be highlighted and additional details would be shown in the top half of the screen. These details would include the amount of time remaining until the departure, the walking or waiting duration, and the distance to the suggested destination in kilometres.

- **Sub-view 2 - Directions:** The user would be presented with a map showing the exact walking route to the suggested destination, along with the live location of the bus that user would eventually get on. Along with the route being marked on the map, walking directions would also be provided in text form.

The user would be able to navigate between the two sub-views with the use of tabs placed at the top of the screen. Also, as soon as the user selects a suggestion, a "stop" button would appear above the tabs, that would be used to stop the "wait or walk" activity. Moreover, when the "wait or walk" activity is in progress, a notification would also appear in the notification panel. This notification would allow the user to navigate back to the "suggestions" view even when the application is not open. It would also allow them to easily see the amount of time remaining until departure from their device's lock screen, without the need to open the application.

Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Database models:** A new model class named `Activity` was created to define the schema of the `Activities` database table (see Section 4.1.2.5). Existing model classes for `Stops`, `Departures` and `Services` tables were also used.

- **Utility:** A new utility class named `ActivityLocationTrackerService` was created. It encapsulated the logic for tracking the location of the user when an activity was in progress and was also responsible for storing the tracking details in the `Activities` table. An existing utility class `LocationProvider` was also used. Note that in order to keep the power consumption low, the user's location was only tracked every 15 seconds instead of the default 1 second.

- **Network Services:** A new network service named `WaitOrWalkService` was created to encapsulate to logic of computing suggestions based on the selections made by the user in "new activity" view. The suggestions were basically computed by first finding at most five stops in the route of the service selected by the user, starting from the selected origin stop, until the selected destination stop. These stops were potentially the stops that the user could directly walk to, skipping the ones in between. Then the

estimated walking duration to each of these stops, along with their departure timetables were considered to filter out the stops that the user would not be able to reach in time. Finally, the remaining stops, along with the corresponding departure and walking route information were used to create suggestions in the form of Java objects of class `WaitOrWalkSuggestion`. Also, since for each potential stop, a separate HTTP request had to be made to find the estimated walking duration, I decided to limit the maximum number of potential stops to 5 in order to keep the computation time low.

- **Controllers:** A parent activity named `SuggestionsActivity` was created, along with two child fragments named `SuggestionsFragment` and `WalkingDirectionsFragment`. These child fragments represented the two sub-views described in the design brief. The parent activity had the responsibility of passing the user selections from "new activity" view to `SuggestionsFragment` so that it could compute the suggestions. `SuggestionsFragment` made use of `WaitOrWalkService` to compute these suggestions, and as soon as one of the suggestions was selected by the user, it fired an `EventBus` event to transfer the selected suggestion to `WalkingDirectionsFragment` so that it could display the walking directions to the suggested destination on a map.
- **Android service:** A new Android service named `CountdownNotificationService` was created. It started running as soon as a suggestion was selected by the user, and had the responsibility of displaying a notification in the Android notification panel. This notification showed the details of the selected suggestion, including a countdown for the remaining time until departure. All of this had to be implemented in an Android service because a service keeps running in the background even when the parent application is not in use. This allowed `CountdownNotificationService` to maintain a timer to keep track of the remaining time until departure without worrying about the life cycle of the parent activity. Another responsibility of this service was to let the user know if they managed to reach the destination in time or not. This was done by maintaining a geographical fence of radius 20 metres around the destination stop, and constantly checking if the user had entered the fence or not.

These key components are shown in the UML class diagram in Figure 4.12.

Screenshots

See Figures A.15, A.16 and A.17.

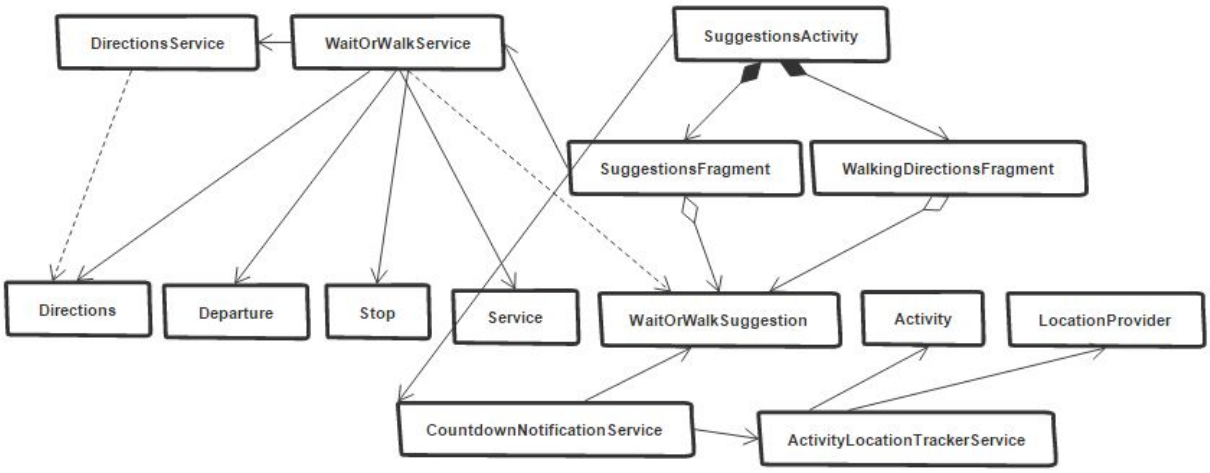

Figure 4.12: UML class diagram showing the key components needed for implementing "suggestions" view

4.2.6 Increment 6

The functional requirement for this increment was as follows:

- Users should be able to plan journeys from one location to another. The journey planner should come up with the shortest possible route to the destination, taking into account the physical activity requirements input by the user. These requirements can include walk duration, total distance covered, average speed, step and calorie counts. Walking directions must also be displayed to allow the user to successfully navigate to their destination.

The above functional requirement was broken down into the following sequence of steps:

1. Implement a "journey planner" view that allows the user to input the requirements for their journey. These requirements include the origin and destination locations, the date and time of journey, and the amount of walking needed during the journey.
2. Implement a "journey chooser" view that allows the user to select a journey from a list of all the suggested journeys.
3. Implement a "journey details" view that allows the user to view the details of a journey.

The following sections describe how each of the above steps were performed.

4.2.6.1 Step 1: Implement "journey planner" view

Design brief

When the user starts up the "journey planner" mode of the application, they would first be presented with the "journey planner" view. In this view, they

would be able to perform the following actions:

- Choose an origin and destination location by selecting the corresponding button. This would open another view that would allow the user to choose a location on the map by either simply clicking on a point on the map or by performing a text search.
- Swap origin and destination locations by selecting the "swap" button.
- Set date and time of journey by selecting the "date" button.
- Set the amount of walking to be included in the journey suggestions by selecting the "route options" button. There will be three options to choose from: "Least walking", "A lot of walking" and "Only walking".
- Submit journey requirements by selecting the "Get Directions" button.

Implementation

Implementing this view was mostly about working with different user interface elements like radio buttons, drop down menus and number pickers, and did not involve any interaction with the existing components. A parent activity named `JourneyPlannerActivity` was added, along with two child fragments named `JourneyPlannerFragment` and `PlacePickerFragment`. `JourneyPlannerFragment` was the main fragment that handled most of the user interactions for this view and kept track of all the selections made by the user, while `PlacePickerFragment` was only used when the user wanted to pick an origin or destination location on the map. It made use of the Google Places API [66] to mark nearby locations on the map and to perform text searches for nearby locations.

Screenshots

See Figures A.18, A.19 and A.20.

4.2.6.2 Step 2: Implement "journey chooser" view

Design brief

When the user selects the "Get Directions" button in the "journey planner" view, they would be presented with the "journey chooser" view. This view would display a list of all the journey suggestions that matched the requirements of the user. Each item in this list would include the summary of the journey (like Walk →Service 10 →Walk), total duration of the journey, start and end time of the journey, and the duration of walking involved.

Implementation

The following key components were created in order to achieve the specifications in the design brief:

- **Network services:** A new network service named `JourneyPlannerService` was created to encapsulate the logic for computing journey suggestions based on user requirements. It used different approaches to compute journey suggestions based on the amount of walking set by the user. If the user wanted the least amount of walking, then this service retrieved the bus journey suggestions from the relevant TFE API endpoint. However, since the returned suggestions only involved the buses that the user would have to take, I had to make use of `DirectionsService` to include walking directions from user's origin to the first bus stop in each returned suggestion, and from the last bus stop in each returned suggestion to user's destination. Due to these additions, the total journey time changed and therefore I had to filter out the journeys that no longer adhered to user's time requirements. If the user wanted "a lot of walking", then a similar process was performed, however, instead of directly working with the journey suggestions returned by the TFE API, I tried to shorten the bus journey by skipping bus stops at the start and end of each bus journey. This was done by removing one stop at a time from both ends of the bus journey, and then checking if the total journey time with increased walking still adhered to the user's time requirements or not. This bus journey shortening process was repeated until the longest walking duration was found or the bus journey had been reduced by 50%. Finally, if the user only wanted to walk to their destination, then the entire route was found by using `DirectionsService` directly. Also, since the journeys had to be later shown on a map, `JourneyPlannerService` made use of `Service` and `Stop` model classes to retrieve the exact sequence of geographical points representing the route of the bus services (only when bus journeys were involved, else, `DirectionsService` sufficed). Note that irrespective of the user requirements, all journey suggestions were represented using instances of the `Journey` class.

- **Controllers:** A parent activity named `JourneyChooserActivity` was created, along with a child fragment named `JourneyChooserFragment`. The parent activity was responsible for passing the user's requirements from "journey planner" view to the child fragment, the child fragment was responsible for retrieving journey suggestions using `JourneyPlannerService` and displaying them in the form of a list.

The above components are shown in the UML class diagram in Figure 4.13.

Screenshots

See Figure A.21.

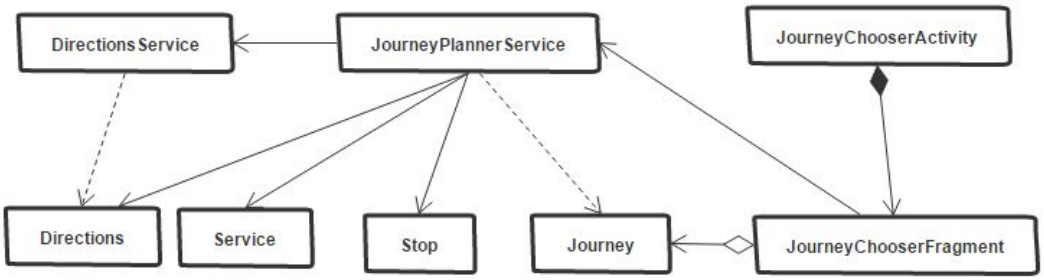

Figure 4.13: UML class diagram showing the key components needed for implementing "journey chooser" view

4.2.6.3 Step 3: Implement "journey details" view

Design brief

When the user selects a journey suggestion in the "journey chooser" view, they would be presented with the "journey details" view. This view would display the following:

- A map showing the entire journey, with different coloured markings for walking and bus legs. The map would also show the start and finish positions, along with every bus stop where the user's bus will stop.
- An ordered list of all the legs that the journey is composed of. Each item in this list would include the start time, duration, and a short textual description of the leg (eg: "Board service 23. Stay on the bus for 3 stops"). An icon would also be displayed to easily distinguish a walking leg from a bus leg.
- A "start" button at the top of the screen. Selecting this button would start a new "journey planner" activity. Moreover, when this activity is in progress, a notification would also appear in the notification panel. This notification would allow the user to navigate back to the user "journey details" view even when the application is not open. It would also allow the user to easily see the amount of time remaining until the end of the journey from their device's lock screen, without the need to open the application.

Implementation

The following key components were created in order to achieve the specifications in the design brief:

- **Android service:** A new Android service named `CountdownNotification Service` was created. Its purpose was exactly the same as the service that was created for "suggestions" view in increment 5 (see Section 4.2.5.2). The only difference was that in this case, the service was responsible for tracking a "journey planner" activity instead of a "wait or walk" activity.

- **Controllers:** A parent activity named `JourneyDetailsActivity` was created, along with a child fragment named `JourneyDetailsFragment`. The parent activity was responsible for providing the child fragment with the `Journey` instance that was selected by the user, as well as handling the life cycle of the `CountdownNotificationService` based on user interactions. The child fragment's responsibility was to display this journey in the form of a map and a list.

The above components are shown in the UML class diagram in Figure 4.14.

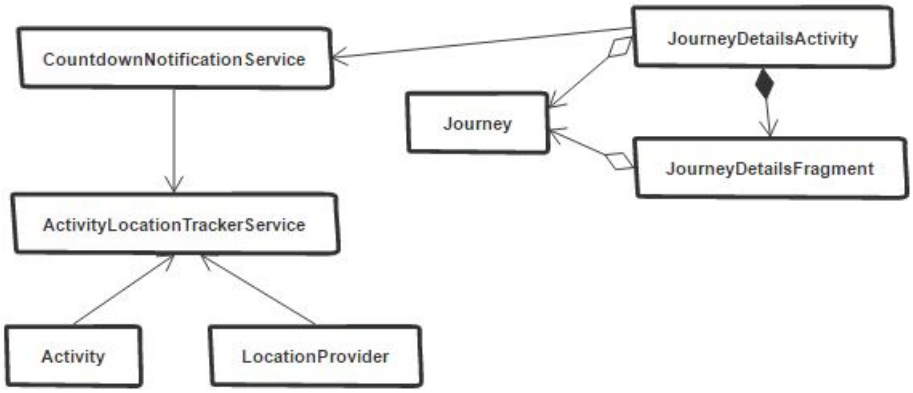

Figure 4.14: UML class diagram showing the key components needed for implementing "journey details" view

Screenshots

See Figures A.22 and A.23.

4.2.7 Increment 7

The functional requirement for this increment was as follows:

- Users should be able to view physical activity statistics of journeys they previously undertook using the application.

The above functional requirement was broken down into the following sequence of steps:

- Implement an "activity time-line" view to display all the activities performed by the user as a chronological list.
- Implement an "activity detail" view to display the physical activity statistics of a specific activity.

The following sections describe how each of the above steps were performed.

4.2.7.1 Step 1: Implement "activity time-line"

Design brief

When the user views the "activity time-line" view, they would see:

- The current day's physical activity statistics. These would include distance travelled, calories burned, steps taken and average speed.
- A list of activities performed to date. Each item in this list would include the names of the start and end locations, the type of activity ("Wait or Walk" or "Journey Planner"), and the activity start time in 24-hour format. By default, this list would be grouped by day, but an option to group by week or month would be provided using a drop-down menu. Also, a physical activity statistic would be displayed for each group. By default, this would be distance, but an option to change it to calories, steps or duration, would be provided via a drop-down menu.

Implementation

The following key components were created in order to achieve the specifications in the design brief:

- **Design models:** No new model classes were added. Existing database model classes for `Activities` and `Users` tables were used.
- **Controllers:** A new child fragment named `ActivityFragment` was created and added to the existing activity named `HomeActivity`. The reasons for reusing this activity were discussed in Section 4.2.4.2. This child fragment represented the "activity time-line" view and handled all user interactions for this view. It first retrieved the currently authenticated user instance using `UserService`, and then retrieved all the activities corresponding to this user by querying the `Activities` table using the `Activity` model class. In order to display the activities in the required format, it grouped first them and computed the average physical activity statistic based on the options selected by the user.

The above components are shown in the UML class diagram in Figure 4.15.

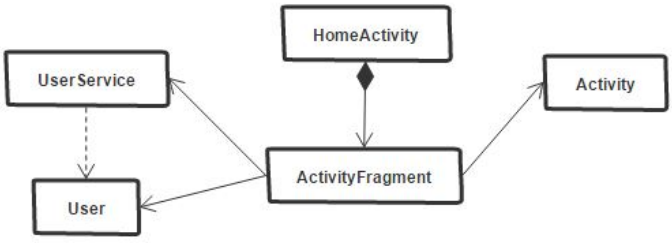

Figure 4.15: UML class diagram showing the key components needed for implementing "activity time-line" view

Screenshots

See Figure A.24.

4.2.7.2 Step 2: Implement "activity detail" view

Design brief

When the user selects a list item on "activity time-line" view, they would be presented with the "activity detail" view for the selected activity. The user would see the following:

- A map displaying the path taken by the user while they were performing the activity.
- A button to delete the activity from the database.
- The name of the start and finish locations, along with the type of activity and the date and time it was performed at.
- A list of physical activity statistics, including activity duration, distance, average speed, steps and calories.

Implementation

The implementation of this view involved minimal use of existing components, that is, all the logic was mostly self-contained. A new parent activity named `ActivityDetailActivity` was created, along with a child fragment named `Activity DetailFragment`.The role of the parent activity was to let the child fragment know which activity was selected by the user, and the role of the child fragment was to retrieve the selected activity from the database (using `Activity` model), and then display its details. The path of the user was marked on a map, while the physical statistics were calculated and displayed in the form of a list.

Screenshots

See Figure A.25.

4.2.8 Increment 8

The functional requirement for this increment was as follows:

- Users should be automatically notified whenever there is a disruption in any of the bus routes.

The above functional requirement was broken down into the following sequence of steps:

1. Implement a "disruptions" view that allows users to view the most recent service disruptions.
2. Enable the application to send notifications to the user whenever there is a service disruption.

The following sections describe how each of the above steps were performed.

4.2.8.1 Step 1: Implement "disruptions" view

Design brief

When the user views the "disruptions" view, they would see:

- A list of service disruptions, where each item has a type (planned or incidental) and cause of disruption, date of occurrence, and the names of the services affected by the disruption. Clicking on such an item opens the Lothian Buses website, with more information on the disruption.

Implementation

The following key components were created in order to meet the specifications in the design brief:

- **Network service:** A network service named `DisruptionsService` was created. It encapsulated the logic for retrieving disruptions related information from the relevant TFE API endpoint, and converting the JSON response into a Java object (`Disruption`).
- **Controllers:** A new fragment named `DisruptionsFragment` was created to represent the "disruptions" view, and handle all the user interactions for this view. This fragment was added to the existing activity called `HomeActivity` for reasons discussed in Section 4.2.4.2. `DisruprtionsFragment` made use of `DisruptionsService` to retrieve the disruptions and display the appropriate information in the form of a list.

These components are shown in the UML class diagram in Figure 4.16.

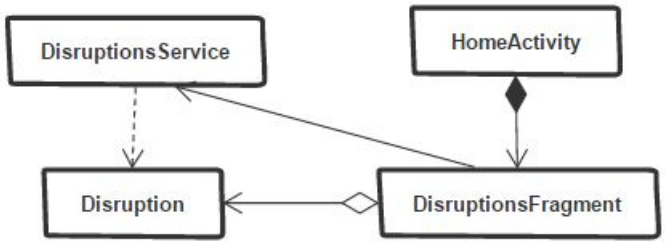

Figure 4.16: UML class diagram showing the key components needed for implementing "disruptions" view

Screenshots

See Figure A.10.

4.2.8.2 Step 2: Enable the client-side application to send disruption notifications to user

Design brief

When the user gets a notification, they would see the icon of the application appear in the status bar. When the notification panel is opened, the user would see a notification for a recent service disruption. Each notification will have two visual styles: "expanded" and "collapsed". When collapsed, the notification will show the disruption type, the time when the notification was received, and the list of services affected. When expanded, the notification will show the disruption type, the time when the notification was received, and the summary of the disruption.

Implementation

In order to meet the specifications in the design brief, a new broadcast receiver named `DisruptionAlarmReceiver` was created. [3] This component essentially allowed me to respond to events even when the mobile device wasn't in use. Within the `Application` class named `App` (the main entry point of the Android application) (see Section 4.1.2.4), I set up an instance of `AlarmManager` [68] (a class that allows scheduling the execution of application code) to fire custom events named `ACTION_DISRUPTION_ALARM` every hour. Since `DisruptionAlarmReceiver` was registered for this event, every time this event was fired, it used `DisruptionsService` to check if there are any new service disruptions. For every new disruption found, it created an Android notification, displaying the relevant information about the corresponding disruption. The way it checked if the user had already been notified of a disruption was by keeping track of the past disruptions (in an XML file provided by Android). So every time the `ACTION_DISRUPTION_ALARM` event was fired, it retrieved the list of previously notified disruptions via `PreferencesManager`, and then simply compared them with the list of newly fetched disruptions. Any disruptions that were not in the previously notified disruptions list, were used to create notifications.

The reason why I decided to use `AlarmManager` to fire events in the background instead of an Android Service (described in Section 4.1.2.2), was because `AlarmManager` executed the event-firing code only when it was scheduled to run (every hour), while an Android service would've kept the device awake at all times, making the application consume considerably more battery. This decision also helped me address the efficiency non-functional requirement (see Section 3.3).

[3] "A broadcast receiver is an Android component which allows you to register for system or application events. All registered receivers for an event are notified by the Android runtime once this event happens." [67]

The UML class diagram in Figure 4.17 shows the components described above.

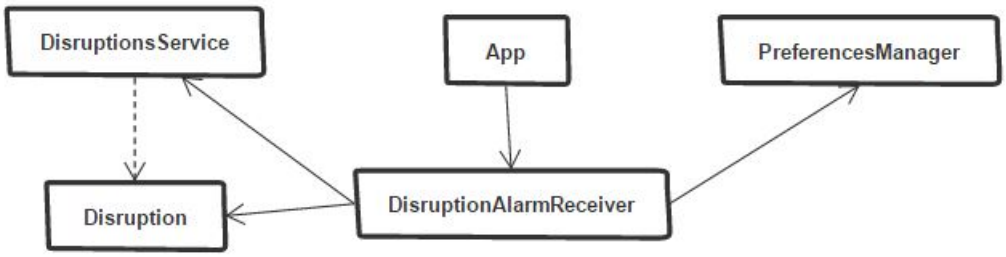

Figure 4.17: UML class diagram showing the key components needed for enabling disruption notifications

Screenshots

See Figure A.11.

Chapter 5

Testing

"Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test." [69] Since I was using the incremental model as my development methodology, software testing was performed just before the delivery of each increment (as seen in Figure 2.1). In this chapter, I describe the different types of tests that were performed on both applications, server-side and client-side.

5.1 Server-side

The role of the server-side application was to expose REST API endpoints that the client-side application could use to authenticate its users' credentials. A combination of unit and integration tests were performed. Unit tests test individual units of code, while integration tests test a group of components that are combined together to produce an output. To be more specific, the following types of tests were performed:

- Tests to check if all database models are functioning correctly by manually adding data to them and then checking if on retrieval, they have the same data that was added initially. That is, to check if all CRUD (Create, Read, Update and Delete) operations are functioning properly.
- Tests to check form validation by ensuring that the correct error messages are returned if the provided input does not adhere to the requirements. For example, an appropriate error message should be returned if a user is trying to create an account with a password that contains special characters other than underscore.
- Authentication tests to make sure every endpoint can only be accessed by users who are actually authorized to access them. For example, one user should not allowed to change the data belonging to another user.

In order to perform the above tests, I had to first choose a testing framework since

Express Web Framework, being a minimalist web development framework, did not include one. I decided to use Mocha.js [70], the most popular testing framework for Node.js applications. It provided me with a very intuitive interface for testing asynchronous JavaScript code. Here is a code snippet showing the basic structure of a Mocha test suite:

```
describe("Model"), function () {
    before(function (done)) {
        // set up testing enviroment
    };
    describe("User", function ()) {
        it("should return error when trying to save duplicate username",
          function (done)) {
            // the code for testing the above behaviour goes here
        }
    }
    after(function (done)) {
        // tear down testing environment
    }
}
```

In the above code snippet, `describe()` is used to group related tests together, while `it()` is used to define individual tests within the groups. `before()` and `after()` are used to set up and tear down the testing environment before and after running the tests within a group. This feature allowed me to set up a mock database with different initial data for the different groups of tests that were dependent on the database. As seen in the above code snippet, I used Behaviour Driven Development (BDD) style descriptions for my test cases. BDD style descriptions are essentially English-language sentences that express the behaviour and the expected outcomes of a test case. Due to the fact that they are human-readable and easy to understand, they can even be shared with non-programmers when discussing test requirements. To further enhance the readability of the tests, I used another library called should.js [71], which allowed me to match my assertion code with the BDD style test-case descriptions. Here is another code snippet showing the easy-to-read assertion interface provided should.js:

```
// BDD style assertion: User should have 'Manas Bajaj' as the name and
// their age should be within 10 and 25 years.
user.should.have.property("name", "Manas Bajaj");
user.age.should.be.within(10, 25);
```

In all, the test suite for testing the database models and all endpoints of the authentication API contained 33 comprehensive test cases (see Figure A.27), which passed with a statement coverage of **89%**. Statement coverage is a metric which is used to calculate and measure the number of statements in the source code which have been executed. [72].

5.2 Client-side

For the client-side application, the majority of the testing was done manually. That is, I played the role of the end-user and tested each application feature to identify unexpected behaviour. To ensure completeness of testing, I used the design briefs created during the design phase of the development life cycle (see Chapter 4) as a reference when testing each application view separately. This way I was able to make sure that I did not skip any implemented functionality.

Since Android devices come in a lot of different shapes and sizes, with varying performance levels and operating system versions (see report on Android fragmentation [10]), I had to perform compatibility testing to ensure that my application was able to support a large range of devices so that it could reach a large audience. I achieved this by setting up multiple custom Android virtual devices with varying screen resolutions and OS versions (all versions ranging from the minimum version identified in Section 4.1.2.1 to the most recent version of that time) using an Android emulator called Genymotion [73] (see Figure 5.1).

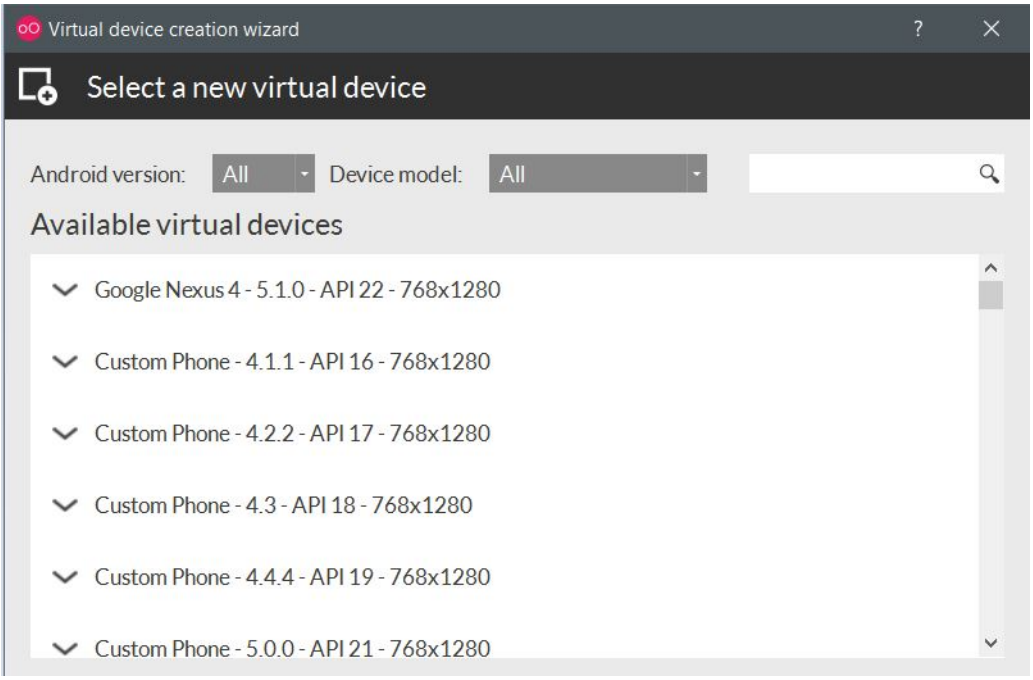

Figure 5.1: Genymotion's virtual device creation wizard

Another form of manual testing that was performed was usability testing. This was performed by the end-users of the application to ensure that the application's user interface was intuitive and easy-to-use. No major problems were identified in usability testing since I made sure that I followed the official user-interface design guidelines provided by Google [11] when designing the layout for each view.

Lastly, automated database tests using were also performed to ensure that CRUD operations on all models worked correctly. A Java test framework called `JUnit` [74] was used to do so.

Chapter 6

Evaluation

After all the increments had been delivered, I moved onto the final stage of the development life cycle - Evaluation, the process of measuring the effectiveness of the system and finding potential enhancements [75]. Instead of carrying out evaluation by sending questionnaires and application installation files to each one of the evaluators separately, I decided to take a different approach. I published my application on Google Play Store [76], and restricted access to the published application to only the set of assigned evaluators by adding them to a beta testing group via Google Play Developer Console [77]. The main advantages of distributing the pre-release version of my application using this approach were as follows:

- Users had no difficulties in installing the application since they were already aware of the installation process via the official application store. On the other hand, if the application was sent as an Android Application Package (APK) file [78], some devices would have refused to install it due to security concerns.
- Whenever I published an updated version of the application, all users were automatically notified of the update.
- Users were be able to provide me with critical feedback without affecting the application's public reviews and ratings.
- Google Play Developer Console allowed me to view detailed reports on any crashes that occurred on the users' devices. Additional information such as the Android operating system version and device model name for all users was also available.

For the creation and distribution of evaluation questionnaires, I made use of Google Forms [79], a tool for collecting information from users via online surveys. The main advantage of using this over the traditional approach was that Google Forms automatically collected all user responses and illustrated them with the use of graphs, making it easier for me to analyse them.

The final questionnaire consisted mostly of multiple choice questions in which I made statements about certain features of the application, and the user had to

choose whether they “strongly agree”, “agree”, “disagree”, or “strongly disagree” with my statements. In Section 6.1, I list the different questions that I asked along with the responses I received, and in Section 6.2, I analyse the responses and react to the received feedback.

6.1 Questions

In this Section, I only list the different questions that were asked as part of the evaluation questionnaire, along with the corresponding answers. The responses are then analysed in Section 6.2.

6.1.1 Question 1: Login/Sign up Process

The statements and the corresponding responses are shown in Figure 6.1.

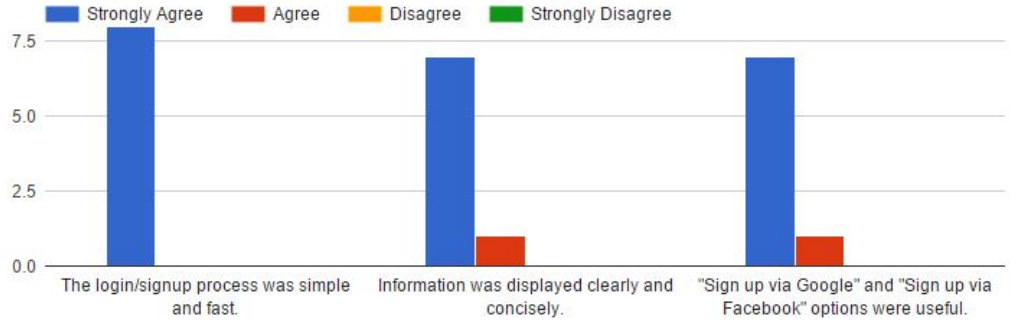

Figure 6.1: Responses for Question 1

6.1.2 Question 2: ‘Nearby Bus Stops’ View

The statements and the corresponding responses are shown in Figure 6.2.

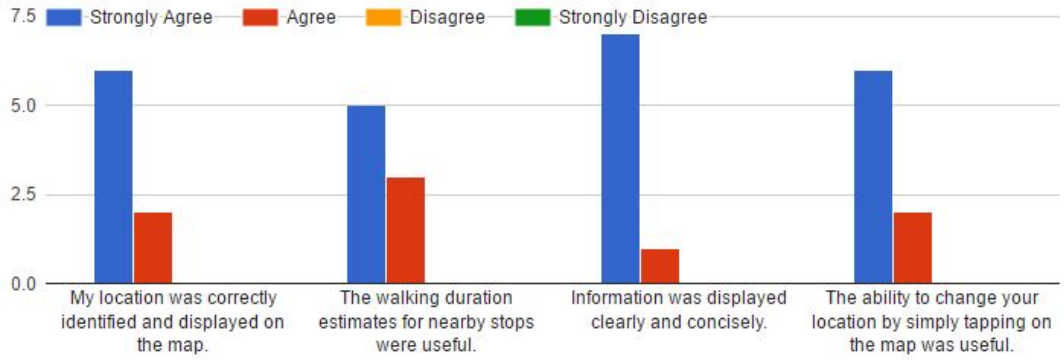

Figure 6.2: Responses for Question 2

6.1.3 Question 3: 'Search for Stops and Services' View

The statements and the corresponding responses are shown in Figure 6.3.

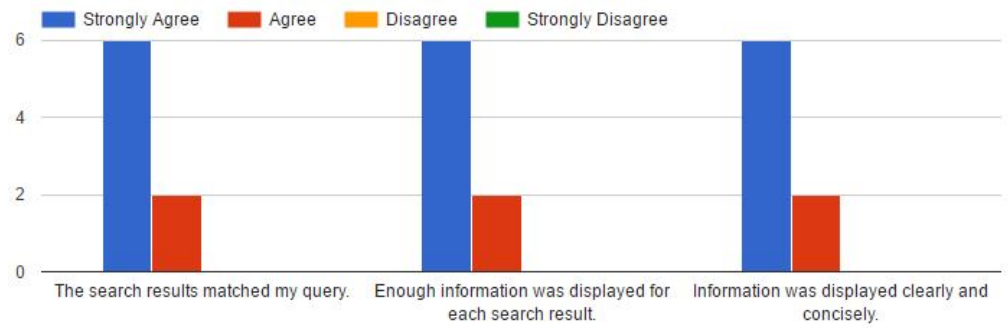

Figure 6.3: Responses for Question 3

6.1.4 Question 4: 'Favourite Bus Stops' View

The statements and the corresponding responses are shown in Figure 6.4.

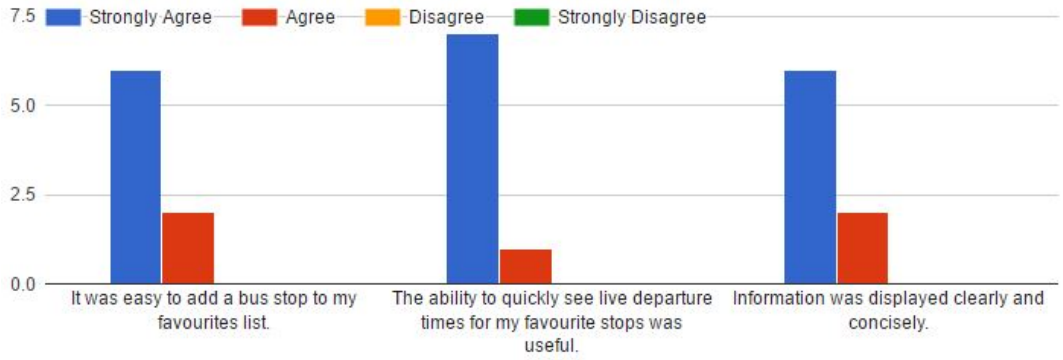

Figure 6.4: Responses for Question 4

6.1.5 Question 5: 'Wait-or-Walk' View

The statements and the corresponding responses are shown in Figure 6.5.

6.1.6 Question 6: 'Journey Planner' View

The statements and the corresponding responses are shown in Figure 6.6.

6.1.7 Question 7: 'Activity' View

The statements and the corresponding responses are shown in Figure 6.7.

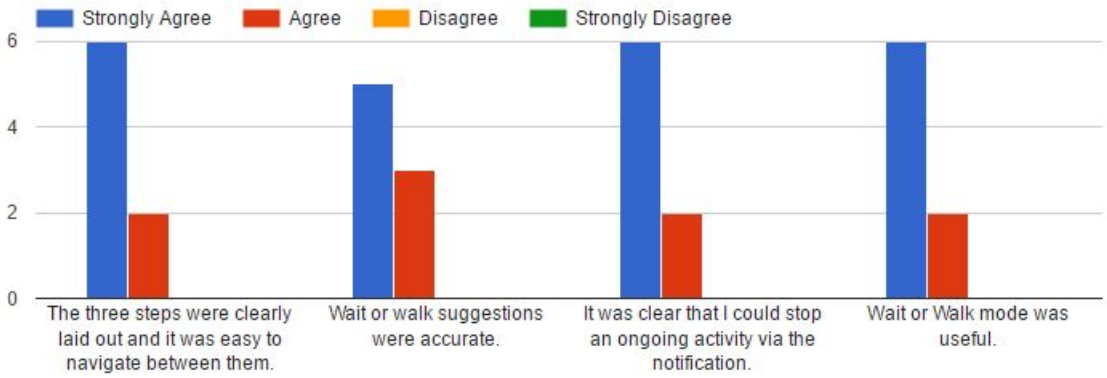

Figure 6.5: Responses for Question 5

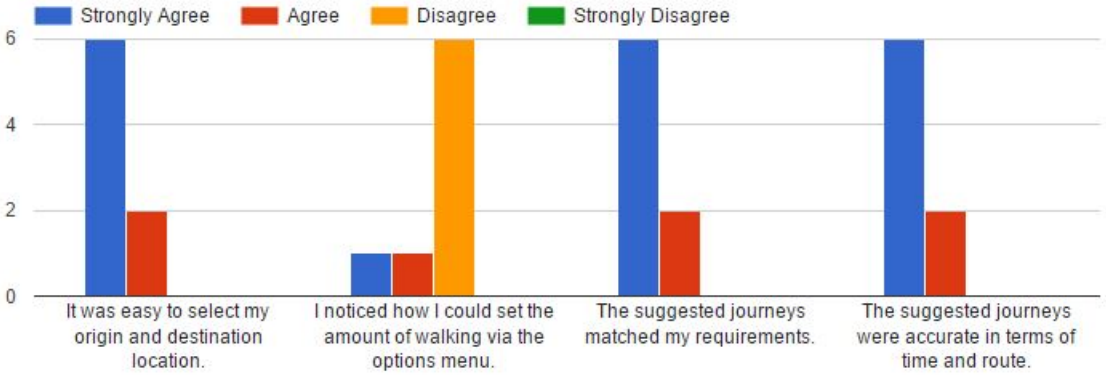

Figure 6.6: Responses for Question 6

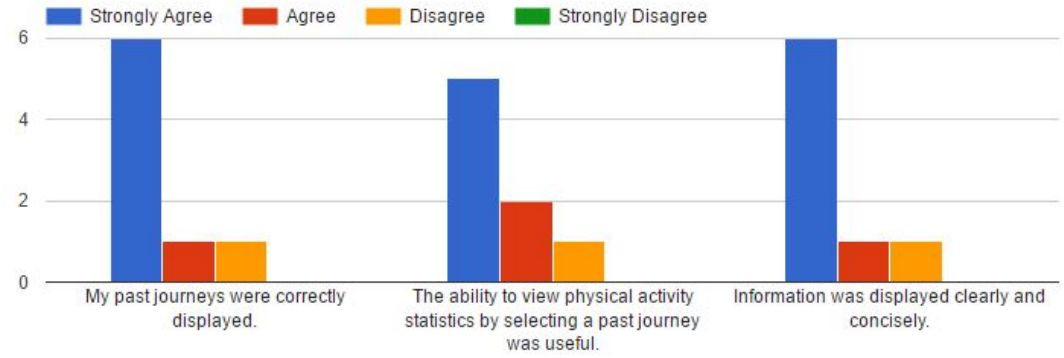

Figure 6.7: Responses for Question 7

6.1.8 Question 8: 'Disruptions' View

The statements and the corresponding responses are shown in Figure 6.8.

6.1.9 Question 9: General Feedback

The statements and the corresponding responses are shown in Figure 6.9.

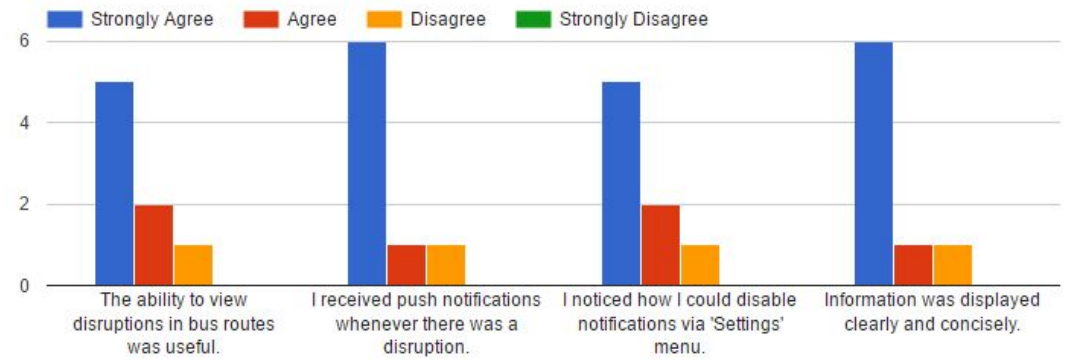

Figure 6.8: Responses for Question 8

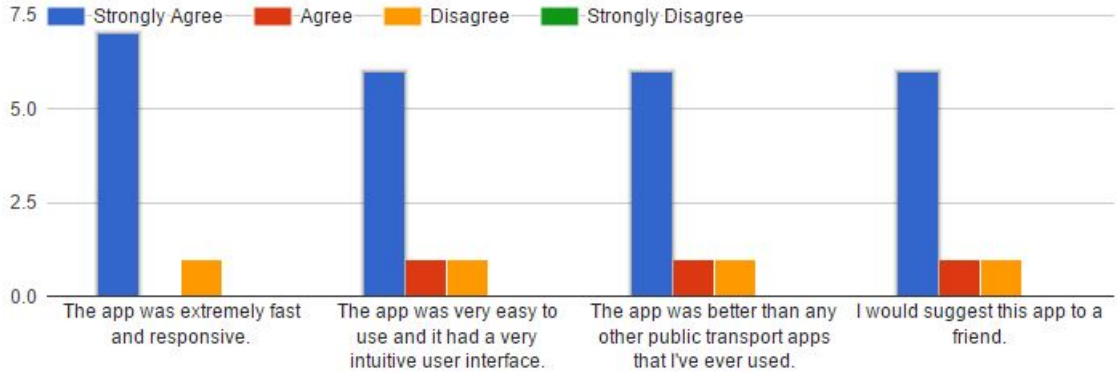

Figure 6.9: Responses for Question 9

6.1.10 Question 10: Content Navigation

This question asked the users for their preference on the way they like to navigate application content. The following two options were provided, along with screenshots showing the corresponding navigation pattern:

- *Sliding navigation drawer menu* (Option 1)
- *Tabular navigation* (Option 2)

The responses are shown in Figure 6.10.

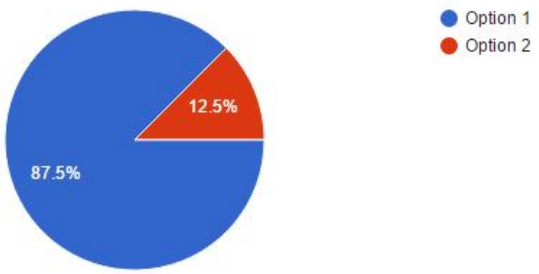

Figure 6.10: Responses for Question 10

6.1.11 Question 11: Suggestions for Improvement

This was the only non-multiple choice question as it asked the users to suggest improvements in the form of text. Note that this was also the only optional question.

The responses are shown in Figure 6.11.

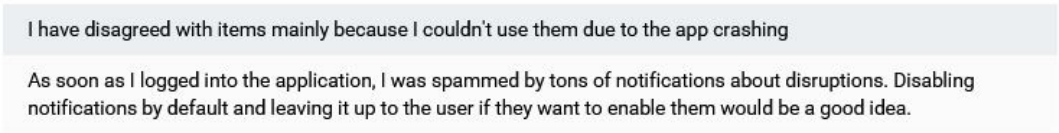

Figure 6.11: Responses for Question 11

6.2 Response Analysis

Overall, the feedback I received was excellent. This was probably because I knew exactly what the users wanted from such an application before I even started developing it (see Chapter 3) and that I was able to provide the end-users with a seamless user experience by ensuring that I always followed the best practices for Android development. Although no new functionality was requested, some minor concerns were raised by the evaluators about the existing functionality. The sections below describe how these were addressed.

6.2.1 Constant application crashes

One of the evaluators stated that they had to "disagree" with a lot of statements in the above questions since the application kept crashing and therefore they were not able to use them (see Figure 6.11). I got back to them and discovered that their device was running the latest version of Android (Android Marshmallow [80]), which hadn't even been released back when I started developing the application. In order to reproduce the issue, I set up a Genymotion virtual device (see Section 5.2) running Android version 6.0 (Marshmallow) and noticed that the application kept crashing right after successfully going through the login process. Since no meaningful error messages were being logged in the debug console, I referred to the release notes of Android Lollipop, and quickly realized that a new "permission model" [81] was introduced, which allowed users to manage application permissions at runtime. Since my application was heavily reliant on accessing the user's location, and given that all permissions were disabled by default, this resulted in my application throwing a runtime exception every time the operating system refused to provide it with the user's location. In order to address this issue, I had to check if the required location access permission had been granted at runtime before attempting to access the location. If it hadn't

been granted then I had to request for it at runtime. This was done using the new API methods `checkSelfPermission()` and `requestPermissions()`.

6.2.2 Unnoticeable route options

As seen in Figure 6.6, a lot of evaluators found it hard to notice that the walking requirement could be set via the options menu. To address this issue, I made use of the accent colour of the application to make the options button more noticeable. The button label was also changed from 'options' to 'route options'.

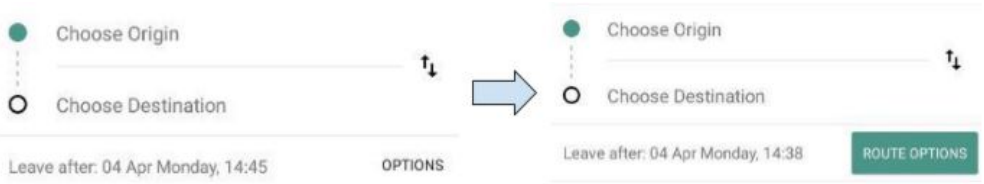

Figure 6.12: Making options button more noticeable

6.2.3 Disruption notifications at start-up

One of the evaluators suggested to disable disruption notifications by default and leave it up to the user to enable them in case they require that feature. I considered this to be a great suggestion as not everyone would require constant disruption notifications, and therefore decided to disable them by default.

Chapter 7

Conclusion

I successfully achieved the goal of this project, which was to create a public transport mobile application that takes users from one place to another, while keeping them fit. The final application was successfully evaluated and was very well-received.

Using the incremental development methodology that was chosen at the start of this project (see Chapter 2), the development life cycle consisted of dividing all the requirements gathered from the stakeholders into multiple increments. Each increment then went through the phases of design, implementation and testing in order to meet all the functional and non-functional requirements. Although the functional requirements were met in an orderly fashion, the non-functional requirements were addressed throughout the development life cycle. Therefore, here is a summary of how I addressed each non-functional requirement identified during the requirement engineering phase:

- **Availability:** Since the client-side application was made responsible for retrieving and storing transport related data from the TFE API, basic features like viewing nearby bus stops and timetables were able to function even when the user was offline (see Section 4.2.1.6).
- **Security:**
  - All user data stored on the client-side or server-side required authentication to access (see Section 4.2.1.3).
  - User passwords were hashed using a slow hash function called bcrypt before they were stored in the server-side database (see Section 4.2.1.3).
  - The use of token-based authentication obviated the need to protect against Cross Site Request Forgery (CSRF) (see Section 4.2.1.3).
  - All information passed between the server and the clients was encrypted before transmission as the authentication server's inbound and outbound traffic was limited to HTTPS (see Section 4.2.1.4).
  - While authenticating users via external services, the confused deputy

problem was prevented by verifying the authenticity of access tokens (see Section 4.2.1.3).

- **Compatibility:**
  - The target platform for the client-side application was chosen to be Android, which had the highest market share at the time the decision was made (see Section 4.1.2.1).
  - The minimum Android version supported by the client-side application allowed it be run on 85% of all existing Android devices, and at the same time allowed it to make use of the latest Android APIs (see Section 4.1.2.1).
  - Compatibility testing was performed to ensure that the client-side application was able to support devices with varying performance levels and screen sizes (see Section 5.2).
- **Usability:** Client-side application user interface was designed using the Material Design guidelines provided by Google. This ensured that new users were able to quickly and easily learn how to use the application as they were already familiar with the different design patterns being used. Note that this was also confirmed during the evaluation phase as most users strongly agreed to the statement *"The app was very easy to use and it had a very intuitive user interface"* (see Section 6.1.9).
- **Efficiency:**
  - One of the reasons why REST was chosen to be the design model for the server side API was that it was able to make efficient use of the bandwidth as there was less data to transfer (compared to SOAP) (see Section 4.2.1.1).
  - The use of `AlarmManager` over Anroid Service to periodically check for bus service disruptions in the background, ensured that the device was not kept awake at all times, and therefore saved battery life 4.2.8.2.
  - Whenever an activity was in progress, the user's location was only tracked once every 15 seconds instead of the default of once every second to keep the power consumption low (see Section 4.2.5.2).
- **Performance:**
  - Due to the decision to store transport related data on a local client-side database, the loading times for displaying such data were considerably reduced as the retrieval process only involved database querying (see Section 4.2.1.6).
  - Each frequently queried database field was set to be an index in order to improve query performance (see Section 4.1.1.5).
  - It was decided not to normalize the data stored in the client-side relational database to make data retrieval faster (by requiring fewer

queries) (see Section 4.2.1.6).

- The use of REST design model for the API resulted in faster client-server communication (see Section 4.2.1.1).
- The use of token-based authentication obviated the need for checking if a user's session exists in the database, resulting in faster authentication of credentials (see Section 4.2.1.3).
- The decision to persist currently authenticated user's details in the client-side database obviated the need to authenticate user's credentials every time the application was started, improving the application start-up time (see Section 4.2.1.5).
- The use of the difference between geographical coordinates instead of the exact haversine distance considerably reduced the time to find the bus stops nearest to the user's location (see Section 4.2.1.6).
- Database and network operations were always performed on a background thread so that the main thread was never interrupted when drawing the user interface, keeping the application responsive at all times.
- Whenever lists of items had to be displayed in the client-side application, I ensured that the view recycling technique was used so that lists could be scrolled without any stutter or lag.

- **Scalability:**
  - The decision to choose Node.js as the server-side web development framework was based on the fact that it could support a lot more concurrent connections compared to traditional servers (see Section 4.1.1.1).
  - The use of stateless token-based authentication over cookie-based authentication ensured that horizontal scaling could be performed with ease as it would not require user requests to be sent to the same server every time (see Section 4.2.1.3).
  - The decision to host use Amazon EC2 as my web hosting service was based on the fact that its web interface allowed easy scaling of the server-side application, both horizontally (by increasing the memory/disk capacity of any server instance) and vertically (by booting new server instances) (see Section 4.2.1.4).
- **Documentation:** The design briefs created during the design phase of the development life cycle were all combined together in a single document to produce user documentation.

In Section 7.1, I list some additional features that could be added to this project in the future to encourage more people to use it.

7.1 Future Work

One of the main reasons why most people do not like to walk is because they think of it as a mundane task. In order to make it more interesting, the following functionality could be incorporated into this project:

- A system of user ranking and achievements based on statistics like the the number of calories burned, number of steps taken, average walking speed, etc. This would increase the participation of sedentary users by motivating them to compete with other users.
- An updated journey planner that would suggest scenic walking routes which include popular tourist attractions, parks, etc.
- A walk-sharing mode (similar to ride-sharing applications like Lyft [82] and Uber [83]) that would allow users to specify a walking journey and a time window, and then match them with other users with similar requirements. This would make walking feel more like an enjoyable social occasion rather than a chore. Moreover, this would also help people feel more comfortable in certain situations like walking in unfamiliar areas at night.

Bibliography

[1] Transport for Edinburgh API. https://tfe-opendata.readme.io/. [Online; accessed 06-March-2016].

[2] Public transport, walking and cycling to work are all associated with reductions in body fat for adults in mid-life. https://www.sciencedaily.com/releases/2016/03/160316215131.htm. [Online; accessed 29-March-2016].

[3] Health benefits. http://www.c3health.org/wp-content/uploads/2009/09/C3-report-on-walking-v-1-20120911.pdf. [Online; accessed 29-March-2016].

[4] Android Operating System. https://www.android.com/intl/en_uk/. [Online; accessed 06-March-2016].

[5] Edinburgh Bus Tracker Android Application. https://play.google.com/store/apps/details?id=org.redbus&hl=en_GB. [Online; accessed 29-March-2016].

[6] Transport for Edinburgh Android Application. https://play.google.com/store/apps/details?id=com.lothianbuses.lothianbuses&hl=en_GB. [Online; accessed 29-March-2016].

[7] Android Froyo. http://developer.android.com/about/versions/android-2.2.html. [Online; accessed 29-March-2016].

[8] Android Lollipop. http://developer.android.com/about/versions/android-5.0.html. [Online; accessed 29-March-2016].

[9] ListView view recycling. http://lucasr.org/2012/04/05/performance-tips-for-androids-listview/. [Online; accessed 29-March-2016].

[10] Report on Android Fragmentation. http://opensignal.com/reports/2015/08/android-fragmentation/. [Online; accessed 08-March-2016].

[11] Android Material Design Guidelines. https://www.google.com/design/spec/material-design/introduction.html#. [Online; accessed 28-March-2016].

[12] Navigation Drawer Pattern. https://www.google.com/design/spec/patterns/navigation-drawer.html#. [Online; accessed 26-March-2016].

[13] Centers for Medicare and Medicaid Services. Selecting a development approach. https://www.cms.gov/research-statistics-data-and-systems/cms-information-technology/xlc/downloads/selectingdevelopmentapproach.pdf, 2008.

[14] Dr. Winston W. Royce. Managing the development of large software systems. http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf, 1970.

[15] Roger S. Pressman. *Software Engineering: A Practitioner's Approach (Seventh edition)*. 2010. pp. 41-42.

[16] Helen Sharp, Anthony Finkelstein, and Galal Galal. Stakeholder identification in the requirements engineering process. http://discovery.ucl.ac.uk/744/1/1.7_stake.pdf, 1999.

[17] Django Web Framework. https://www.djangoproject.com/. [Online; accessed 16-March-2016].

[18] Express Web Framework. http://expressjs.com/. [Online; accessed 16-March-2016].

[19] Object/Relational Mapping. http://hibernate.org/orm/what-is-an-orm/. [Online; accessed 16-March-2016].

[20] Node.js. https://nodejs.org/en/. [Online; accessed 16-March-2016].

[21] V8 Engine. https://developers.google.com/v8/. [Online; accessed 16-March-2016].

[22] Node Package Manager. https://www.npmjs.com/. [Online; accessed 16-March-2016].

[23] JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Online; accessed 20-March-2016].

[24] Node.js module. https://nodejs.org/api/modules.html. [Online; accessed 20-March-2016].

[25] Communication endpoint. https://en.wikipedia.org/wiki/Communication_endpoint. [Online; accessed 20-March-2016].

[26] Express middleware. http://expressjs.com/en/guide/writing-middleware.html. [Online; accessed 20-March-2016].

[27] Event-driven programming paradigm. https://en.wikipedia.org/wiki/Event-driven_programming. [Online; accessed 20-March-2016].

[28] Steve Burbeck. Application programming in smalltalk-80: How to use model-view-controller (mvc). http://www.math.sfedu.ru/smalltalk/gui/mvc.pdf, 1992.

[29] Event-driven programming paradigm. `https://github.com/caolan/async"`. [Online; accessed 20-March-2016].

[30] Event-driven programming paradigm. `https://github.com/caolan/async#waterfall"`. [Online; accessed 20-March-2016].

[31] Mongoose. `http://mongoosejs.com/`. [Online; accessed 22-March-2016].

[32] IDC mobile operating system market share. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. [Online; accessed 19-March-2016].

[33] Apple iOS. `http://www.apple.com/uk/ios/what-is/`. [Online; accessed 19-March-2016].

[34] IDC mobile operating system market share. `http://www.geeksquad.co.uk/articles/what-is-project-butter-in-android-jelly-bean-4-1`. [Online; accessed 19-March-2016].

[35] Communicating with other fragments. `http://developer.android.com/training/basics/fragments/communicating.html`. [Online; accessed 22-March-2016].

[36] EventBus. `https://greenrobot.github.io/EventBus/`. [Online; accessed 22-March-2016].

[37] GSON. `https://github.com/google/gson`. [Online; accessed 22-March-2016].

[38] AsyncTask. `http://developer.android.com/reference/android/os/AsyncTask.html`. [Online; accessed 23-March-2016].

[39] AsyncJob. `https://github.com/Arasthel/AsyncJobLibrary`. [Online; accessed 23-March-2016].

[40] Active Android ORM. `http://www.activeandroid.com/`. [Online; accessed 28-March-2016].

[41] Java annoitation. `https://en.wikipedia.org/wiki/Java_annotation`. [Online; accessed 28-March-2016].

[42] Simple Object Access Protocol. `https://www.w3.org/TR/2000/NOTE-SOAP-20000508/`. [Online; accessed 16-March-2016].

[43] Roy Thomas Fielding. Representational state transfer. `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`, 2000.

[44] Hypertext Transfer Protocol. `https://www.w3.org/Protocols/rfc2616/rfc2616.txt`. [Online; accessed 16-March-2016].

[45] Extensible Markup Language. `https://www.w3.org/XML/`. [Online; accessed 16-March-2016].

[46] JavaScript Object Notation. `http://www.json.org/`. [Online; accessed 16-March-2016].

[47] MongoDB. https://www.mongodb.org/. [Online; accessed 22-March-2016].

[48] MIME Type. https://tools.ietf.org/html/rfc2045. [Online; accessed 16-March-2016].

[49] HTTP Cookie. https://en.wikipedia.org/wiki/HTTP_cookie. [Online; accessed 16-March-2016].

[50] Sessions. https://en.wikipedia.org/wiki/Session_(computer_science). [Online; accessed 16-March-2016].

[51] HTTP Authorization Header. https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html. [section 14.8; Online; accessed 16-March-2016].

[52] Bcrypt. https://en.wikipedia.org/wiki/Bcrypt. [Online; accessed 28-March-2016].

[53] Why use bcrypt? https://codahale.com/how-to-safely-store-a-password/. [Online; accessed 28-March-2016].

[54] JSON Web Token. https://tools.ietf.org/html/rfc7519. [Online; accessed 17-March-2016].

[55] Cross Site Request Forgery. https://en.wikipedia.org/wiki/Cross-site_request_forgery. [Online; accessed 17-March-2016].

[56] Software Development Kit. https://en.wikipedia.org/wiki/Software_development_kit. [Online; accessed 19-March-2016].

[57] OAuth. http://oauth.net/. [Online; accessed 19-March-2016].

[58] Confused deputy problem. https://en.wikipedia.org/wiki/Confused_deputy_problem. [Online; accessed 19-March-2016].

[59] Amazon Elastic Compute Cloud. https://aws.amazon.com/ec2/. [Online; accessed 19-March-2016].

[60] HTTP Secure. https://tools.ietf.org/html/rfc2818. [Online; accessed 19-March-2016].

[61] SQLite database. https://www.sqlite.org/about.html. [Online; accessed 23-March-2016].

[62] UML class diagram. https://en.wikipedia.org/wiki/Class_diagram. [Online; accessed 23-March-2016].

[63] Haversine formula. https://en.wikipedia.org/wiki/Haversine_formula. [Online; accessed 25-March-2016].

[64] Reverse Geocding. https://en.wikipedia.org/wiki/Reverse_geocoding. [Online; accessed 25-March-2016].

[65] Google Maps Directions API. https://developers.google.com/maps/documentation/directions/. [Online; accessed 28-March-2016].

[66] Google Places API. https://developers.google.com/places/. [Online; accessed 27-March-2016].

[67] Broadcast Receiver. http://www.vogella.com/tutorials/AndroidBroadcastReceiver/article.html#broadcastreceiver. [Online; accessed 26-March-2016].

[68] AlarmManager. http://developer.android.com/reference/android/app/AlarmManager.html. [Online; accessed 26-March-2016].

[69] Software Testing. http://www.kaner.com/pdfs/ETatQAI.pdf. [Online; accessed 28-March-2016].

[70] Mocha.js test framework. https://mochajs.org/. [Online; accessed 28-March-2016].

[71] should.js assertion library. https://github.com/shouldjs/should.js. [Online; accessed 28-March-2016].

[72] Statement coverage. http://istqbexamcertification.com/what-is-statement-coverage-advantages-and-disadvantages/. [Online; accessed 28-March-2016].

[73] Genymotion emulator. https://www.genymotion.com/. [Online; accessed 28-March-2016].

[74] JUnit test framework. http://junit.org/junit4/. [Online; accessed 28-March-2016].

[75] Evaluation phase. https://en.wikipedia.org/wiki/Systems_development_life_cycle#Evaluation. [Online; accessed 30-March-2016].

[76] Google Play Store. https://play.google.com/store. [Online; accessed 30-March-2016].

[77] Google Play Developer Console. http://developer.android.com/distribute/googleplay/developer-console.html. [Online; accessed 30-March-2016].

[78] Android Application Package (APK). https://en.wikipedia.org/wiki/Android_application_package. [Online; accessed 30-March-2016].

[79] Google Forms. https://www.google.co.uk/forms/about/. [Online; accessed 30-March-2016].

[80] Android Marshmallow. http://developer.android.com/about/versions/marshmallow/android-6.0.html. [Online; accessed 29-March-2016].

[81] New permission model in Android Marshmallow. http://developer.android.com/about/versions/marshmallow/android-6.0-changes.html. [Online; accessed 30-March-2016].

[82] Lyft. `https://www.lyft.com/`. [Online; accessed 30-March-2016].

[83] Uber. `https://www.uber.com/`. [Online; accessed 30-March-2016].

Appendices

Appendix A

Screenshots

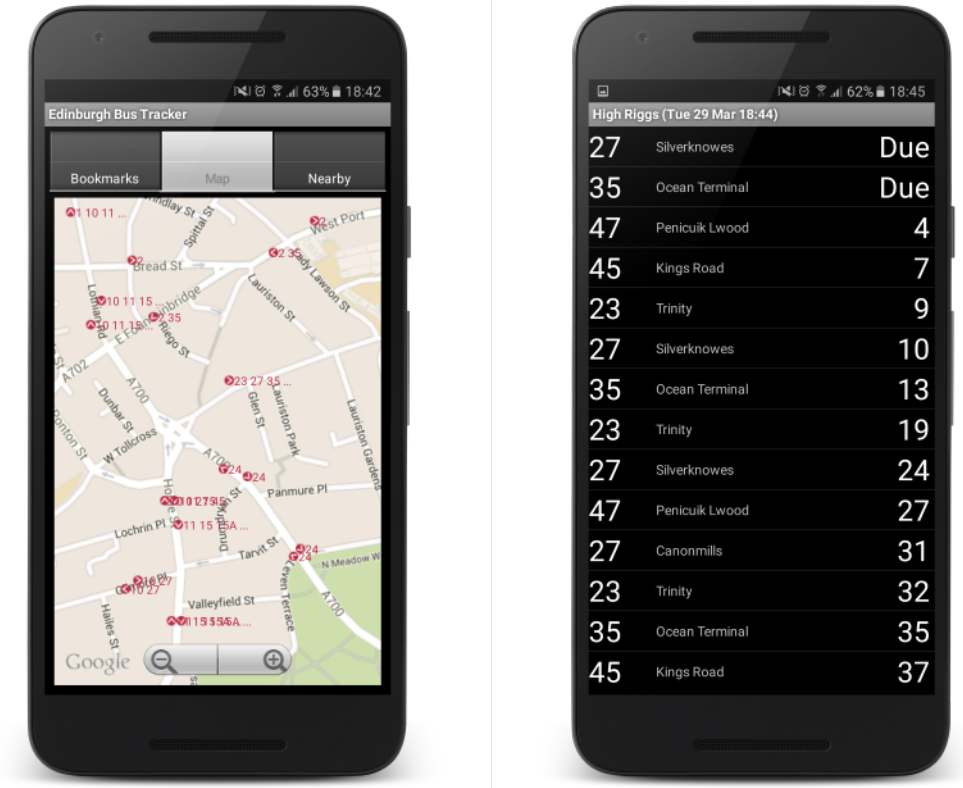

Figure A.1: Screenshots for Edinburgh Bus Tracker Android Application

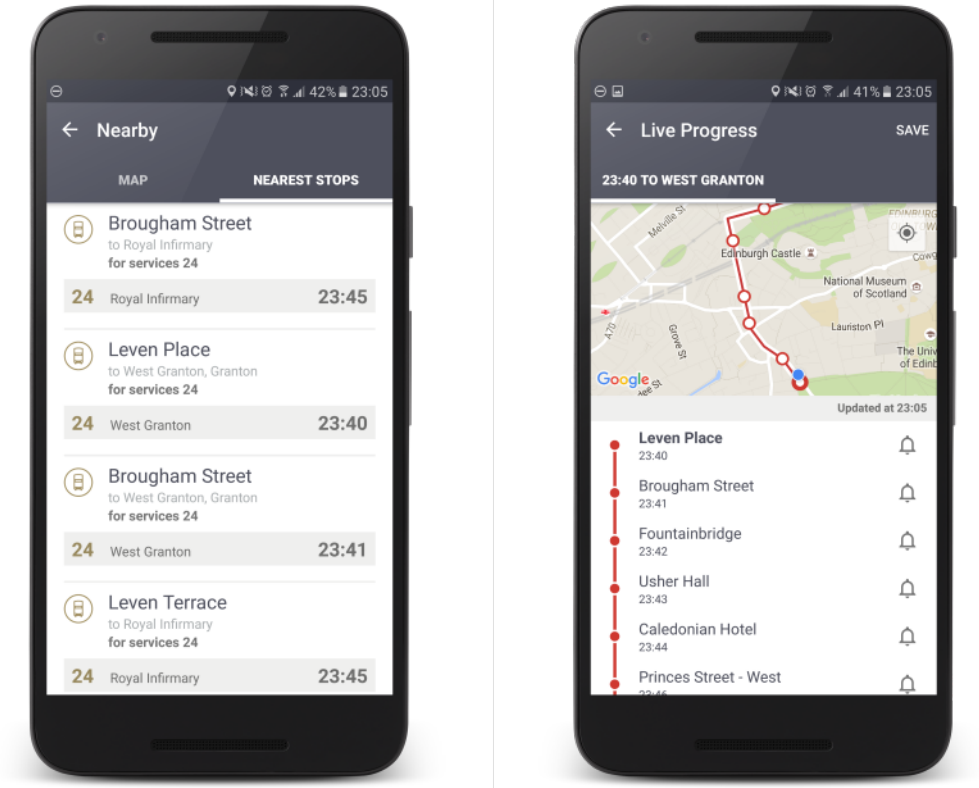

Figure A.2: Screenshots for Transport for Edinburgh Android Application

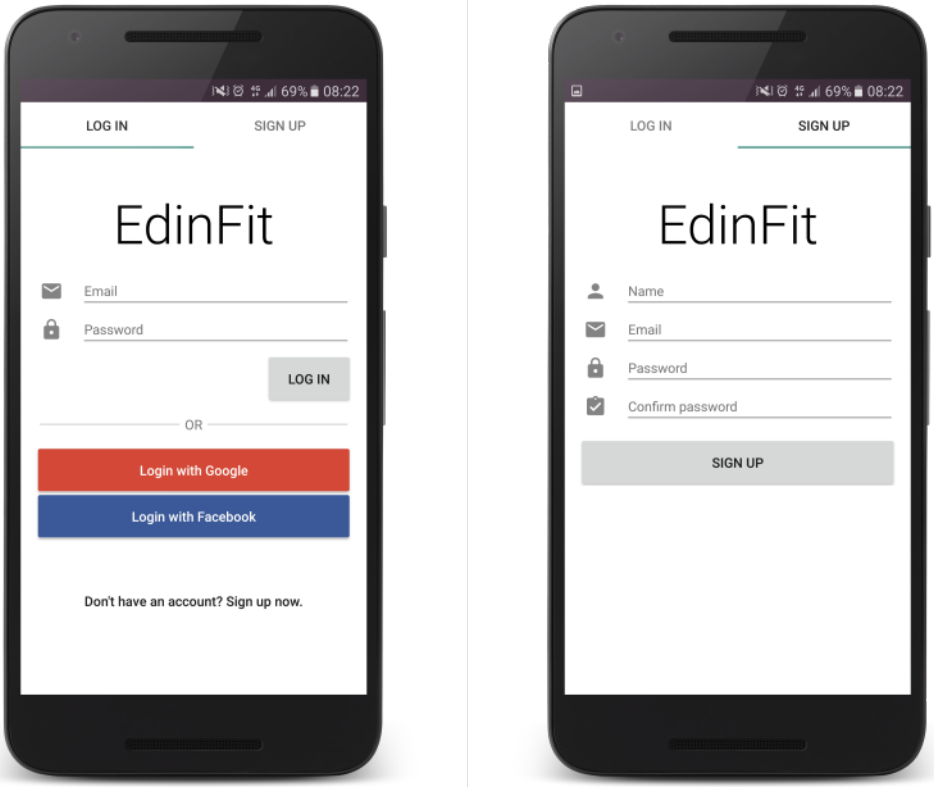

Figure A.3: Screenshots showing login sub-view (left) and sign up sub-view (right)

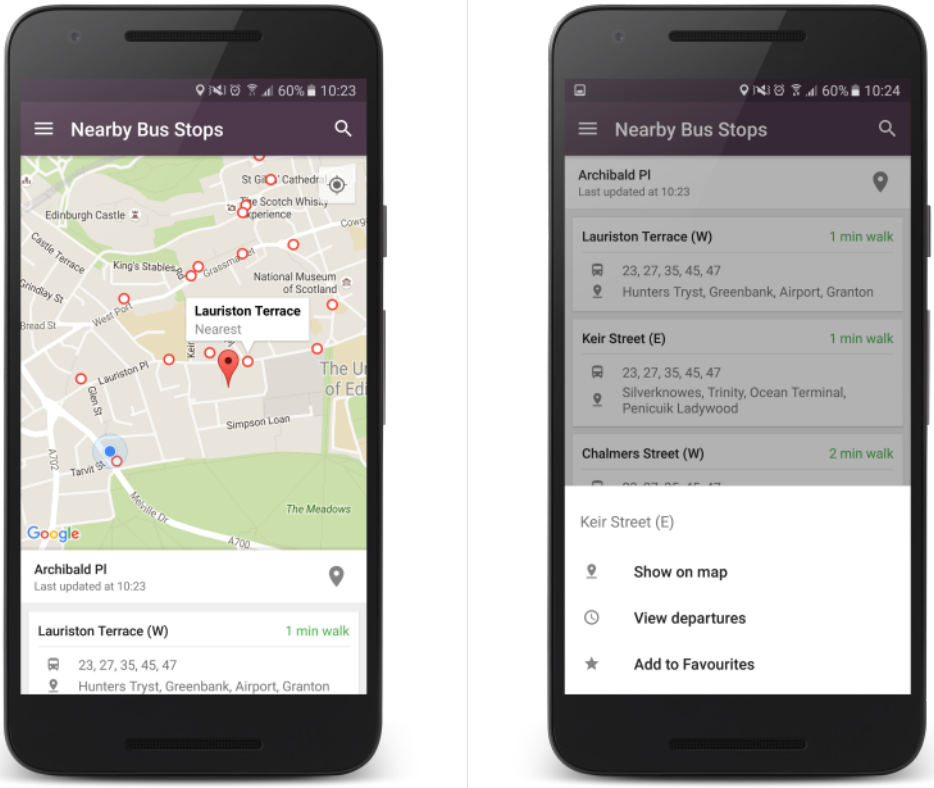

Figure A.4: Screenshots showing "nearby bus stops" view

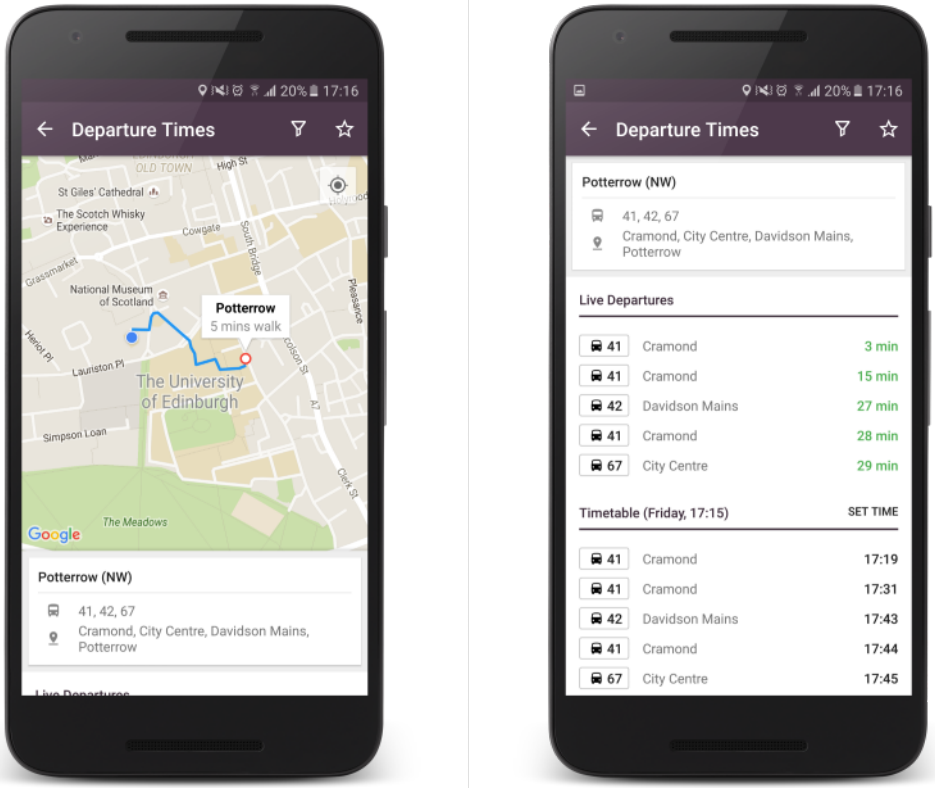

Figure A.5: Screenshots showing "departure times" view

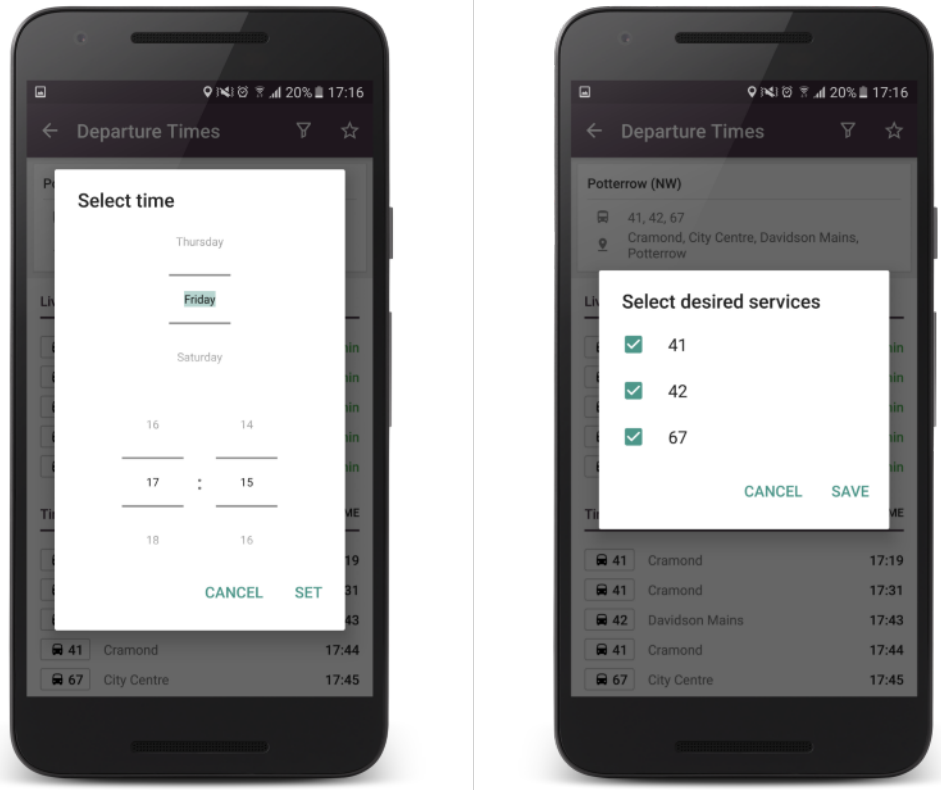

Figure A.6: Screenshots showing "departure times" view

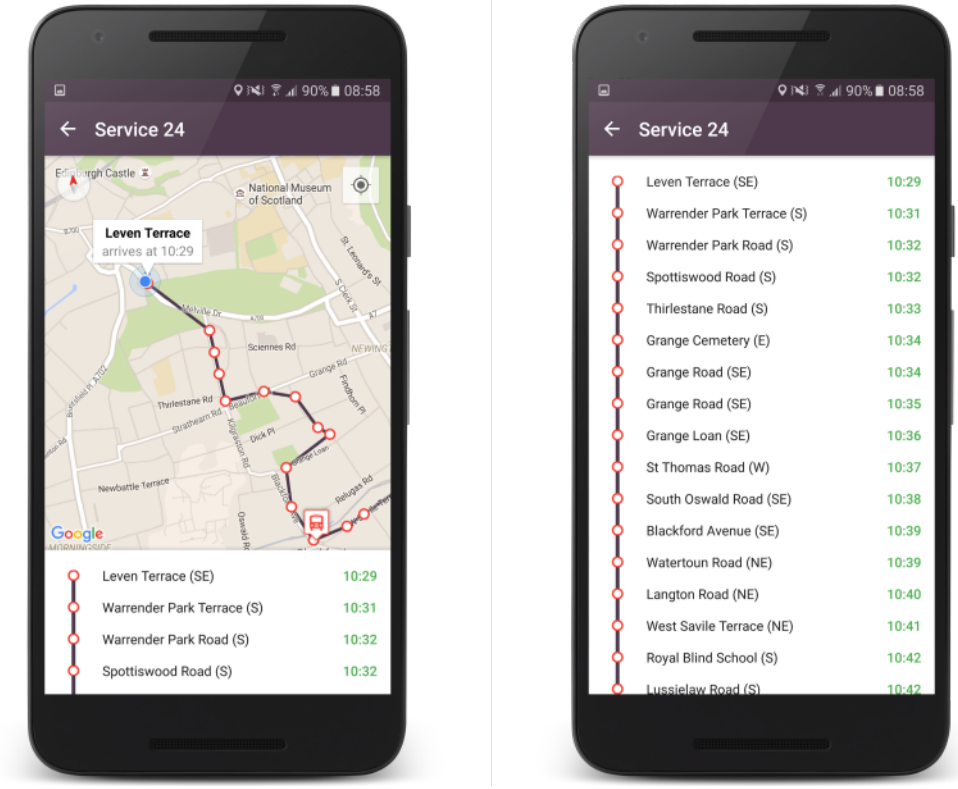

Figure A.7: Screenshots showing "live service" view

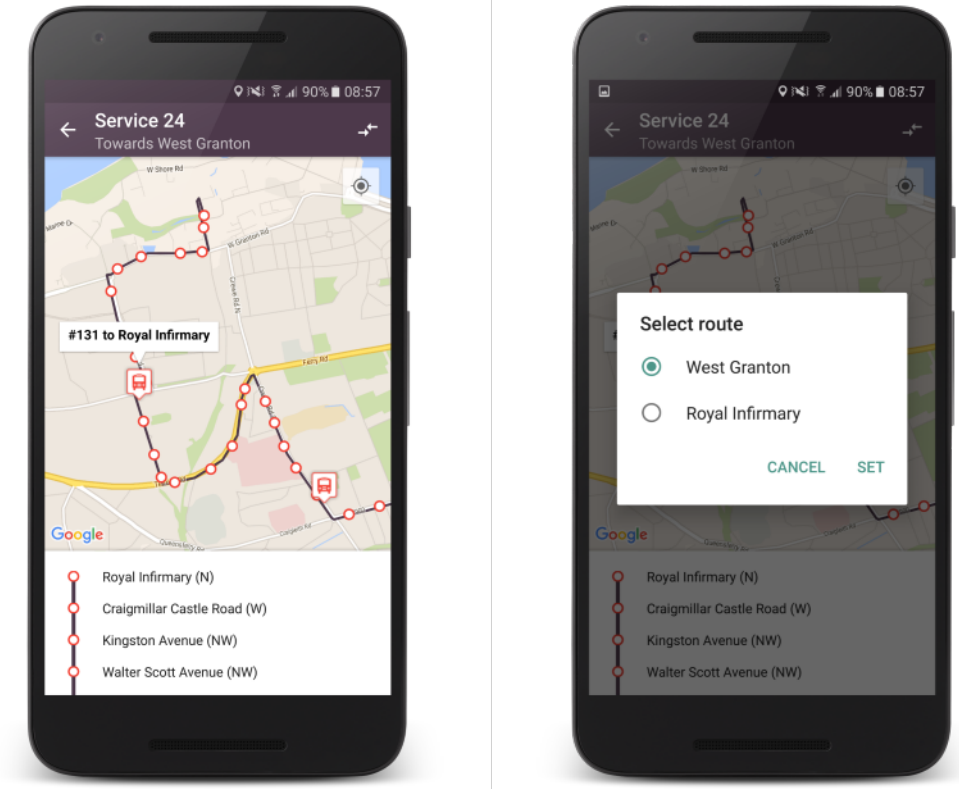

Figure A.8: Screenshots showing "service" view

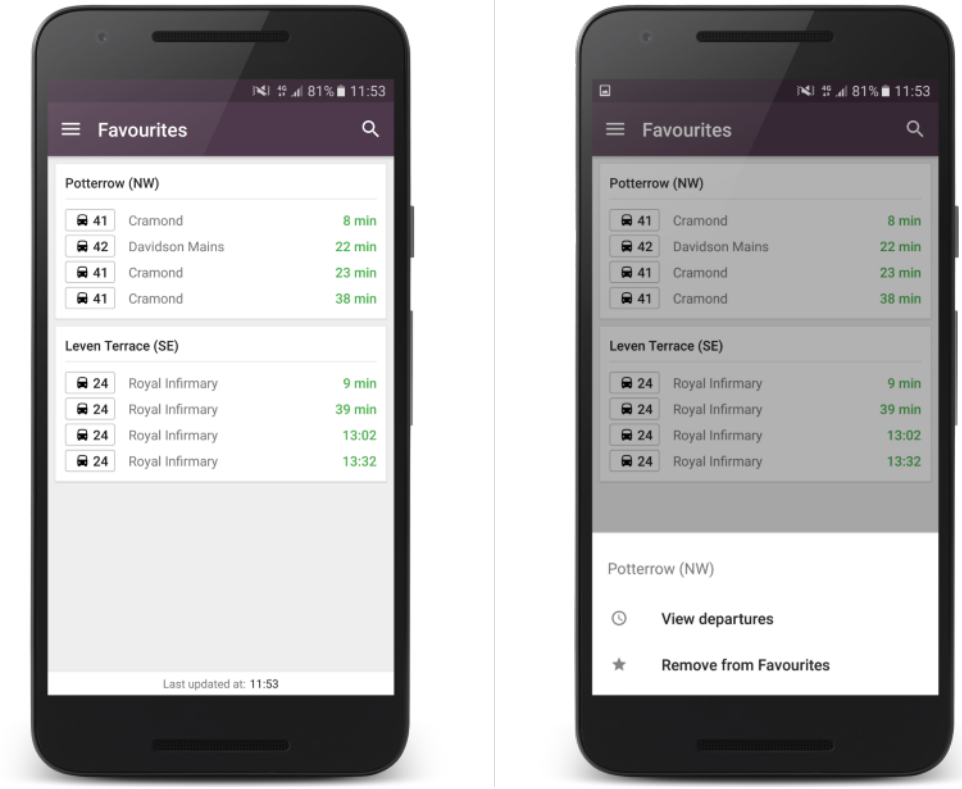

Figure A.9: Screenshots showing "favourites" view

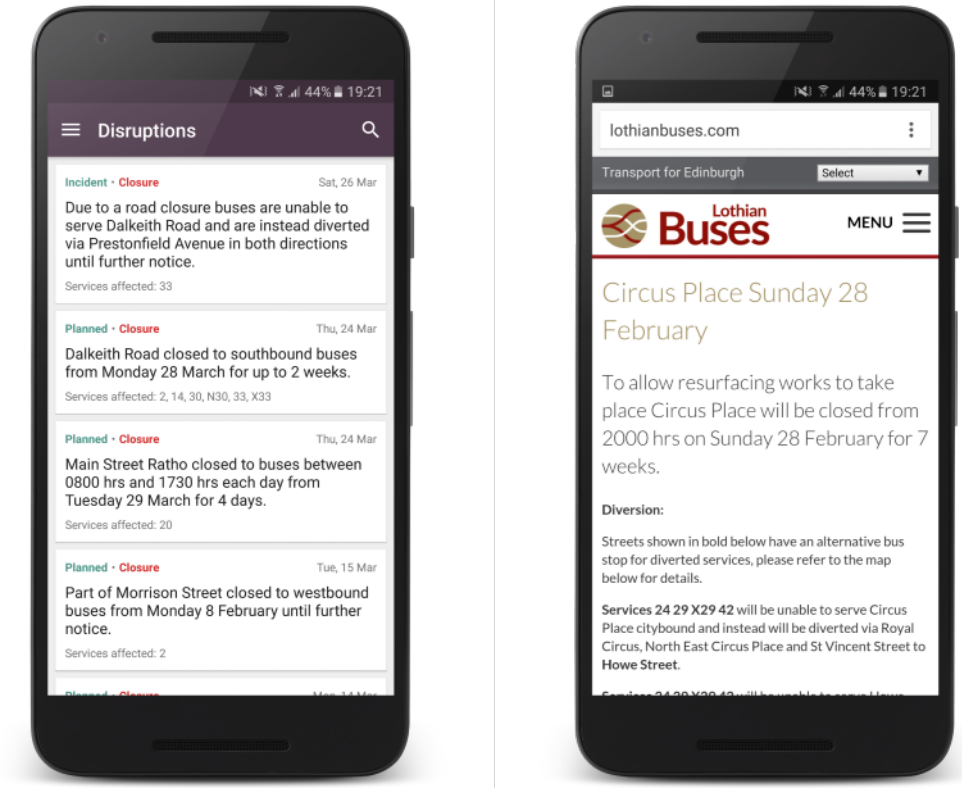

Figure A.10: Screenshots showing "disruptions" view [left] and a browser window [right] which opens when a disruption is selected

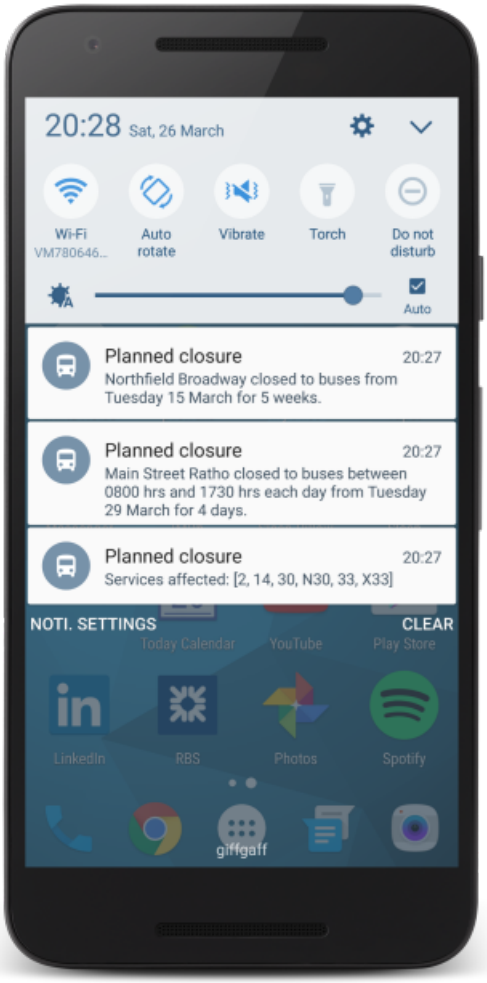

Figure A.11: Screenshot showing expanded (first 2) and collapsed disruption notifications.

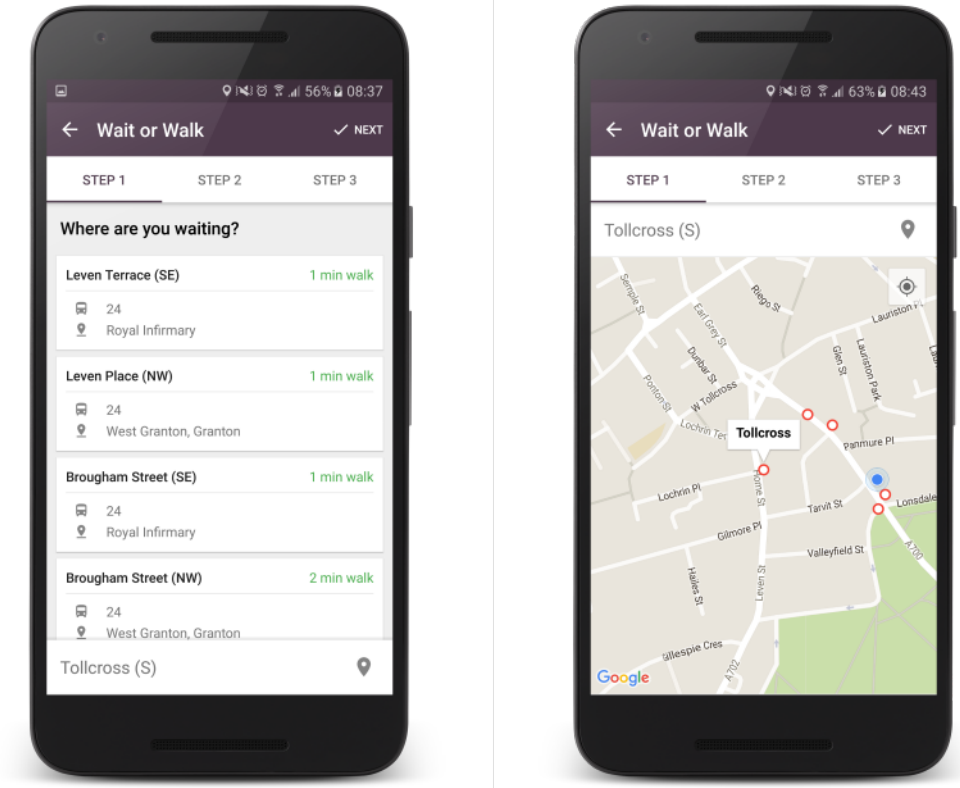

Figure A.12: Screenshots showing Step 1 sub-view of “new activity” view

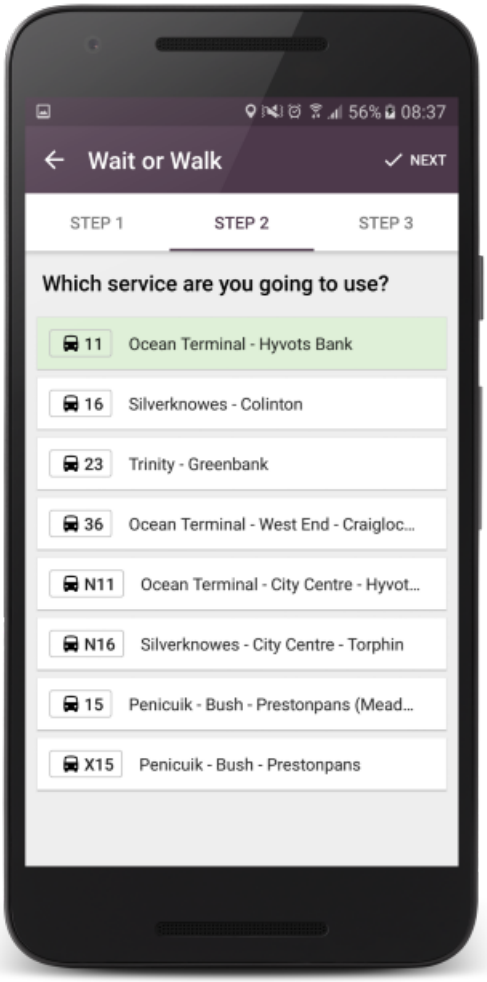

Figure A.13: Screenshot showing Step 2 sub-view of “new activity” view

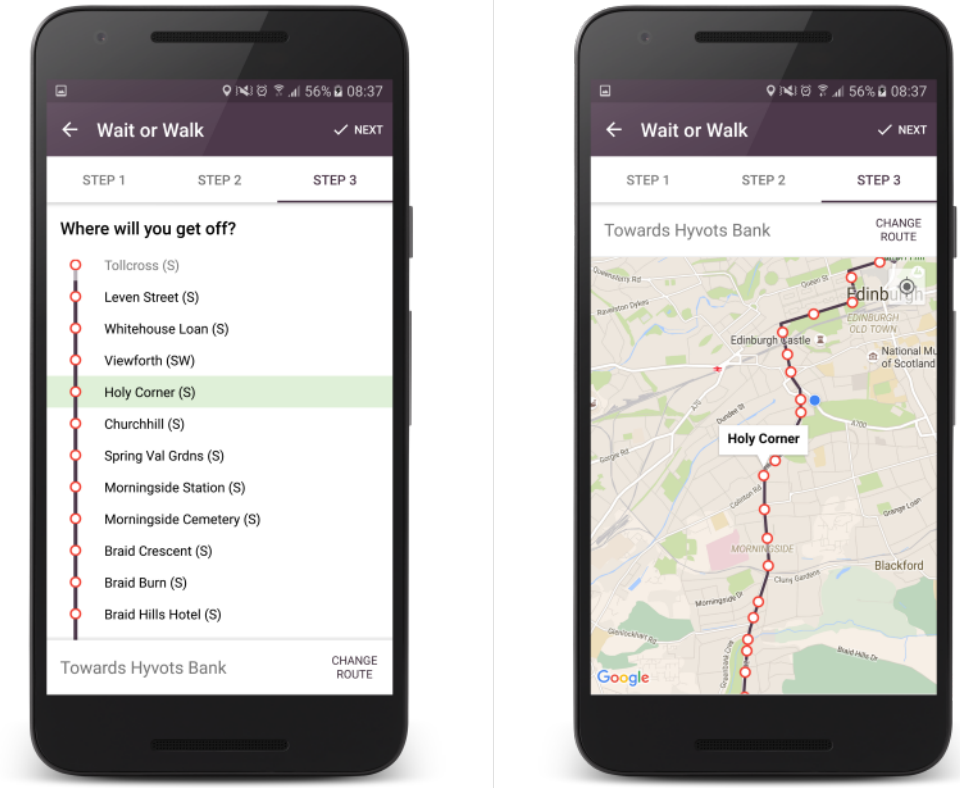

Figure A.14: Screenshots showing Step 3 sub-view of "new activity" view

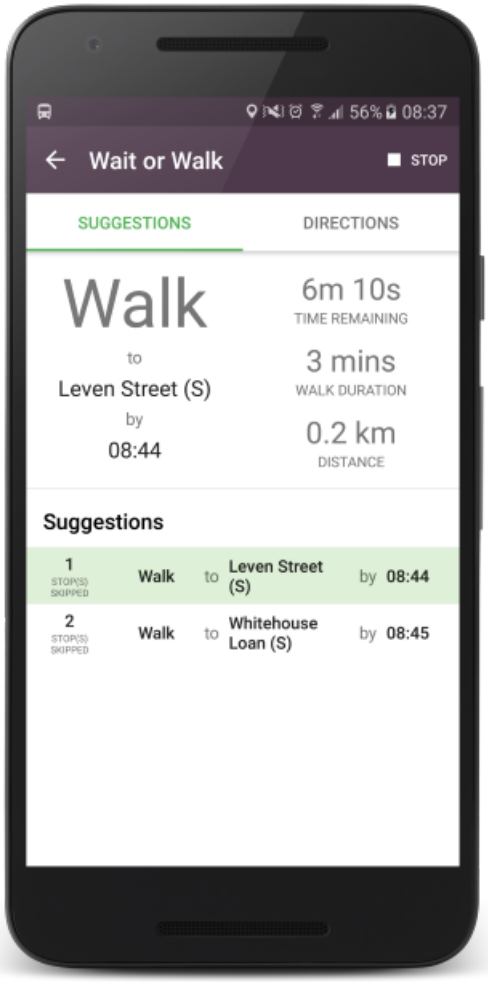

Figure A.15: Screenshot showing Suggestions sub-view of “suggestions” view

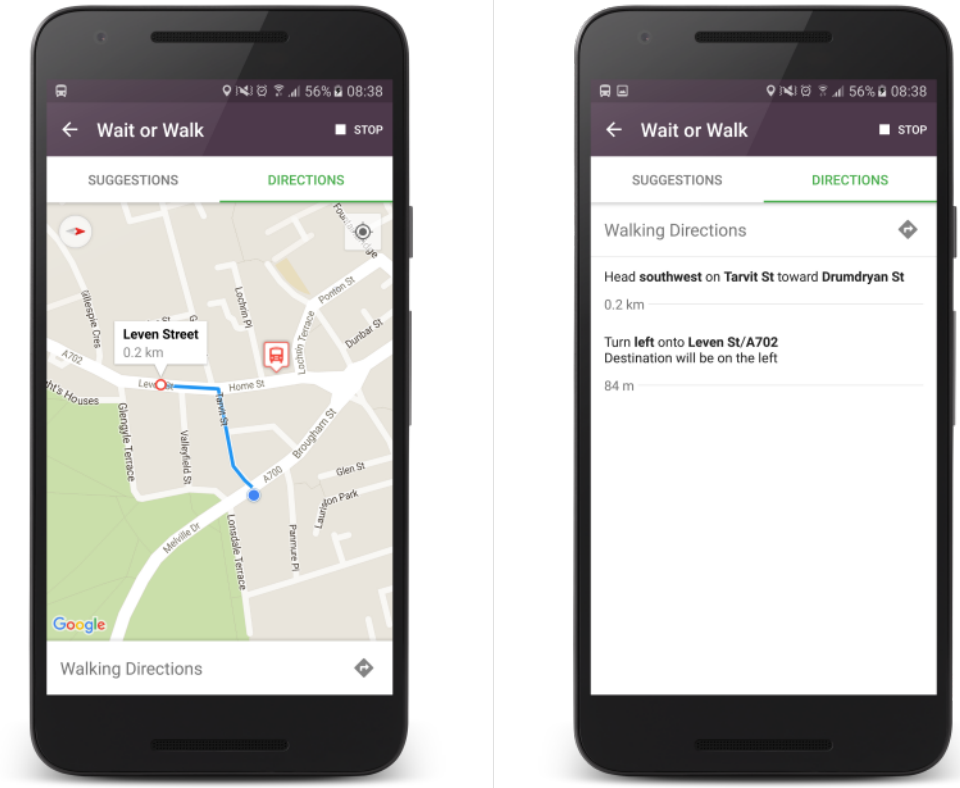

Figure A.16: Screenshots showing Directions sub-view of "suggestions" view

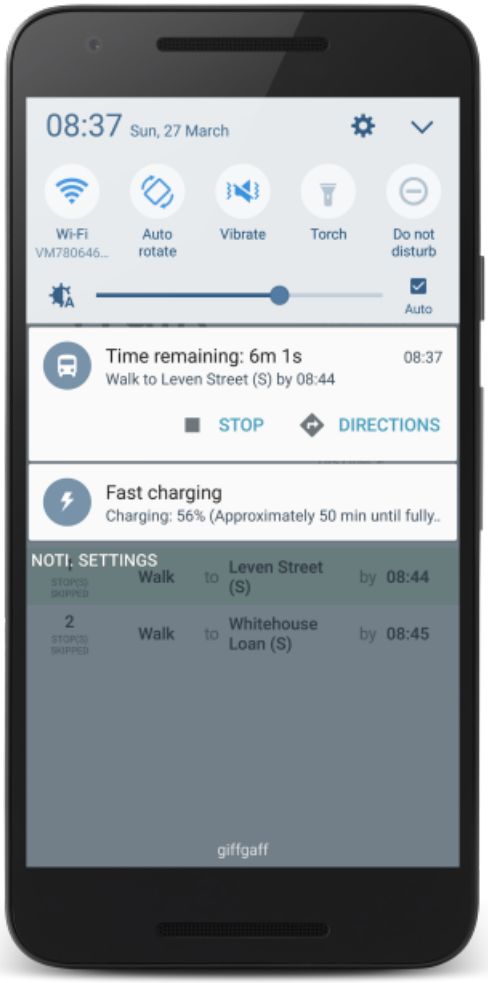

Figure A.17: Screenshot showing a notification indicating that a “wait or walk” activity is in progress

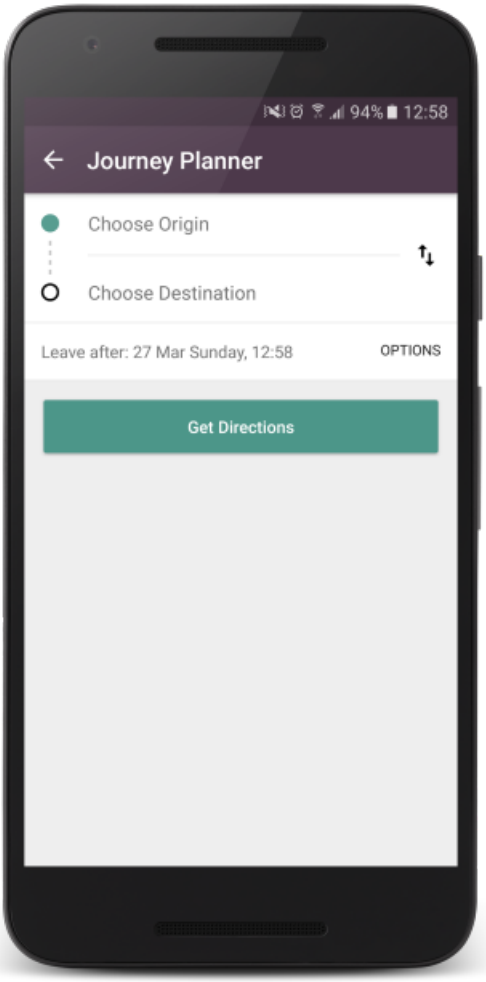

Figure A.18: Screenshot showing "journey planner" view

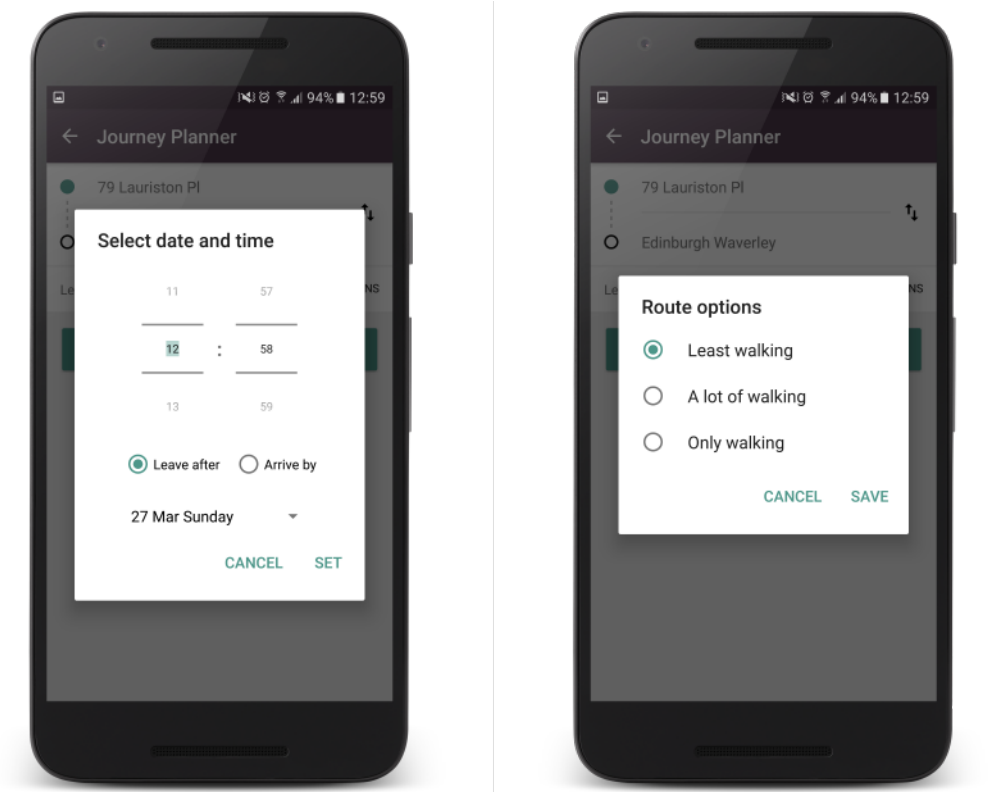

Figure A.19: Screenshots showing time and route option selection in "journey planner" view

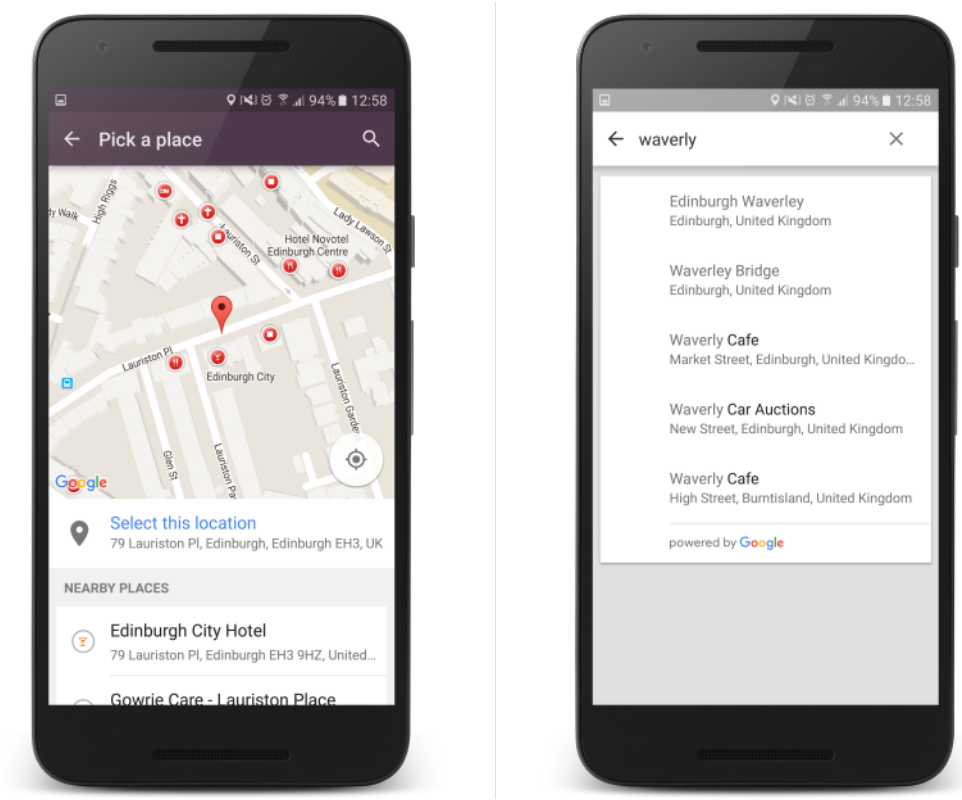

Figure A.20: Screenshots showing origin and destination selection in "journey planner" view

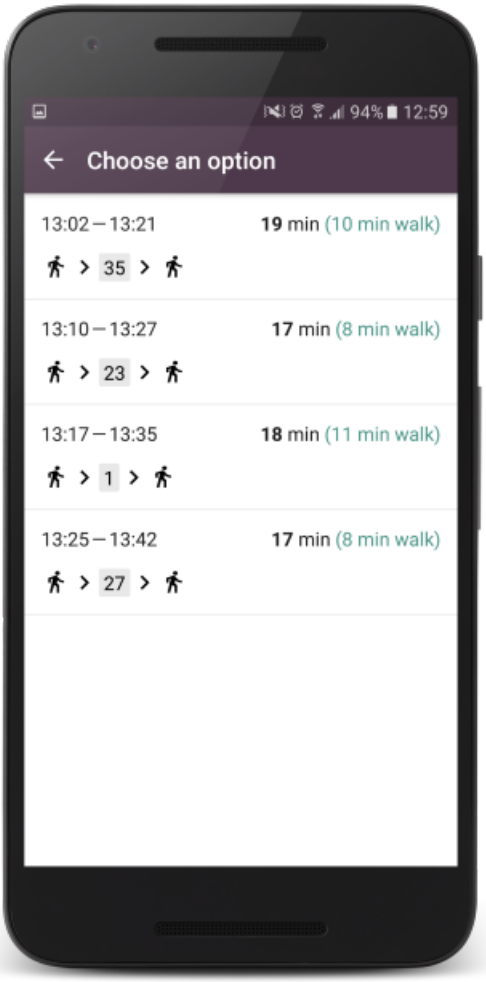

Figure A.21: Screenshot showing “journey chooser” view

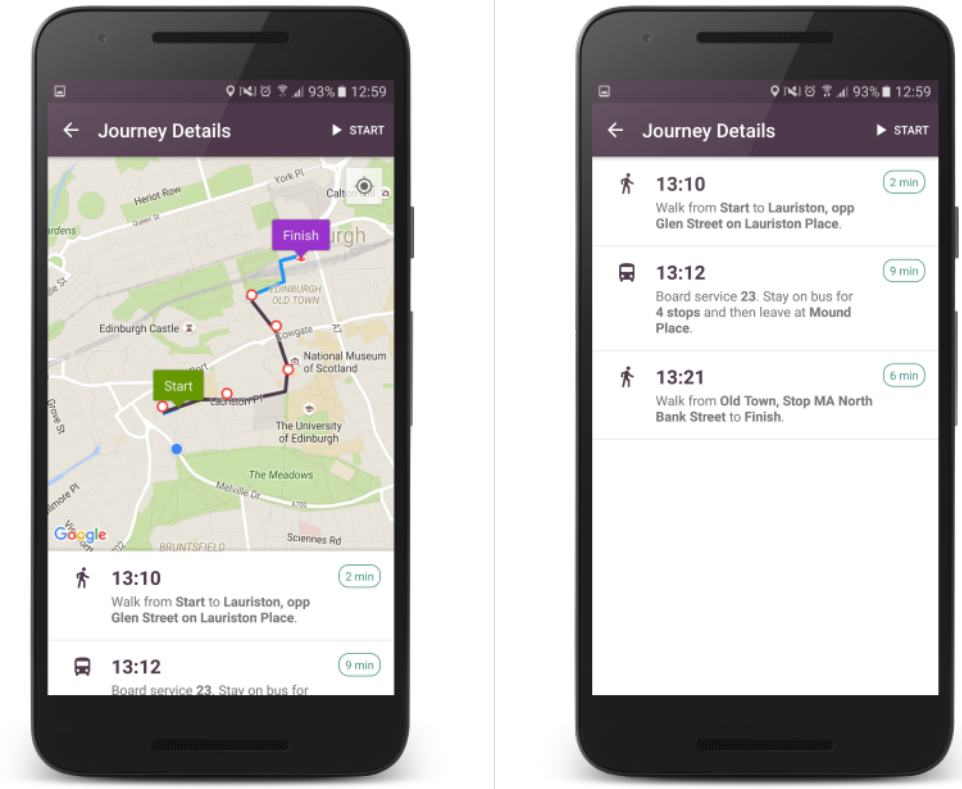

Figure A.22: Screenshots showing "journey details" view

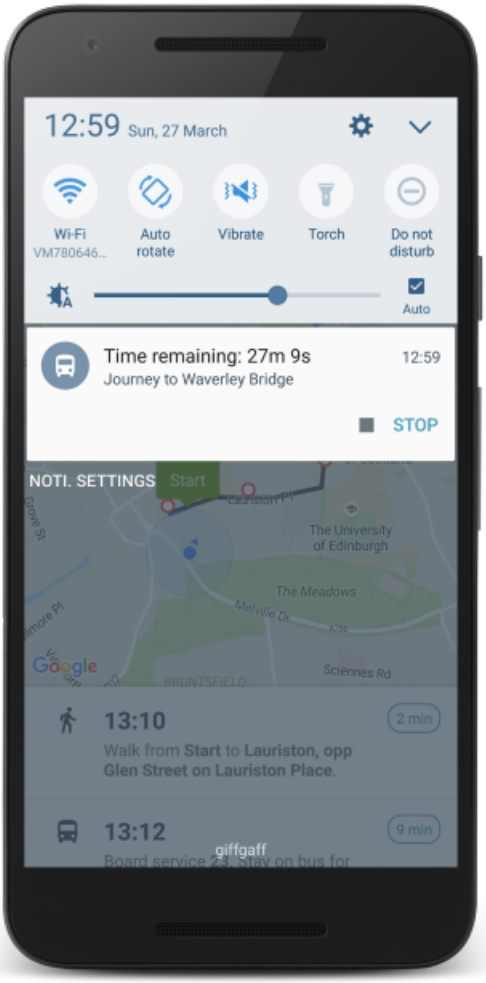

Figure A.23: Screenshot showing a notification indicating that a "journey planner" activity is in progress

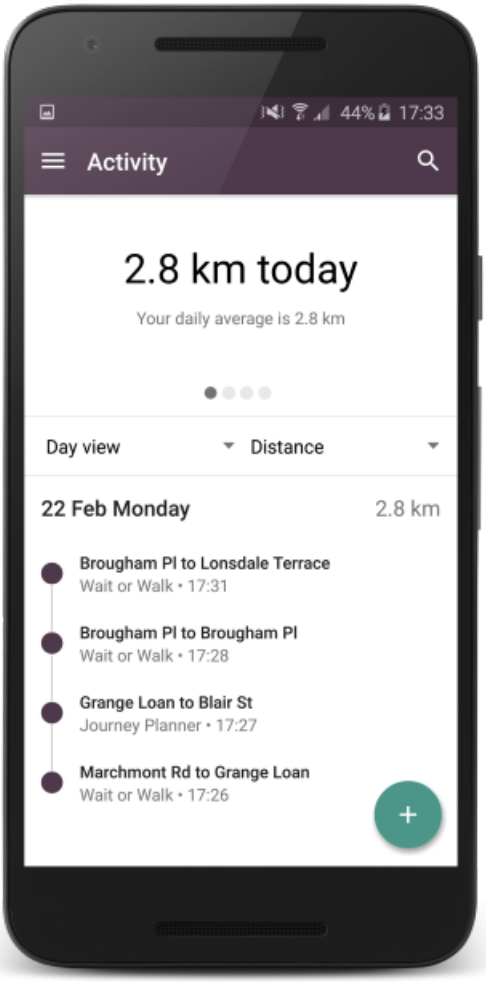

Figure A.24: Screenshot showing "activity time-line" view

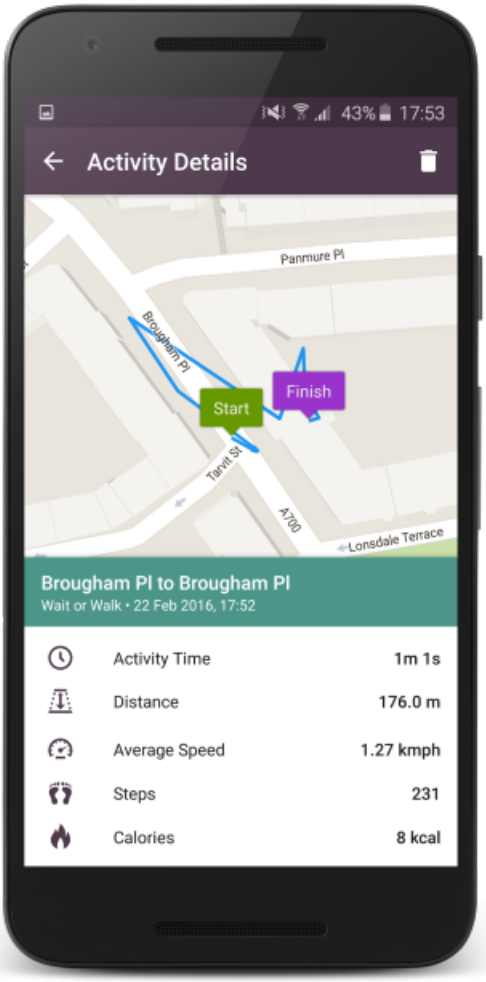

Figure A.25: Screenshot showing “activity detail” view

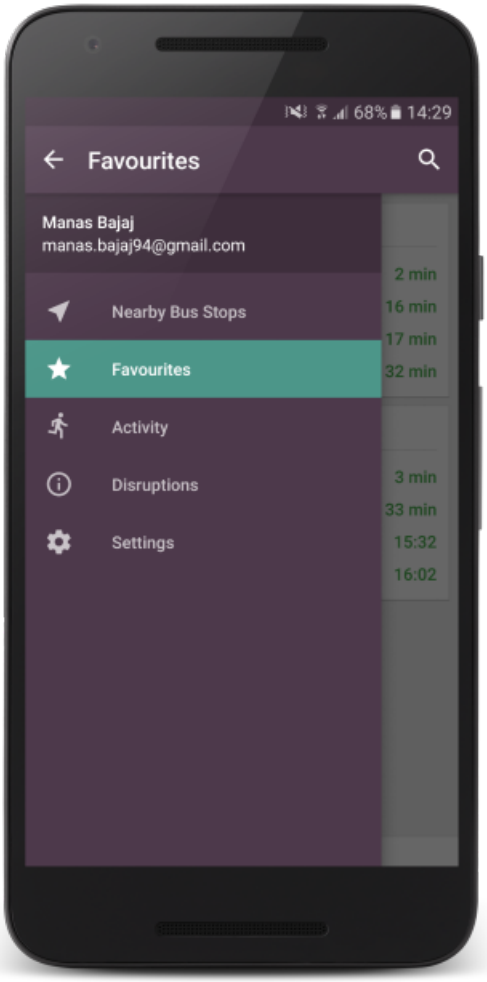

Figure A.26: Screenshot showing the navigation drawer

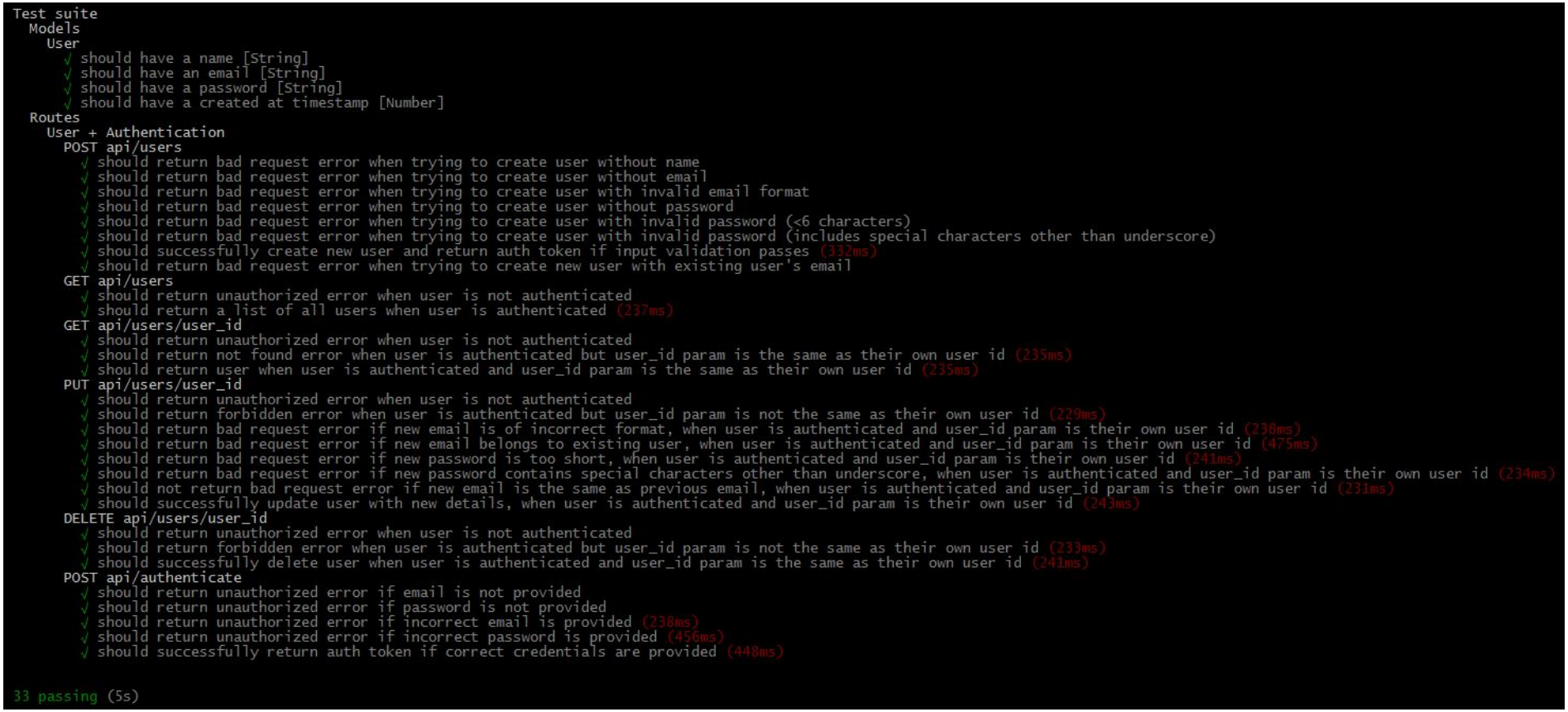

Figure A.27: Server-side tests