Imperial College London

Department of Computing
Individual Project Final Report

# Amble

## A Social Walking App

*Author:*
Jonathan Muller

*Supervisor:*
Professor Michael Huth

June 19, 2017

**Abstract**

In today's society, it is extremely important to keep fit and exercise. This can be seen by the increasing popularity of fitness applications over the recent years. The problem that is posed by fitness apps is to try to engage with the user in an attempt to motivate them to exercise more. Various methods were researched to try and solve this problem, in particular exercising in groups and using gamification, both of which were employed in this project.

This project aims to provide a means to encourage people to exercise more and discover more about the world around them. We have created an iOS application that allows the user to track the walks they go on and view points of interest around them – specifically historical plaques that commemorate notable people. We have also created a back-end API that the mobile app can interact with to store and retrieve user data. The application can be used anywhere in the world, although rural areas may contain a far fewer number of points of interest, which may hinder your experience.

We then tested and evaluated the application with real users to assess whether it had an impact on how often they exercise and whether they were able to discover new places in their area. The results obtained suggest that the application produced did have a positive impact on the user in the areas described above, but with a few limitations that could be improved on in the future.

## Acknowledgements

I would like to express my thanks to my supervisor, Professor Michael Huth, for his guidance and time spent over this project. I would also like to thank my friends and family for their support and inspiration that motivated me during the course of the project.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

In this day and age, it is extremely important to keep fit. With the rapid development in technology in the recent years, people are more inclined to stay inside looking at a screen rather than to go outside and exercise. The main demographic seeing an increase in the use of smartphones and computers are teenagers and young people. The increased day-to-day use of technology is shown to have an impact on the number of obese adolescents in the UK [1], which can increase the chances of people developing serious illnesses including diabetes [2]. It is therefore of great significance to provide a means of exercising that helps people, especially adolescents, keep healthy.

The difficulty in trying to encourage people to exercise lies in the ability to engage the user and find something that they enjoy to do. If exercising is seen as something that you enjoy, it then becomes a pleasure to do rather than a burden. Gamification plays an important part in helping people exercise, and has been used in lots of fitness apps available on the iOS App Store already [3]. The range in which companies have implemented gamification into their fitness applications ranges from a score-based system where users can compete against their friends to a complete game that allows the user to explore the world around them. An example of the latter is the popular mobile game Pokémon Go, where you have to walk around and capture virtual 'creatures' that are scattered around in the real world.

Another problem in trying to keep fit is the difficulty of motivating yourself to exercise regularly. A study conducted in 2001 found that exercising with another person helped to reduce stress and increase calmness compared with exercising alone, however it also resulted in people being more tired [4]. As well as this, exercising with another person also allows you to motivate and set goals for each other to achieve, which can be a challenge when exercising by yourself.

The idea behind this project is to encourage people to walk more often via helping them to find out more about the area around them. Exploring your surroundings can be very interesting and walking is one of the best ways to discover new areas. Let's say, for example, that there is a monument along your commuting route. When travelling via another means of transport, such as a car or train, you might not have the chance to notice this monument. When walking, along with a tool which displays points of interest as you were walking, you would be able to see something that you may never have noticed before.

The idea also extends to helping people walk together to promote regular exercise. A way to do this is to allow the user to schedule walks for a point in the future, and invite their friends along to join them. This means that users will have a fixed event in their calendar that will help keep a more structured fitness routine.

## 1.2 Objectives

The aim of this project is to produce a working application that encourages people to walk more and helps discover new places in the world. The main objectives for the project to measure success on are as follows:

**Obj 1 Encourage walking:** users should be encouraged to keep fit and exercise more. One way to do this is to make it easy for users to exercise with another person as it has been shown to increase motivation and reduce stress. Gamification is another method that could be used to encourage walking more. Users should be able to compete with their friends as a means of motivating one another.

**Obj 2 Help discover the world:** while walking users should be able to discover new places or points of interest in the area around them, which will hopefully motivate them to explore new areas of the world as well as increasing their fitness at the same time.

**Obj 3 Test and evaluate with real users:** evaluation should be conducted with real users to see if the project has an impact on how often they exercise.

# Chapter 2

# Background

## 2.1 Existing Applications

There are a number of existing applications available that attempt to solve the problem posed by this project. These range from fitness applications to various navigation applications, which although may not be completely relevant to this project, do provide some similar features such as place recognition that will be useful to research.

Table 2.1 shows how well each of the existing applications related to this project have implemented certain features. The maximum score for the feature category is displayed in brackets. The full matrix detailing what aspects each feature category is split into and why the score was given to each app can be seen in Appendix A.

| **Features** | **MapMyWalk** | **Strava** | **Let's Walk** | **Google Maps** | **Citymapper** | **Pokémon Go** |
|---|---|---|---|---|---|---|
| Design (2) | 1 | 2 | 0 | 2 | 2 | 2 |
| Ease of use (3) | 3 | 3 | 1 | 3 | 3 | 3 |
| Tracking location (2) | 2 | 2 | 2 | 1 | 1 | 1 |
| Navigation (4) | 3 | 1 | 1 | 3 | 1 | 2 |
| Social interaction (5) | 2 | 2 | 2 | 0 | 0 | 0 |
| Total (16) | 11 | 10 | 6 | 8 | 7 | 9 |

Table 2.1: Matrix showing how well existing walking apps perform at given features. Each app is given a score for a category, with the maximum score shown in brackets next to the feature category.

The rest of this section discusses each application in detail, explaining their benefits and limitations. All of the applications researched are free to use unless stated otherwise.

### 2.1.1 MapMyWalk

MapMyWalk [5] is a popular fitness application for iOS and Android that allows you to track your walks and complete challenges to help you keep fit. You are able to track a walk as you go on one, or log a previous workout that you have done without the app. The app also provides a premium subscription for £4.49 per month, which gives you the ability to monitor your heart rate and set training goals that are designed to help you walk more.

The area in which MapMyWalk lacks is social interaction. A user can publish a walk that they have previously tracked to their profile, but there is no real aspect of communication with other users other than adding each other as friends. There is also a limited level of gamification in the app, with challenges being the only option available to encourage a higher rate of fitness. Challenges can either be added by the user or chosen from a precompiled list, with the latter being fairly limited in the range of options to choose from.

### 2.1.2 Strava

Strava [6] is another fitness application that primarily focuses on running and cycling. It features a sleek user interface with similar functionalities as MapMyWalk. The journey view within the application can switch between either showing the map of your workout or a statistics screen as shown in Figure 2.1, visibly showing the time elapsed in the workout, the distance travelled and your average pace per kilometre. Strava also provides a premium subscription for £5.99 per month, which features personalised coaching and advanced analysis of your workouts.

The setback of Strava is that you cannot track walks in the app as you are constrained to either running or cycling. With regards to this project, it performs well in allowing the user to record and share their workouts but it does not provide any features for the user to explore areas around them. This is expected given that it is a fitness application and does not cater for walking whatsoever.

### 2.1.3 Let's Walk

A lesser-known iOS fitness application is Let's Walk [7]. Users can record new walks and view a list of either their friends' walks or public walks nearby. The app is also focused on helping you maintain a balanced diet – the amount of calories you consume can be added for particular meals during the day. A calorie goal per day can then be set, with the app recording how many calories were burned during a walk and updating the goal accordingly.

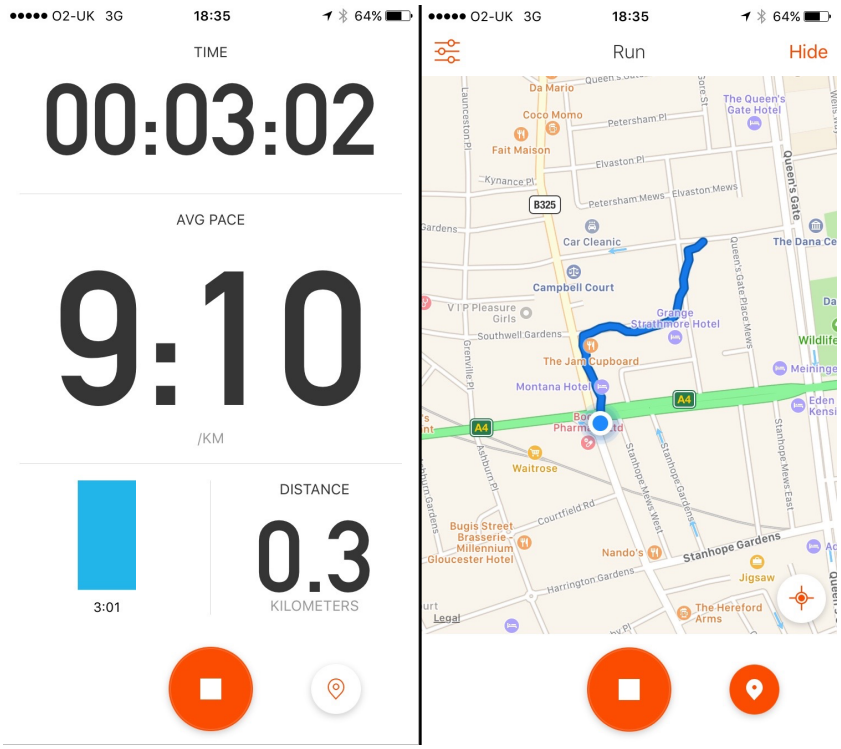



Figure 2.1: Journey view in Strava, allowing you to switch between statistics (left) or a map of your progress (right).

Let's Walk tries to emulate many of the features implemented by the more well-known apps as mentioned above, but a lot of these features seem unpolished. There is a global ranking section of the application showing which users have walked the most over the last week, month or year, however there seems to be little to do with this information other than view a leaderboard.

### 2.1.4 Google Maps

Although not a fitness application per say, Google Maps [8] is one of the oldest services that provides route planning via different transport modes. The mobile app contains current information about public transport, traffic and displays well-known cycling routes on a map, however there is little in the way of customisation for walking. When entering a destination, the app generates a route but users can also choose from a few different routes on the map, with the app showing the difference in time each one would take. However, no information is given as to whether a certain route is quieter than another, for example.

One feature of the Google Maps iOS application that is interesting to note is the ability to search for places along a route during a journey. Once a user has started a walking journey,

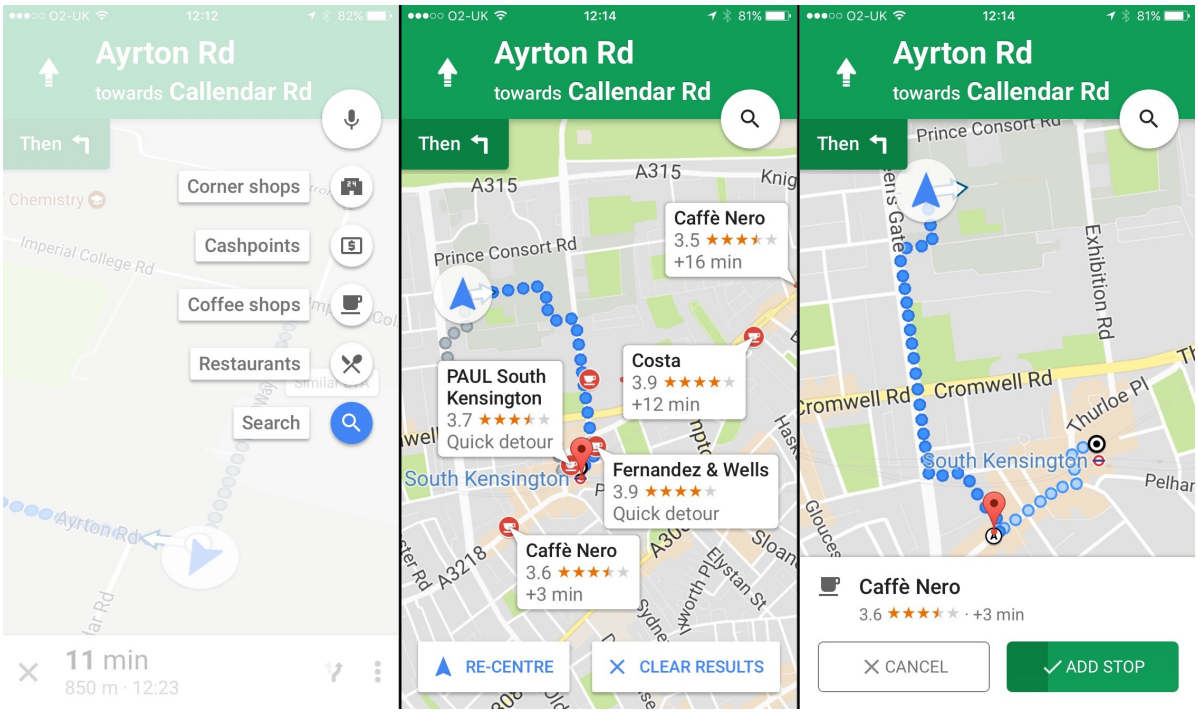



Figure 2.2: Adding places along your walking route in Google Maps for iOS. A list of categories to choose from (left), then shows all the places from within a search on a map (middle). A stop can then be added and the journey will be updated (right).

they are able to search for places that are along the route. Google provides some categories of places to choose from, such as cashpoints and restaurants, but users can search for a specific place if they wish. The app will then display the results of the places search on the map, showing how much additional time would be added on to your journey if you were to stop at a place, if any. One or more places can then be added to your journey and the walking directions will subsequently update to include these new stops. Figure 2.2 shows an example journey from Imperial College to South Kensington Underground station. It details the full process of choosing *coffee shops* as the place category, selecting a particular coffee shop on the map and the stop being added to the journey.

The places search feature is important as it is unique within any of the existing journey planner apps I have researched and it relates to one of my objectives regarding displaying points of interest when a user is on a walk (**Obj 2**). More research is conducted in Section 2.3 to discover what tools these applications use to implement this feature.

### 2.1.5 Citymapper

Originating in London, Citymapper [9] has become one of the leading journey planners for major cities around the world including Paris, Barcelona, New York, Tokyo and Sydney. One of the key features of Citymapper is that different modes of transport can be combined to create a faster journey time. For example, a journey from Imperial College to Oxford Circus (as shown in Figure 2.3) could just use the Tube, but it could be faster to hire a bike and cycle to a different station and then take the Tube.


Figure 2.3: Two routes generated on the Citymapper website superimposed – cycle and Tube route (Route 1) and Tube only route (Route 2).

The walking directions in Citymapper are relatively limited. If you choose to walk on any route that you input, you are greeted with the screen shown in Figure 2.4. Details such as estimated time of arrival, calorie burn and time of journey are displayed on this screen. Once you press *Go*, you are transferred to the journey view, which simply tracks your location on a map and allows you to share your estimated time of arrival with others.

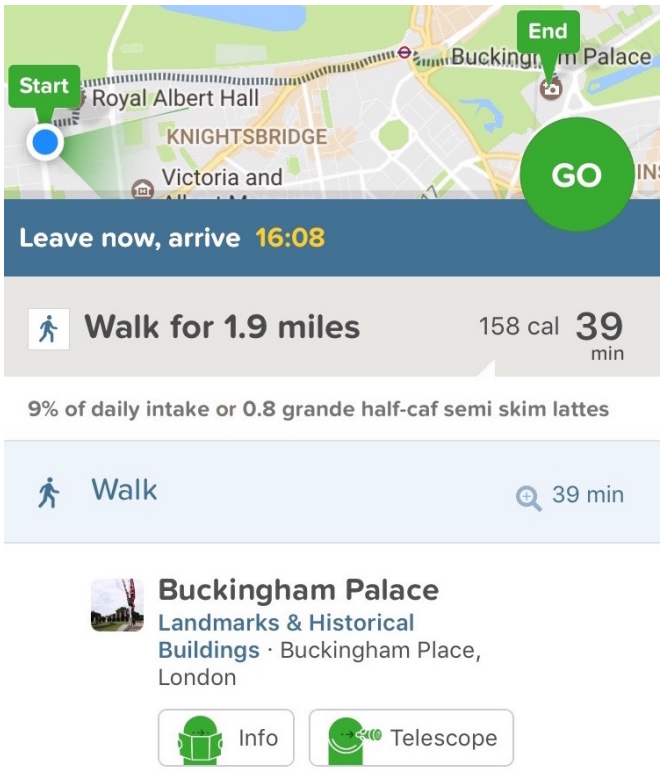



Figure 2.4: Walking view in Citymapper

### 2.1.6 Pokémon Go

Pokémon Go, released last July, quickly grew to become one of the most popular games of the year. Although not necessarily a walking application in the normal sense, the aim of the game is to capture virtual 'creatures' called Pokémon that appear in real world places. Thus, the game motivates you to walk more to collect more and more Pokémon.

One of the more interesting parts of the application that is relevant to this project are the Pokéstops within the game. A Pokéstop is a location in the game where various in-game items can be collected. They are displayed on a map using blue beacons as shown in Figure 2.5. These locations were crowdsourced by users of the game and are normally points of interest in the area such as a statue, a building or a famous plaque. Although not completely, this feature of the application does somewhat tie into my objective for helping people discover the world (**Obj 2**).

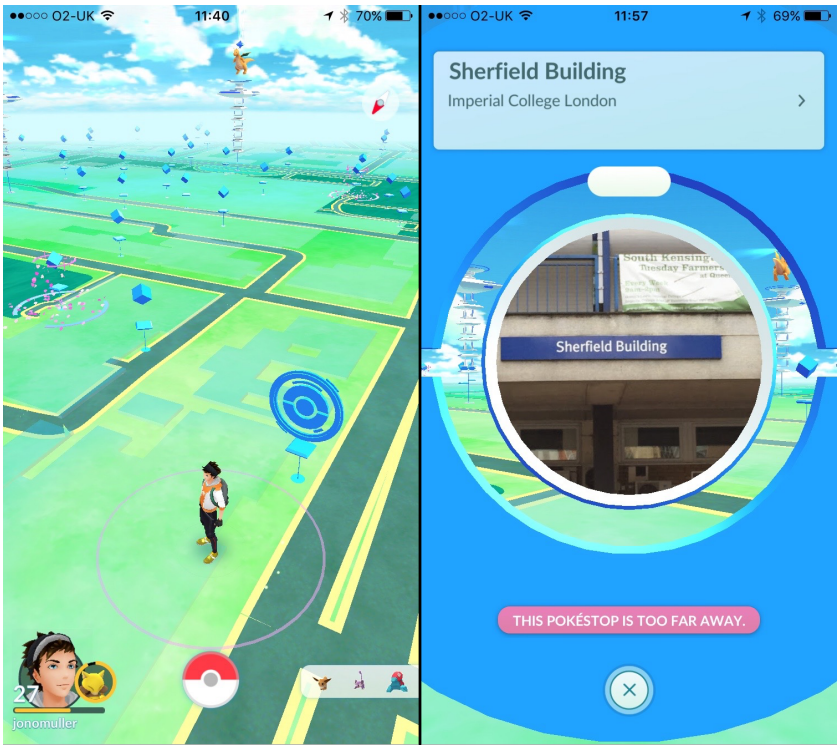



Figure 2.5: Screenshots of Pokémon Go showing Pokéstops in an area (left) and a detailed view of a Pokéstop (right).

### 2.1.7 Summary

From the subset of fitness applications that I researched in this section, it can be seen that some of the objectives I proposed in Section 1.2 have been achieved but no single application encompasses all of my proposed objectives. I have found that it is important for this project to have a sleek design and easy-to-use interface as this is something that stood out straight away when looking at existing applications.

## 2.2 Gamification

Gamification is used in mobile applications not only for fitness but also in a wide range of areas including productivity, finance and mental health. We have seen how gamification can be used in existing fitness applications, with some apps setting challenges for users to complete within a given timeframe – such as running a half marathon in February.

This section discusses the alternative uses of gamification in existing applications, followed by the way in which gamification could be used in this project.

### 2.2.1 Alternative Uses

An application and website called Habitica [10], labelled as a "gamified task manager", helps motivate you to complete household tasks by unlocking features and levelling up an in-game avatar. Completing real-life tasks earns you gold for your character, which can then be redeemed for either virtual rewards such as equipment or real-life treats like watching an episode of a TV show, for example. The home page of Habitica, shown in Figure 2.6, is split into columns containing your bad habits, daily tasks, to-do list and rewards. It also shows the progress of your avatar, along with what is needed to progress to the next level. This type of app helps you become more productive at home in a fun and creative way that users enjoy.

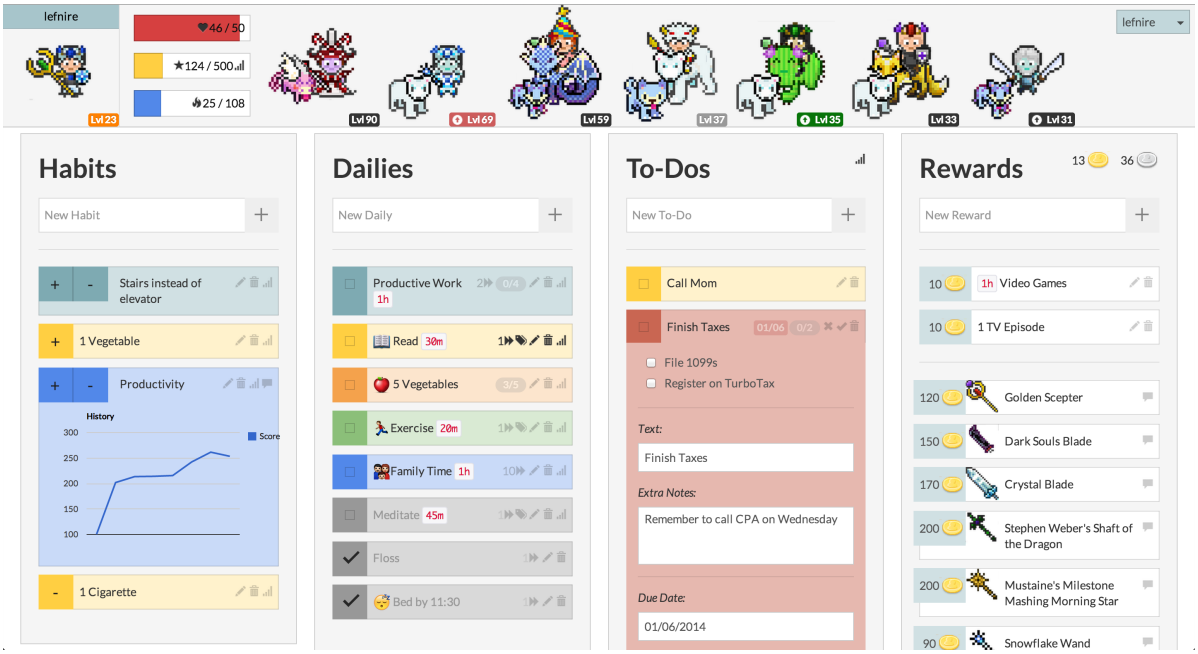



Figure 2.6: Home page of Habitica showing your habits, tasks and rewards as well as your avatar at the top [11]

Gamification is also used in some finance applications. Mint [12] is an application available in America that links to your bank accounts to help you manage your bills and track how much money you spend. It shows a breakdown of what you are spending your money on and creates a budget with the option of using any left over money on goals designed to help you save money, as shown in Figure 2.7. You can create goals to be either a long-term one-off payment – buying a house, for example – or a short-term monthly payment such as a subscription service. This game mechanic of working towards a goal to help you buy something you want is extremely effective and is why gamification is so widespread.

The idea of using gamification in applications is to motivate you to complete some form of task that you otherwise might not have attempted. It can make an app seem more engaging to

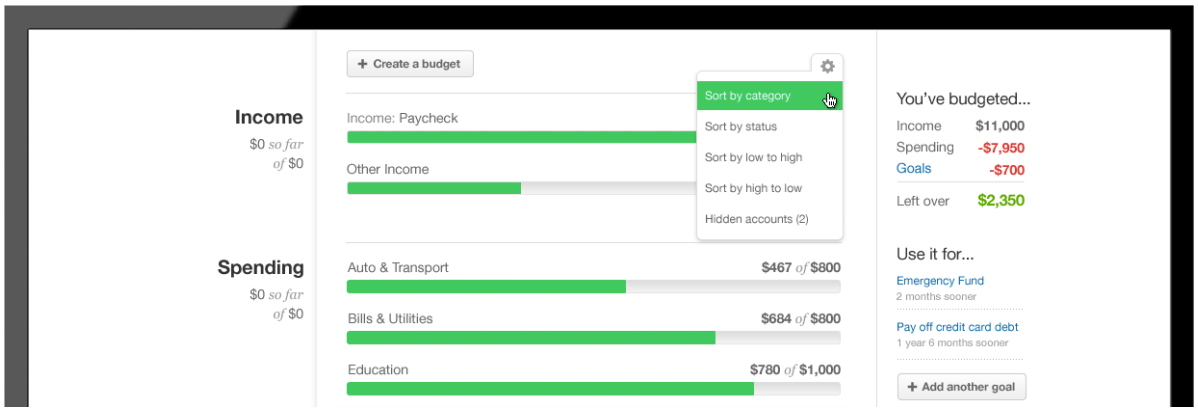



Figure 2.7: The breakdown of spending in Mint, giving you the option to use any left over money from your budget for one or more goals [13]

the user, providing them with something to achieve every time they use the app. It is especially useful in fitness applications as keeping fit and staying healthy is very important, and should therefore be carefully considered for this project.

### 2.2.2 Summary

In terms of this project, I had to consider two main factors regarding the type of gamification to use: its usefulness to the user and the feasibility of implementation. I discussed with potential users the extent to which they thought gamification should be used, ranging from a achievements-based system to a full character game. While some did say that the game aspect would be fun, many said the way that would motivate them to exercise more would be an achievements and scoring system. Achievements (and points) would be gained the more you walk, with the points contributing to a tally on the user's profile.

From this, I created a list of achievements that I found would be most useful to implement initially.

- **Distance:** earned for every walk. The further you walk, the higher number of points will be gained.
- **Streak:** earned if you walk everyday for a number of days, forming a walking streak. The number of points gained is based on how many days are in the streak.
- **Group exercise:** earned if you walk in a group, with the number of points depending on how many people there are.

The use of achievements also allows for extensibility in the future, with more able to be added at any time.

## 2.3 Points of Interest

In order to achieve my proposed objective of helping users discover more of the world around them (**Obj 2**), I had to research which sources would be able to provide me with nearby points of interest based on a location.

### 2.3.1 Google Places API

Google's Places API [14] would be a good resource to use, allowing you to search for places by over a hundred types including *point of interest*, *place of worship* and *museum*. One issue with using Google Places is that you must use Google Maps to display the places on. From the Google Places Policies, *"If your application displays data from the Google Places API for iOS on a map, that map must be a Google map"* [15]. I would need to consider this issue when choosing the map source to use for the application, which is discussed in Section 2.4.4.

### 2.3.2 MKLocalSearch

Apple provides a framework for obtaining places called `MKLocalSearch` [16]. To generate a list of places, you pass the `init()` method a `MKLocalSearchRequest`, which contains a 'natural language query' string describing what type of place you would like to search for on a map.

There seems to be little documentation online explaining what range of places this API returns. After some initial testing, I found that the downside of this service is the natural language search query – you cannot search by the type of place. This results in unintended results being returned from a request. As an example, to find any monuments nearby you would search for the keyword *monument*, but the search results will include Monument Underground station.

### 2.3.3 Pokéstops

A different option that could be used to generate points of interest around a geographical location originates from Pokémon Go. As mentioned in Section 2.1.6, Pokémon Go contains thousands of Pokéstops – crowdsourced points of interest from all over the world. There exists an API [17] that returns a list of Pokéstops in JSON format given a location. Each item that the API returns contains the name and location of a point of interest, as shown in Listing 2.1, and so could therefore be used in this project.

```
{
    "distance": 44,
```

```
        "name": "Sherfield Building",
        "bearing": 200,
        "latitude": 51.498359,
        "image": "http://lh3.googleusercontent.com/07q4ms3tgDKsQMy04xye
                  _i-UiraPO3jOS18TXwKpTMecgIXm2jXByO1CAUWVW9vNgqfx12ZtjqLdZrOlfsPu",
        "guid": "2cc0f9d9c7ba49348299c15749c49ea1.16",
        "compass": "S",
        "longitude": -0.178544
    },
```

Listing 2.1: Example of one item returned from the Pokéstop API, with attributes including its name, latitude, longitude and distance from your location

### 2.3.4 Historical Plaques

Based on initial feedback of the project, users suggested that the multitude of 'blue plaques' around London would be a good way to display points of interest. These plaques are placed on buildings and public places in London to commemorate a notable person or place in history. The blue plaque scheme is administered by English Heritage [18], a charitable organisation responsible for the upkeep of over 400 historical sites across the United Kingdom.

All data from the plaques from English Heritage is publicly available by request under the Freedom of Information Act 2000 – an act to publicise information held by public institutions. The English Heritage website provides an option to search for a plaque given a query string, however there does not seem to be an option to search for plaques given a map area, meaning it would not be that effective for the purposes of this project. A request to retrieve all plaque data from English Heritage would both be tedious and a waste of space, so I therefore needed to look for another option.

I came upon an open source service in OpenPlaques [19] – a collection of plaques run by volunteers, which seemed to be well populated. OpenPlaques contains all the English Heritage blue plaques from London, but also other plaques around the UK and the world that various other organisations have put up. OpenPlaques also offers an API that supports querying plaques in an area by supplying a bounding box.

### 2.3.5 Summary

The options discussed all provide extremely different ways to retrieve points of interest. Both Google's and Apple's services may be the most convenient – as they would be integrated into the map source chosen – but they both contain limitations, most importantly that there is no extra information returned about a point of interest other than its title.

The other two services that use Pokéstops and historical plaques provide, in my opinion, a more appealing approach to finding new points of interest in an area. Based on my own thoughts and the thoughts of users I asked, I chose to use the latter option as the source of points of interest for the project. Using historical plaques seemed the more interesting option to enable the user to learn about places and motivate them to investigate more into these places, which is essentially what my aim was.

## 2.4 Technologies

The features of existing applications are not the only important part to research – the technologies that they use to implement these features are just as useful. This section discusses how existing applications implement certain features and which technologies integrate well with each other.

### 2.4.1 Mobile Operating System

Many of the applications that I have researched have been developed for iOS, the operating system running on iPhones and iPads. I have chosen to develop my application on iOS due to my previous experience with iOS app development. Another factor in this choice is that I have gained a lot of experience programming in Swift, one of the programming languages used to develop iOS apps, over the last few years. Swift is extremely readable and easy to use, which is the reason why I am choosing to use it over the other programming language available, Objective-C. I am also familiar with the tools required to develop an iOS application, specifically the integrated development environment (IDE) Xcode.

### 2.4.2 Version Control

Version control is a crucial part of the development process with any project, especially this implementation-heavy project. It provides a full history of changes made to a codebase along with the ability to revert back to previous versions of code and work simultaneously on different features by creating branches. Git and GitHub are my chosen version control system and platform respectively due to past experience and familiarity.

Along with version control, errors or bugs in code must be detected quickly in order to not hinder the development process. To do this, the practice of continuous integration (CI) was used to integrate and test code frequently, making it easier to locate the origin of an error. There are many CI tools that can be used to automate code testing including Jenkins, Travis CI and BuddyBuild. I had no prior knowledge with any of these tools and so I researched the benefits and drawbacks of each service, which can be seen in Table 2.2.

| | **Jenkins** | **Travis CI** | **BuddyBuild** | **GitLab CI** |
|---|---|---|---|---|
| Platform | Windows, macOS, Linux | Hosted | Hosted | Hosted on GitLab |
| Cost | Free | Free for students | ~£55/month or free trial | Free for small individual projects |
| Main Feature | Plugins available for variety of languages | Linked well with GitHub | Designed solely for mobile development | Integrated with GitLab |
| Platform Specific | No | No | Yes | No |

Table 2.2: Comparison of continuous integration tools

I concluded that Travis CI was the most suitable option for me since it was hosted on GitHub and is not platform specific, meaning that I could use it to test both the mobile app and API even though they were written in different languages. Using an option like BuddyBuild could only be used for the mobile app, with another tool needed for the API.

### 2.4.3 Location Tracking

To track the user's location within iOS, an application can use the classes from the Core Location framework [20] inbuilt into the iOS SDK (software development kit). This framework allows a developer to obtain the user's current location in the form of latitude and longitude. To keep a history of where the user has been, these coordinates could be stored in an array and updated every few seconds. This will be useful for my application to show on a map where a user has previously walked.

### 2.4.4 Map Source

One of the most important technology aspects to consider is which source of map to use. The majority of existing applications use Apple's maps apart from, of course, Google Maps. This is because Apple's MapKit framework [21] is much better integrated with the Core Location framework mentioned above, making it much easier to display the user's location on MapKit than on the Google Maps SDK. It also provides convenient functions for coordinate conversions and calculating distances between two coordinates, which is useful for this project when trying to calculate the distance of a tracked walk.

However, there are some advantages of Google Maps over Apple Maps – one being that Google Maps tends to be more detailed than Apple Maps. An example of this is shown in Figure 2.8, where in my opinion roads are easier to see and buildings are clearly visible on

Google Maps than on Apple Maps.

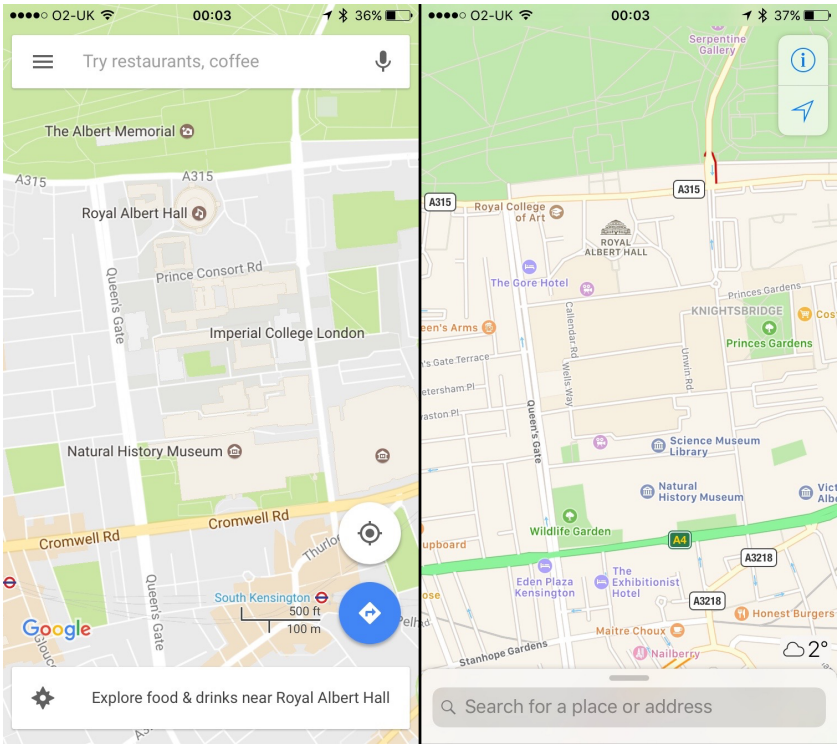



Figure 2.8: An example of the difference between Google Maps (left) and Apple Maps (right), showing Imperial College London.

For the reasons listed above, I chose to use Apple Maps and their MapKit framework as the source of maps within the application. The one issue of the lack of detail in some areas of the maps was not deemed important enough to impact my decision and did not outweigh the benefits of the MapKit framework that are listed above.

### 2.4.5 Server Architecture

Before deciding what specific technologies to choose for the server, such as the programming language, we must consider the type of architectural style to use. The aim of the API exposed by the server is to provide an abstraction between complex database queries and provide functionality that a client can use to query and update details. The Representational state transfer (REST) architectural principle fit the needs for this aim very well. Every component in a RESTful web service is a resource that can be accessed via the standard HTTP methods, including *GET*, *POST* and *DELETE*. In comparison with other web services, such as the Simple Object Access Protocol (SOAP), RESTful services have much less overhead when sending data due to the extra XML header information sent when using SOAP.

With the architecture chosen, I assessed the options for the language to write the API in. There are a wide range of programming languages that could be used, and it came down to personal preference and how well it would suit the needs for this project. The popular options include PHP, Ruby, Python and Node.js [22] – the server-side environment run in Javascript. The only language in this list that I have experience in is Python, which would mean I would have to spend less time learning the language if I were to choose it. However, the advantage of Node.js for the purpose of this project is that objects in Javascript are built using Javascript Object Notation (JSON).

RESTful web services support the use of multiple different data formats to serialise responses from the server, including HTML, XML and JSON. By choosing Node.js as the server language and JSON as the data type, I was able to easily parse data from HTTP requests and send responses in JSON, despite not having any previous experience with Javascript.

### 2.4.6 Database

There are two types of database that could be used: relational or non-relational. Relational databases are based upon relational algebra where data is stored in tables with queries to this data in the form of Structured Query Language (SQL). Each table in a relational database normally identifies a type of entity, and each instance of that entity is uniquely identifiable so that it can be referenced in other tables. SQL therefore supports the querying of this data across multiple tables using relational operators such as joining two tables. The most used services to model data in a relational database are MySQL and PostgreSQL, with the latter providing support for storing arrays – a useful feature for this project as the coordinates of a walk need to be stored.

The opposite of this data model is a non-relational, or NoSQL, database in which each data entry is stored as a JSON-like document. This approach has many advantages for this project, primarily the similar use of JSON to both store data and build objects from in Node.js. As a result, data can be queried or inserted directly from a network request without any type conversions. NoSQL databases also have the benefit of faster query speeds and the ability to scale data horizontally – an important factor to consider when designing a database.

The leading service to use for managing document-based NoSQL databases is MongoDB. It was used in this project especially due to its similar use of Javascript and JSON, both of which are already employed in the Node.js API. I also used a popular object data modelling library, Mongoose [23], which helped with validation and allowed me to define schemas for each of the type entities.

### 2.4.7 API Deployment

I opted to host my API on the platform-as-a-service Heroku [24], with the alternative being Imperial's Cloudstack service run by the Department of Computing. Having had little experience with either of these services, I learned that Heroku had certain features that gained itself an advantage over Cloudstack.

Heroku is integrated with GitHub very well, my chosen version control platform, supporting automatic deployment when commits are made that can also be dependent on the results of continuous integration tests. Heroku also has a number of add-ons that can be used on web applications, including mLab – a cloud-hosted version of MongoDB that is essential for me following on from the choice of database discussed in the previous section. Conversely, Cloudstack does not support MongoDB outright and would therefore take more time to set it up.

### 2.4.8 File Storage

An online file storage system was needed for the project to store thumbnail images of tracked walks, which is discussed in more detail in Section 3.4.4. Due to the ephemeral nature of Heroku's file system, it is not possible to store permanent files there and therefore an alternative online file storage service must be used.

From the wide range of services available, there is one stand-out option that I chose to use – Amazon's simple storage service known as AWS S3. Its free usage tier allows up to 5GB of data storage which would be plenty for the purposes of this project. It also provides an SDK for Javascript which was necessary in order to implement the storing and deletion of images from the API.

## 2.5 Summary

The outcome of the background research of the project has enabled me to narrow down the requirements that I want to complete. Researching existing fitness applications was useful to determine the way in which workouts are recorded, as well as the range of ways gamification is used. Alternative uses of gamification provided a different approach, including gaining rewards for completing household chores, however I chose to use an achievements-based system based on user feedback. The result of the points of interest research is a service that provides interest and motivation for the user – a core objective of the project.

Finally, I was able to choose the technologies that would suit the project best based on the conclusion of the previous research. We can now create a detailed list of requirements that

should be completed in the project, and begin to design and implement these requirements.

# Chapter 3

# Design

## 3.1 Overview

The project can be split into two main sections: the front-end mobile application that the user can interact with and the back-end server to store data and communicate with the mobile app.

These two sections link together very closely and are both required to produce a working implementation. It is therefore extremely important to not only consider the design of the user interface but also more technical aspects, such as the way the user's data is stored in the database and how the API will communicate with the mobile app.

## 3.2 Project Requirements

Based on the broad objectives from Section 1.2 and the background research conducted in Chapter 2, a detailed list of technical requirements was created. These requirements essentially map to features that need to be designed and implemented in the project. The objective that each requirement supports is shown in brackets.

**Req 1** Build a fully functioning iOS application with a simple design and an easy-to-use user interface (**Obj 3**).

**Req 2** Allow the user to track the routes of the walks they go on, as well as provide statistics about the walk such as distance travelled and number of steps (**Obj 1**).

**Req 3** During a walk, the application should display certain points of interest on a map near the user's current location (**Obj 2**).

**Req 4** Each user should be able to register an account within the application and publish their tracked walks to their profile (**Obj 1**).

**Req 5** Users should be able to invite other users to go on a walk together and schedule this walk for a point in the future (**Obj 1**).

**Req 6** The application should contain gamification - each user will have a score on their profile based on how far they have walked in total, how many walks they have been on and how often they go for a walk (**Obj 1**).

With these requirements established, I was able to begin the design process for both the front and back-end in accordance with the features described.

## 3.3 Mobile App Design

The mobile application produced for this project is the front-end product that the user interacts with. It therefore needs to have a sleek user interface and be well organised and easy to use.

### 3.3.1 User Interface

Before any implementation took place, I conducted an initial survey to find out various technical details such as which features would be most popular, which in turn allowed me to generate some design mockups of the application.

I chose to try and follow the iOS Human Interface Guidelines [25] provided by Apple. These guidelines cover the basic design principles that should be used when creating a clear, fluid and easy to use iOS application. The guidelines provided a number of concepts to help structure how data is organised within the app.

Different app-level sections of an iOS application should be split using the tab bar UI element. This is a bar that is constantly present at the bottom of the screen containing a number of icons that the user can select to switch between sections of the app. The app for this project can be split into three main sections: tracking walks, managing sent and received walk invites and viewing the user's profile. From this, I was able to construct mockups for each of these sections – shown in Figure 3.1.

As the design process wore on, I was able to create more detailed mockups utilising more of the resources from the Human Interface Guidelines. I updated the mockups so that each screen of the application contained a navigation bar – a widely-used UI design pattern in iOS. Setting this as a standard throughout the app provides the user with a constant place where they can navigate between screens of the app and alter content that is displayed on the current

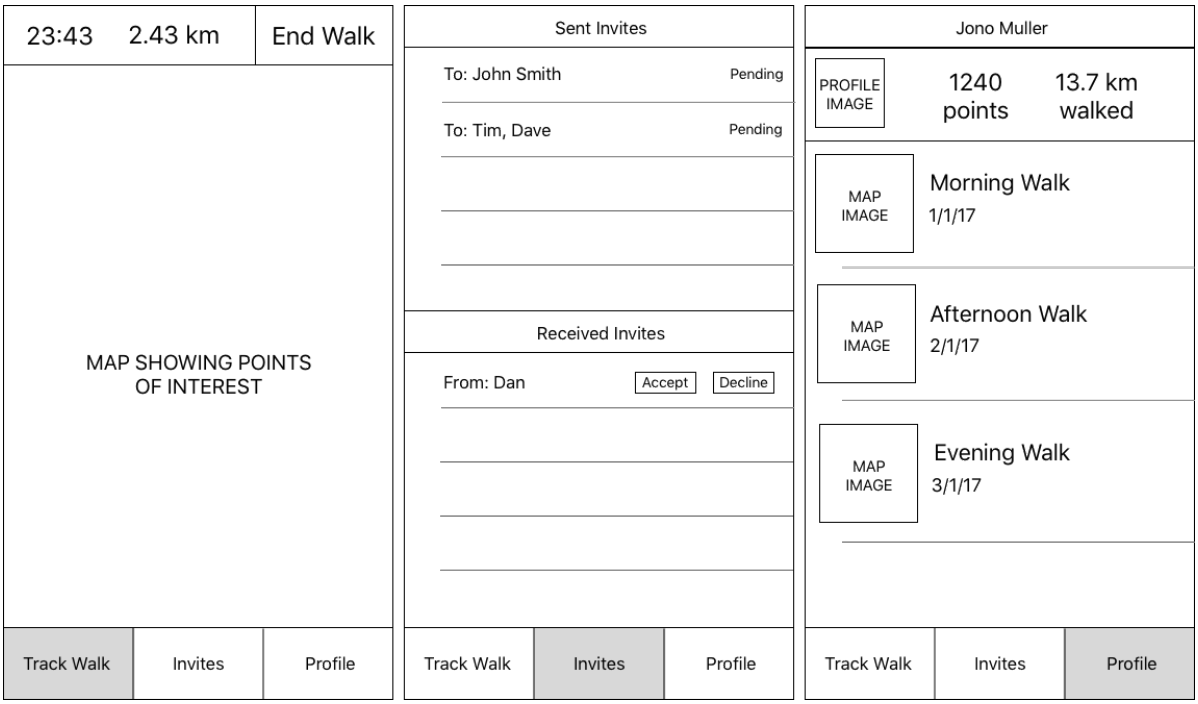



Figure 3.1: Initial mockups of the mobile application, showing how the app is split up into three main sections using the tab bar design.

screen. I also chose to use collections to display the list of walks on a user's profile instead of the standard tabular method, mainly due to its more visual approach. The updated mockups for the profile view containing collections and the walk detail view showing a navigation bar can be seen in Figure 3.2.

Another design element that I chose to use is a segmented control – a switch containing two or more elements to toggle between different views, normally used to switch between similar content in the same section. Its purpose fits nicely into the invites section of the app, where both sent and received invites need to be displayed but each contains similar information. By using a segmented control, placed in the navigation bar to maximise screen content, users will be able to toggle between viewing their sent and received invites from within the invites tab. The full mockup is displayed in Figure 3.3.

### 3.3.2 Model-View-Controller Design Pattern

An encouraged design pattern to use when creating a mobile application, namely an iOS app, is the Model-View-Controller (MVC) pattern. When following this pattern, objects in an application are split into three layers. Relating to this project, the three layers will be used as follows:

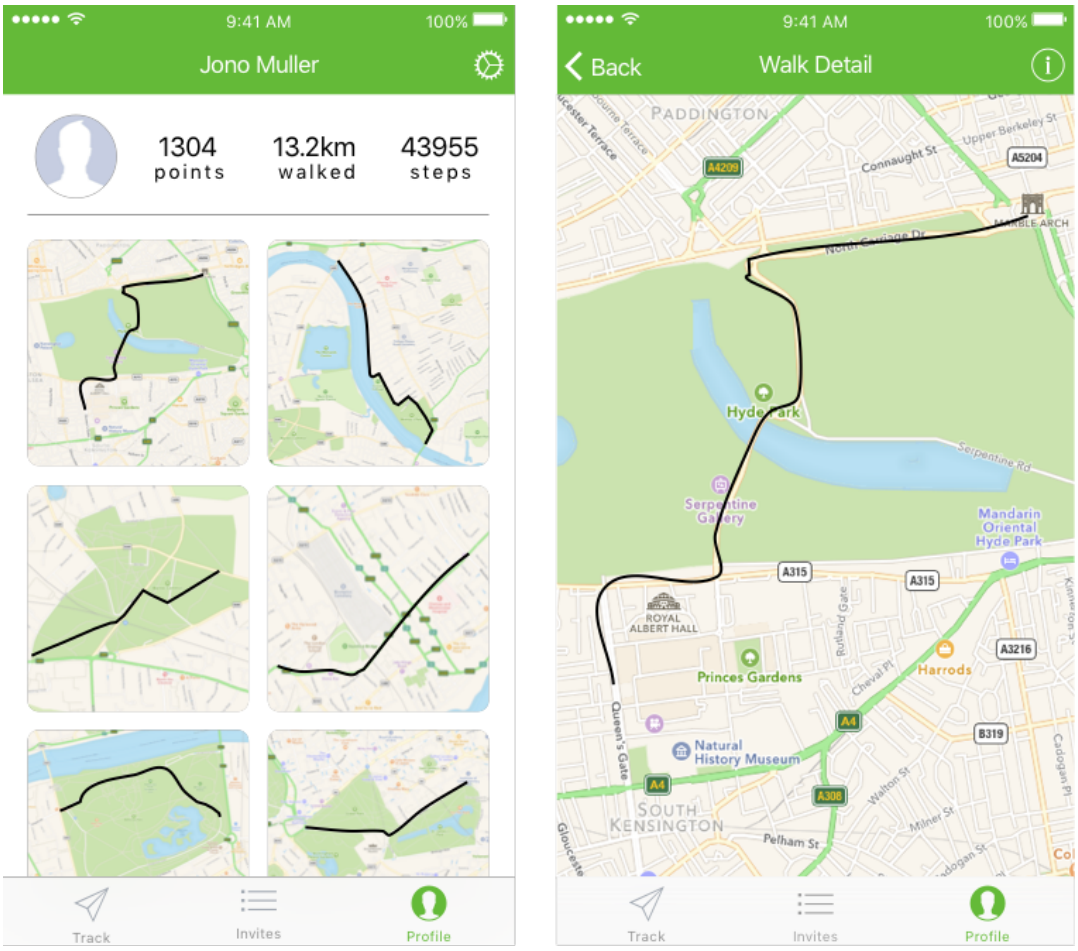



Figure 3.2: Updated mockups of the user profile view (left) that contains a collection of the user's walks, and the walk detail view (right) that is presented when one of the walks from the profile view is selected.

- **Model** stores the data within the application and specifies the logic that alters that data. For this project the model will contain data relating to user information, tracked walk details and invites.
- **View** specifies objects that are visible to the user. They can either be used directly from Apple's UIKit framework or customised and reused as needed. The main view objects that are used for this project include `MKMapView` that displays a map on screen and `UITableView`, useful for displaying an indefinite list of data such as a list of invites.
- **Controller** is a medium between the other two layers. It deals with updating the **Model** based on changes from the **View** and vice versa. The UIKit framework provides default classes to handle this layer, such as a `UITableViewController` class that updates a `UITableView`. A diagram showing how the three layers link together in the project can

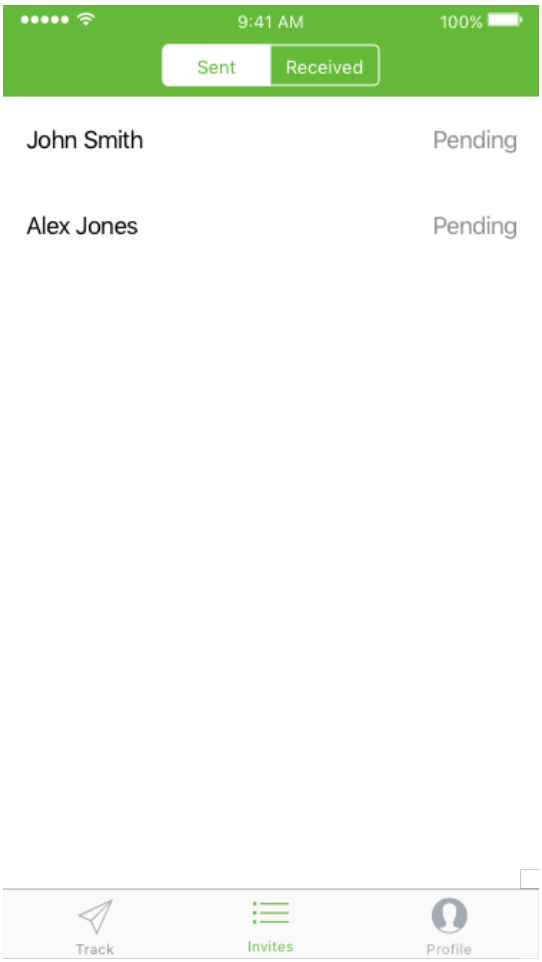



Figure 3.3: Redesigned mockup of the invites section of the application, using a segmented control in the navigation bar to switch between viewing sent and received invites.

be seen in Figure 3.4.

## 3.4 API Design

Given that the API dealt primarily with JSON because of its easy connectivity with Node.js and MongoDB, responses were returned in JSON. Each response contained a success boolean as one of the values in JSON, determining whether the request was successful or not. HTTP error codes were used to determine the nature of a response as well, but the success flag was useful for error checking within the API.

The following section discusses how the structure of the API is organised as well as how the data was split up in the database.

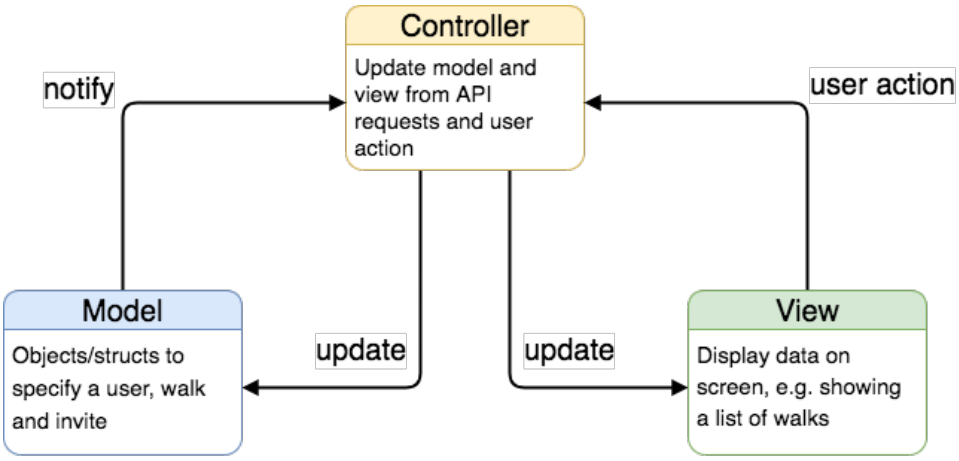



Figure 3.4: How the Model-View-Controller design pattern links together for this application.

### 3.4.1 Database Models

The data that needed to be stored in the database was split into multiple tables, or collections as they are called in MongoDB, to organise the data effectively. The full UML diagram showing each table's fields and how they link together can be seen in Figure 3.5. A description of each of the four main tables is as follows:

- **User** stores information about a user, created when they register. It contains cumulative values for the user's score, how far they have walked and how many steps they have taken. It also stores a user's device token, which is used to send a notification to a user's phone when inviting someone to go on a walk.
- **Invite** contains details of an invite sent from one user to one or more other users. Each user in the list of invite recipients contains a boolean flag to indicate whether that user has accepted the invite.
- **Achievement** is a fairly basic collection that stores the type of achievement that was gained from tracking a walk as well as how many points were gained from that achievement. The gamification and achievement aspect of the project is discussed in more detail in Section 4.9.
- **Walk** stores the details and representation of a walk, given a list of coordinates. It contains a reference to which user(s) tracked the walk, along with what achievements were gained.

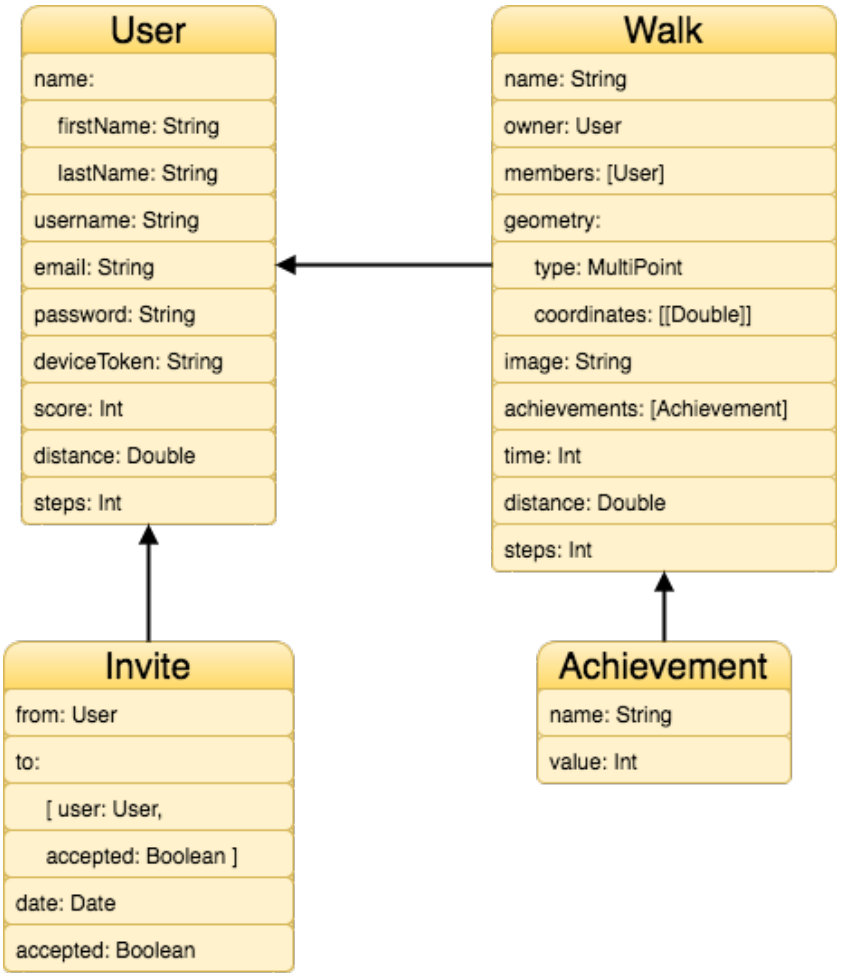



Figure 3.5: Diagram of the different database models used in the project. Arrows indicate where a field in a table references another table.

### 3.4.2 Endpoint Structure

The REST architectural style dictates that each element in the web service uniquely identifies one resource. Based on this, I chose to create a number of endpoints that each provided access to a resource on the server. An endpoint is a reference to a uniform resource indicator (URI) that exposes a resource on the server, given a specific HTTP method. These endpoints were grouped into similar categories, where each category is known as a route, to provide clarity to the client. For example, all endpoints that update user information are grouped under the `/users/` route, with specific user endpoints a subpath of this route. The diagram representation of all of the routes used for the API is shown in Figure 3.6.

Each endpoint in the API followed the CRUD functions – standing for Create, Read, Update and Delete. These functions are used to define what operation is being performed when changing

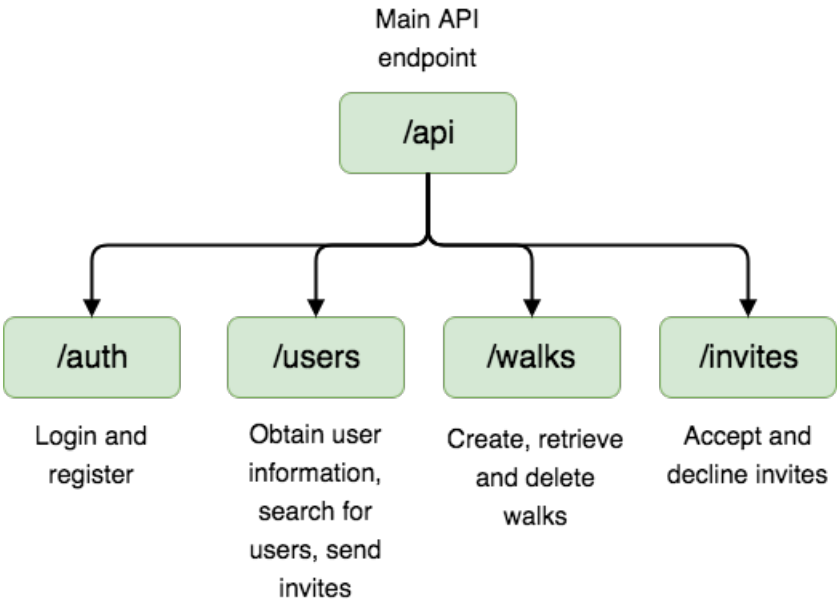



Figure 3.6: Diagram of each of the routes designed for the API. Each route contains a description of what endpoints exist within that route.

data in the database. These functions map to HTTP methods very well, which can then be used in each endpoint to clearly state what operation is used. For example, an endpoint that deletes certain data from the database should adopt the *Delete* function of CRUD and therefore also use the DELETE HTTP method.

### 3.4.3 User Authentication

For a user to be able to save their tracked walks and invite other users to go on a walk, an authentication system needs to be used. There are two types of authentication that can be used to identify a user when making a request to the API:

- **Basic authentication** requires a username and password, which are compared against the values stored in the database and authenticates the user if these values match.
- **Token-based authentication** authenticates the user through a unique key given to the client by the API, and can restrict access to certain functions of the API depending on that user.

Basic authentication alone is the only method necessary for an application to support user authentication and multiple device registration. The best practice however is to use basic authentication along with token-based authentication to prevent passwords having to

be re-entered. For example, when just using basic authentication, any restricted API request would require the user to enter their username and password again, which can get extremely tedious. When using token-based authentication, this key can be stored on the client-side when it is first received upon login or registering an account, and can then be used in subsequent API requests without the need for re-authentication.

A way to secure the authentication system even more is to create access tokens using a JSON Web Token (JWT). These are tokens that are digitally signed and encoded, meaning that source of the data can be verified when sending a request from the client to the API. JWT is also implemented in Node.js and can therefore be both verified and signed directly from the API. There are alternatives to using JWT, such as services like OAuth that better deal with user sessions, however the overhead required to set up OAuth would not be worthwhile.

Using both basic and token-based JWT authentication, the authentication system of the API could be designed. When the client sends a request to login using their username and password, basic authentication is used and a JWT is returned by the API if the authentication is successful. The returned JWT is then stored by the client and if any API requests specify that authorisation is required, the JWT is sent in the *Authorization* header of the request.

### 3.4.4 File Storage Methods

There are two methods that can be used when uploading images via an API to a file storage service such as AWS S3. The first of which is direct upload where the client – the mobile app in this case – uploads the file directly to S3, bypassing the API altogether. The other option, known as pass-through upload, is to upload the image via a request to the API and then let the API handle the upload to S3. The former of the two options is the most commonly used since it does not require any extra processing to be performed by the API, which could result in slower response times especially on Heroku.

While the direct upload method does also require a slightly more complicated implementation, I decided to choose it based on the needs of this project and its ability to be scaled effectively.

## 3.5 Summary

The design section outlined the elements of the project that needed to be completed before any implementation could begin. The basic user interface design of the main screens of the mobile application were created using the MVC design pattern, allowing for front-end development to begin. Meanwhile, the structure of the database and endpoints in the API meant that implementation in the back-end could begin by using the technologies chosen in Section 2.4.

# Chapter 4

# Implementation

## 4.1 Overview

Since the back-end API and front-end mobile app are so tightly connected, implementing them linearly would not be logical as errors between the app and server will not be detected as quickly and requirements of features may change over time. I therefore implemented the two in parallel, normally a feature at a time, so that a working implementation for each subsequent feature was produced at the end of each iteration.

This chapter splits the implementation by feature, with the order presented matching the chronological order of the project.

## 4.2 Endpoint Routing

To set up endpoints on the API to respond to client requests, the popular Express web framework [26] was used. Express's router object determines how the API handles a request to a certain URI with a specific HTTP method.

The main file loaded when a request is made, `app.js`, specifies the main endpoint for the API and links this endpoint to an Express router. This router specifies the path for each of the main API routes, following the design from Figure 3.6 in the previous section. Each path contains a router that matches each of the endpoints it serves as well as a controller containing functions, known as middleware functions, which are executed when a router endpoint is matched from a request. An example of how a request to login a user is routed through the API is shown in Figure 4.1.

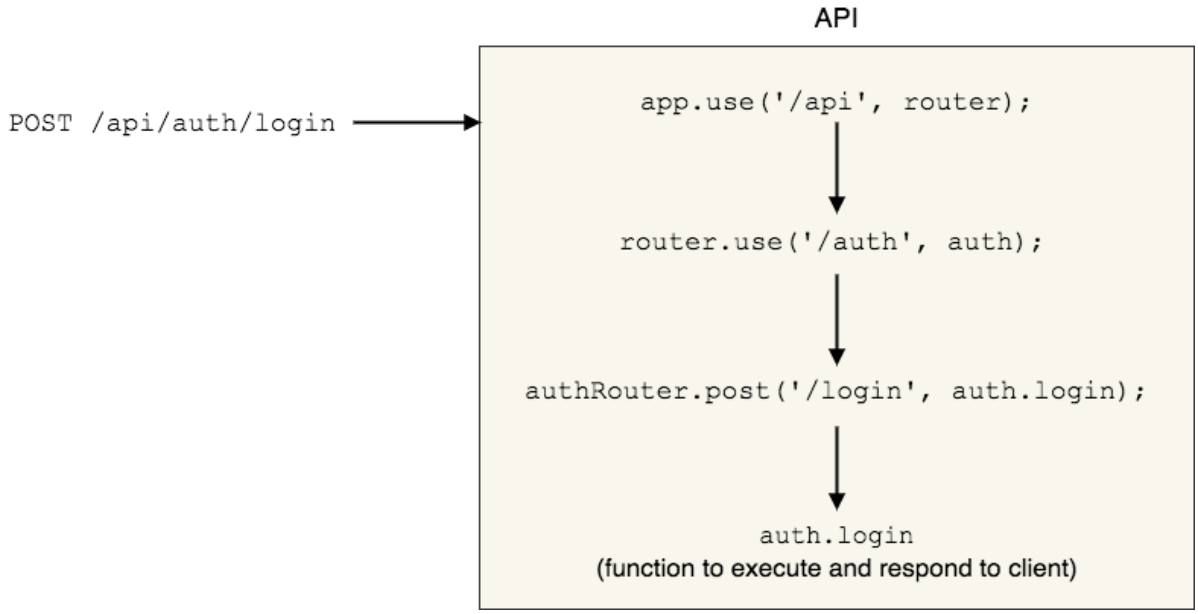



Figure 4.1: Example of how a login request in the API is routed using Express, with the desired endpoint function eventually being called to execute the login code and return a response to the client.

## 4.3 Communication with API

To handle requests from the multiple APIs used within the mobile application, a singleton class `APIManager` was created that was accessible throughout the app to organise each API call. The main function of `APIManager` was to abstract the network requests through the HTTP networking library Alamofire – a more elegant way to handle networking in comparison to Swift's default network tools, providing simple JSON encoding and serialisation as well as response validation.

Alamofire's Router design pattern allowed me to define a complete set of paths, methods and parameters needed for the endpoints of a particular API and hence construct a request with with any HTTP headers as appropriate. The Router was implemented as a protocol containing the necessary fields that needed to be overridden, and each API that needed to be documented in this way adopted this protocol and used an enumeration pattern to define each of the available endpoints for that API. The specific endpoint case for a given API router would then be passed as a parameter to Alamofire's `request()` method, which would construct and send the HTTP request as necessary.

Finally, an enumeration was created to represent the success or failure response received from the API. Enumeration cases in Swift can contain parameters and so I defined the success response to contain a JSON object that contained the response from the API, constructed in

Alamofire's asynchronous request callback. Meanwhile, the failure case contained a Swift error object whose error code and description were populated from the information received in the API response. Figure 4.2 explains in more detail the path of method calls and how data is passed through callbacks when making a network request.

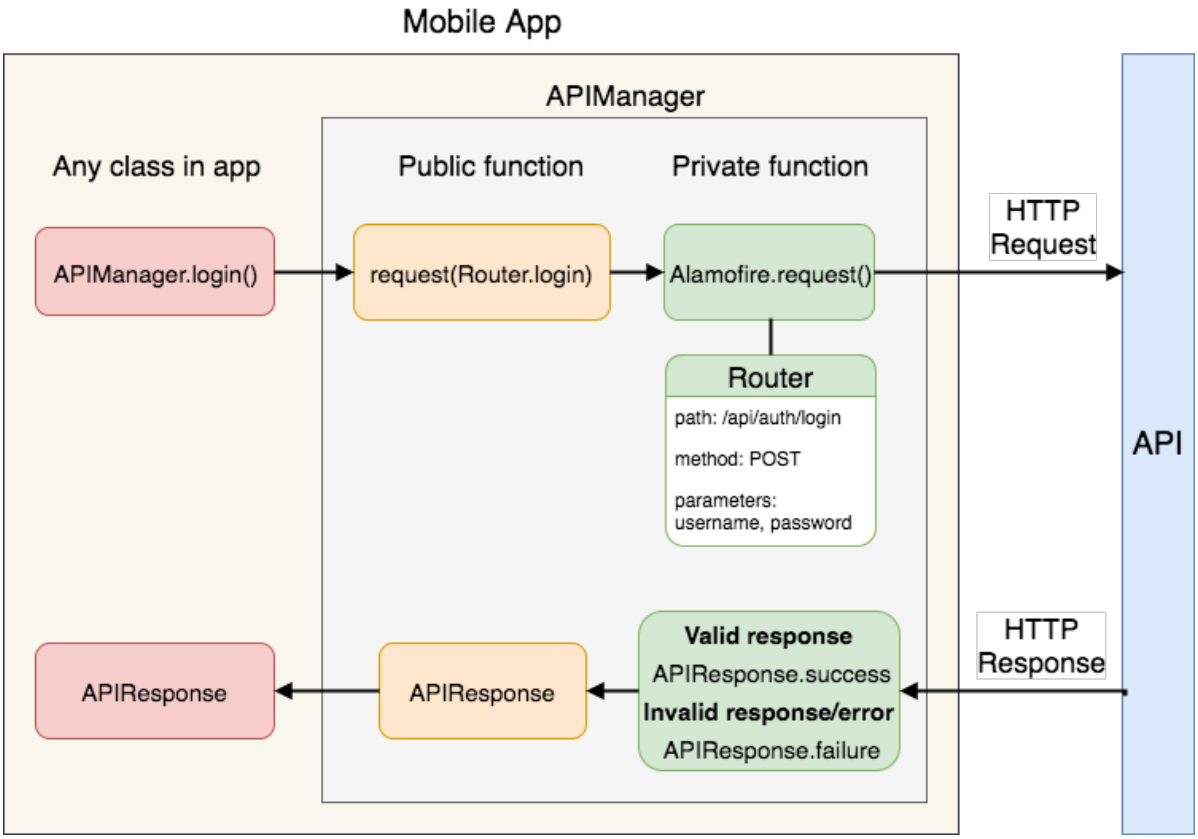



Figure 4.2: The path of method calls and callbacks used when making a network request within the mobile app, namely logging the user in. The top row of arrows represent method calls being made from the app whilst the bottom row of arrows represent the callbacks for each method once the network response is received.

## 4.4 Authentication

The bulk of the authentication system for the project needed to be implemented in the back-end. Passport [27] is a popular library that provides authentication middleware for Node.js. It supports multiple types of authentication – each of which is implemented as a *strategy*. Based on the request, the middleware can then call `passport.authenticate()` passing the desired authentication strategy as a parameter.

### 4.4.1 Local Authentication

For this project, both basic (local) and JWT authentication strategies needed to be implemented. The local strategy function is called when a `POST /auth/login/` request is received. This function first checks whether a user with the given username exists in the database. If a user does exist, the password given is compared to the password stored in the database for that user. Passwords are hashed and salted before they are inserted in the database using the bcrypt library, which also provides a function to compare a plaintext and hashed password used in the local authentication strategy. Given that the passwords match and the login is successful, a JWT is signed using a private key, with the payload of the token containing the authenticated user object. This JWT is then returned to the client in the HTTP response message.

On the client-side, if a successful response is received from a login request then a singleton app-wide User object is created, which is then saved to the iOS keychain – a secure persistent storage location in the operating system. Each time the mobile app is launched, the keychain is checked to see if a User object is stored, indicating if the user is logged in. If an object is found, the main screen of the app is shown allowing the user to track walks, otherwise the initial authentication screen is displayed.

### 4.4.2 JWT Authentication

The documentation for the API specifies which requests require the user to be authenticated using JSON web tokens. The function to implement the JWT authentication strategy in the API looks for the token in the *Authorization* header of the request in the form of `JWT token`. To check whether the JWT is valid, the user's ID is extracted from the token and searched for in the database. The Router for the back-end API in the mobile app contains a boolean variable `requiresJWTAuth` to specify which enumeration cases, each representing an endpoint, requires authentication. When this variable is true, the JWT is added in the header when constructing the request as described above.

Server-side validation checks can then be performed, depending on the request, to determine if the user has permission to perform the requested operation. For example, users should only be able to delete walks that they have tracked and not be able to delete any other users' walks. Therefore when a `DELETE /walks/:walkID` request is sent, the walk – referenced by its ID in the *walkID* parameter – is checked to see that it has been tracked by the user obtained through the JWT. If the users match, the walk is deleted, otherwise a **401 Unauthorized** error is returned.

## 4.5 Tracking walks

Implementing the walk tracking component of the app was a main part of the project and the foundation that other parts of the app could be built upon. As was designed, when the user chooses to start tracking a walk a map is displayed showing the route from the user's location and statistics about the walk currently being tracked.

### 4.5.1 Obtaining user location

A location manager, part of the Core Location framework of the iOS SDK, was used to determine the geographical location of a user. The class to implement this is `CLLocationManager`, which requests permission to use location services when the user's first opens the app. The integration of the Core Location framework in iOS means that the user's location obtained through the location manager can be displayed on an `MKMapView` by simply setting its boolean instance variable `showsUserLocation` to true.

`CLLocationManager` has a delegate method `didUpdateLocations()` that is automatically called by the operating system every time new location data from the user is available. This method passes in an array of `CLLocation` as a parameter, the class which represents the location data generated by a location manager. A `CLLocation` object stores the coordinate of a location by its latitude and longitude, and other details such as the altitude and position accuracy. When this delegate method is called, the locations array parameter is appended to a class variable `locations` to record a history of a walk.

### 4.5.2 Calculating walk statistics

The distance of a walk, stored as a floating point variable in the class, is also calculated and updated in `didUpdateLocations()`. `CLLocation` provides an instance method to return the distance in metres between the receiver's location and another location. For each new location that is passed into the `didUpdateLocations()` method, the distance is calculated between that location and the last location in the global locations array, which is then added to the global distance variable. This global distance variable therefore keeps track of the distance between all of the locations in the global locations array.

With the route of the walk and its distance now being logged, the UI had to be updated to display this information. To display the statistics about a walk, a timer was employed that fired every second. The duration of a walk was stored in seconds in a global class integer variable. Each second when the timer fired, this global time variable is incremented by one to track the duration of the walk. The time and distance variables are then formatted and displayed on the screen.

### 4.5.3 Drawing the map route

To display the walk route as a line on the map, I had to use the `MKPolyline` class in the MapKit framework. An `MKPolyline` is a shape that can be constructed via one or more coordinates. In `didUpdateLocations()`, a polyline is constructed between the last coordinate in the global locations array and each location from the list of new locations parameter. This polyline is then added to the map by using the `MKMapView add()` instance method. An `MKMapView` delegate method was also used to adjust the colour and thickness of the line drawn on the map.

### 4.5.4 Saving tracked walks

Once the user chooses to end their current walk, they are presented with an option to save the walk to their profile. A popup appears prompting the user to enter a name for the walk, which is used to identify the walk on the user's profile. After a name has been entered, a `POST /walks/create` request is sent to the API to store the walk in the database. This request accepts a number of fields to define the walk as parameters in the request body. The map route, implemented by the array of `CLLocation` mentioned previously, is sent in the request as a JSON string representing an array of arrays of floating point numbers, with each array element corresponding to a coordinate in the form of longitude and latitude, like so:

```
[[-0.174877, 51.498800], [-0.174523, 51.497810]]
```

This JSON string is then parsed by the API and stored in the database using the GeoJSON format [28], a specially designed format used to encode geographical data structures. An array of coordinates used in this project can be formatted in GeoJSON using the `MultiPoint` geometry type.

## 4.6 Querying the Database

Every endpoint in the API either queries or updates the database. Because of this, calls to the database must be easy-to-use and available throughout the API. The object modelling library Mongoose that was used provided an abstraction between the database and the objects defined in the API. Schemas can be defined that directly map to MongoDB collections, allowing me to define the fields of the database and their type following the UML diagram from Figure 3.5.

Mongoose handles object validation automatically. When saving an instance of a schema to the database, Mongoose checks that the type of each field matches the type defined in the schema. If the types do not match, a validation error is returned. In order to format this error and return it to the client in JSON format, a helper function was defined that received an error

object as a parameter. The function iterates through the errors in the error object and finds the appropriate error message. This message is then returned to the client in JSON format along with a **400 Bad Request** status code.

When dealing with referencing other collections, as is required in order to implement the UML diagram, object IDs are used. Each document in a MongoDB database is identified by a unique 12-byte hexadecimal value that is assigned to the *_id* field of the document when it is created. These object IDs can be stored in a document to reference another document in the database, without the need to store a key that links the two documents as with a MySQL database. Mongoose provides convenience functions to find a document by its ID, which is useful in this project as IDs could be returned to the client and sent in another request, such as retrieving a walk's details given its ID.

### 4.6.1 Query Population

Mongoose's query population feature was also utilised in the API. Population is the process of replacing object IDs in a document with with the full referenced document from another collection. Given that a field is of type `ObjectId` and the schema the field references is specified by the *ref* keyword, that field can be populated by using the `populate(field)` function.

This method was employed when retrieving a user's sent and received invites. If a user's received invites were returned to the client without population, the information would be useless as the invitation sender would only contain their object ID. Hence, the sender and recipients of an invitation in the *Invite* model are referenced to the *User* model. The code in Listing 4.2 shows the *from* field defined in the *Invite* schema model. The field is declared to be of type `ObjectId` and reference the *User* model, meaning that during population Mongoose will try and replace the object ID in the *from* field with the full user document it references.

```
1       from: {
2         type: Schema.Types.ObjectId,
3         ref: 'User',
4         required: true
5       }
```

Listing 4.2: Extract of code taken from the *Invite* schema, showing the definition of the *from* field, which is of type `ObjectId` and has a reference to the *User* model.

These fields are then populated when retrieving sent and received invites in the respective endpoints. The code in Listing 4.3 is taken from the function called when a `GET /users/invites/sent` request is made, which returns a list of the user's sent invites. Here consecutive function calls are made to the *Invite* model: `find()` to filter invites that were sent by the user making the request (verified by the JWT), `populate()` to populate the

recipients with their full user documents, and `exec()` to execute the query asynchronously, returning invites to the client with a **200 OK** status code.

```
Invite.find({from: req.user._id})
      .populate('to.user')
      .exec(function(error, invites) {
        if (error) return helper.mongooseValidationError(error, res);

        res.status(200).json({
          success: true,
          invites: invites
        })
      });
```

Listing 4.3: Extract of code taken from the function called when a `GET /users/invites/sent` request is made, detailing how the population method is used to return a full list of users to client.

## 4.7 Storing images on a server

Having chosen to use the direct upload method described in Section 3.4.4, I outlined the steps that needed to be followed when uploading a file:

**Step 1** Request a pre-signed URL from the API to upload a file using the AWS SDK, giving the client the necessary permissions to upload at that location.

**Step 2** Upload the file directly from the client to the URL.

**Step 3** Store a reference to the URL where the file is stored either locally or in the back-end so that it can be accessed in the future.

Even though the only current use for the file upload capability in the app is to store the thumbnail images of tracked walks, I wanted to make the code as re-usable as possible so that any feature implemented in the future could upload files with ease.

To implement **Step 1**, I created an endpoint in the API at `/walks/create/upload` to retrieve a signed URL from AWS S3. This accepted a GET request with no parameters and returned a URL that would expire in 60 seconds. The expiry time was used so that numerous empty URLs could not be created by continuously sending a `GET /walks/create/upload` request, while still allowing enough time for the file to be uploaded by the client.

Upon receiving the URL from the API, the mobile app then uploads the file to this location as in **Step 2**. Alamofire provides an `upload()` method along with its `request()` method that supports sending of larger amounts of data such as a file. The method accepts the file as a parameter in `Data` format, a datatype commonly used throughout iOS to store and transfer files or objects. Images can be converted to their Data representation using the `UIImageJPEGRepresentation()` function declared in Apple's UIKit, passing in a native `UIImage` object.

While the previous two steps have been fairly general to upload a file, **Step 3** of the method is what applies this code to the feature being implemented. In the case of this project, I needed to upload an image of a tracked walk and store a reference to the URL of that image in its own *Walk* document in the database. Once the image has been uploaded successfully, a Walk document is created via sending a `POST /walks/create` request to the API. The parameters of that request include the name of the walk, an array of coordinates and the URL of the image that was uploaded in the previous step.

The three steps described above are illustrated in Figure 4.3.

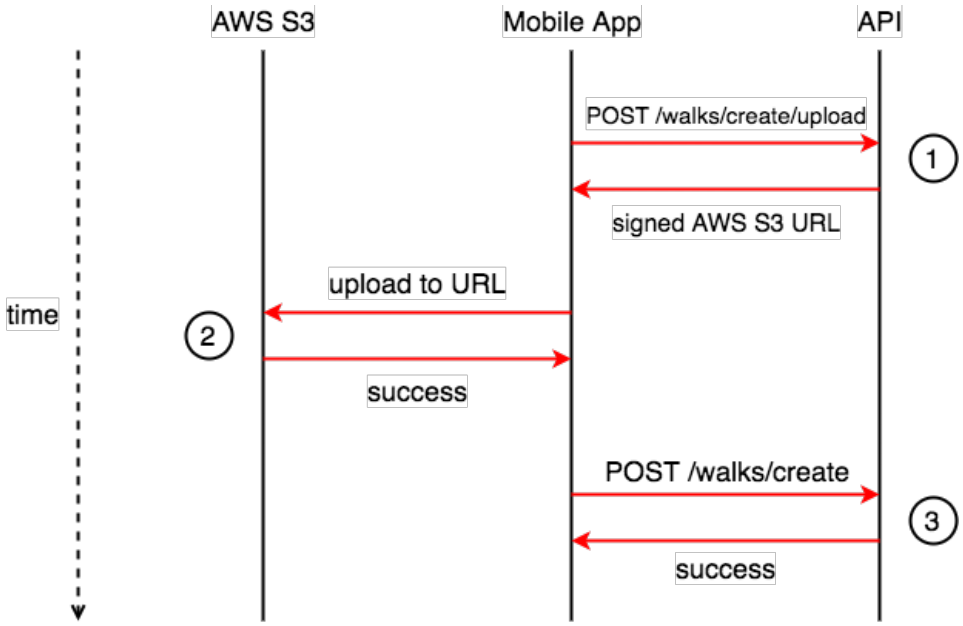



Figure 4.3: The three steps used in this project to upload a file.

## 4.8 Displaying points of interest

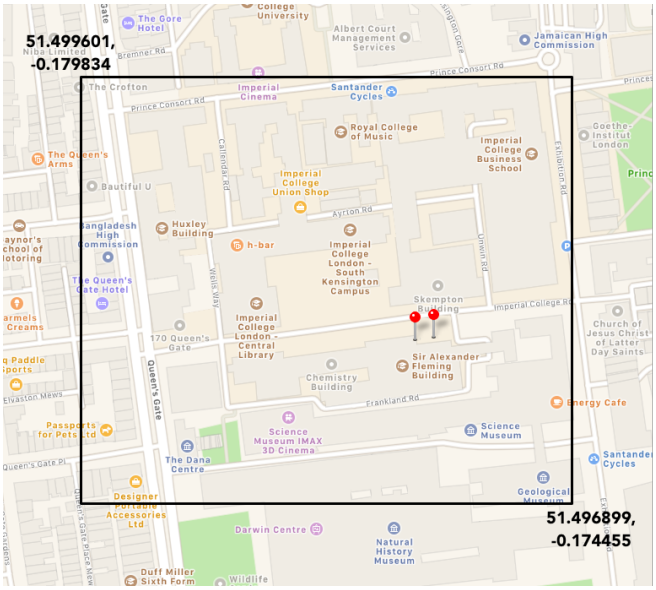
In order for data about the plaques to be retrieved, another Router had to be defined in the mobile app. Three endpoints were required for this router: one to gather a list of plaques inside

a given coordinate region, one to request detailed information about a plaque given its ID, and one to request information about the related person(s) or objects of a plaque.

The OpenPlaques API is able to return data in a number of formats including JSON, which made it easy to parse responses since JSON had already been used previously in network calls. An area can be queried in OpenPlaques to find what plaques are present by specifying the top-left and bottom-right coordinates of the area's bounding box. The JSON returned contains a list of details about the plaques, with each element containing the plaque's unique ID, coordinates and inscription. For example, to query the plaques near Imperial College, a request would be made using the following URL

```
http://openplaques.org/plaques.json?box=[51.499601,-0.179834],[51.496899,-0.174455]
```

which would result in the bounding box in Figure 4.4 being queried and the JSON returned for the two plaques found.





```
[
  {
    "id": 10332,
    "latitude": 51.49794,
    "longitude": -0.17565,
    "inscription": "Sir Geoffrey Wilkinson
FRS (1921-1996) 1939-1943 student,
1956-1996 professor, Imperial College.
1973 Nobel Laureate for pioneering studies
on organometallic compounds"
  },
  {
    "id": 10331,
    "latitude": 51.49794,
    "longitude": -0.17575,
    "inscription": "Sir Derek Barton FRS
(1918-1998) 1938-1942 student, 1957-1978
professor, Imperial College. 1969 Nobel
Laureate for new concept of organic
conformational analysis"
  }
]
```


Figure 4.4: Example request to search for plaques near Imperial College, showing the bounding box for the request (left) and the JSON returned for the plaques retrieved (right).

To display the plaque data on a map during a walk, repeated requests were made to the OpenPlaques API while the user was walking. A request is made every 60 seconds to search for plaques in an area of 500 square metres with the centre at the user's location. The time interval and area span values were chosen to reduce the number of requests made to the API and also to ensure that the user could not walk into a new area before the new area request had been made.

An array of `Plaque` structs was used to store the data retrieved from the bounding box API

request. While not all data is returned from said request, the coordinates enabled me to display all of the plaques as pins on the map by creating a subclass of `MapKit`'s `MKPointAnnotation` class for each plaque and calling the `MKMapView` instance method `addAnnotation()`.

## 4.9 Gamification

The structure of the points system first needed to be implemented in the API, which the mobile app could then conform to. An object was defined as an enumeration in the back-end to represent the possible achievement types that could be gained from tracking a walk. Based on the chosen achievements from Section 2.2.2, the values that this object contained were `DISTANCE`, `DAY_STREAK` and `GROUP`.

Upon creating a walk via the `POST /walks/create` endpoint, each achievement in the achievements parameter is checked against the list of valid achievement types. A document is created in the database for each valid achievement type and its value, which are then stored as an array in the corresponding *Walk* document. The values from each of the achievements are summed and added to the score of the user(s) that tracked the walk. If one of the achievements is not valid, a **400 Bad Request** error is returned to the client.

In the mobile app, a similar enumeration is defined to match the valid achievements listed in the back-end. When the user is saving a walk, the app checks whether there are achievements that the user should earn for the walk currently being saved. The achievements are sent to the server as a JSON string representing an array of dictionaries – each element in the array a valid achievement with dictionary keys *name* and *value*, referring to the achievement type and number of points scored respectively. An example of a list of achievements sent in a request is as follows:

```
[{"name":"DAY_STREAK","value":400},{"name":"DISTANCE","value":100}]
```

### 4.9.1 Day Streaks

While the implementation details for the distance and group achievements are fairly self-explanatory, checking for a streak of consecutive days required a little more thinking. We must store two values in persistent storage to achieve this: the last date the user tracked a walk and the number of days in the streak (the streak count). I used `UserDefaults` to achieve this – a core class in iOS that stores data in key-value pairs, commonly used for storing app settings.

When saving a walk, the values are retrieved from `UserDefaults`. The saved date is compared with the current date using the core iOS function `dateComponents()`, which takes two dates as inputs and outputs the difference in time between them in a specified unit. If the

day component returned from this function is equal to 1, meaning the days are consecutive, the streak count is incremented by one – otherwise it is set to one to reset the streak. In both cases, the last saved date is overwritten with the current date and the new values are stored in `UserDefaults`.

## 4.10 Inviting users to go on a walk

As with other features of the project, the back-end implementation was completed first. For walk invitations, four endpoints had to be created:

- `POST /users/invite` creates a new Invite document in the database with the sender as the user sending the request (verified by the JWT) and the recipients specified as a parameter in the request body. It also sends a push notification to recipients' iOS devices if possible, which is discussed in more detail in Section 4.10.1.
- `GET /invites/:inviteID/accept` marks the invite with the specified invite ID as accepted for that user, given that the user has permission (if the invite was sent to them).
- `GET /invites/:inviteID/decline` declines the specified invite given the user has permission, deleting the invite document from the database.
- `GET /invites/:inviteID/start_walk` notifies the API that the sender has started the walk for a given invite. Due to time constraints, the current implementation simply deletes the invitation from the database so that it does not remain in the list of a user's received invites, however in the future I would like to implement a live tracking system as discussed in Section 6.2.1.

In the mobile app, the invites tab was implemented using a table view and a segmented control as designed in Section 3.3.1. Depending on the selected segment in the segmented control, the table view was used to display either the user's sent or received invites. When the user selects a different segment, a delegate method is called that pulls any new data from the API and refreshes the table view to display the correct data corresponding to the segment.

Custom table view cell classes were created to display both the sent and received invites, containing custom views and buttons depending on the invite type. Common attributes between the cells, such as the invite name and date, were collected into a superclass that both the sent and received invite cells conformed to. The two types of cells can be seen in Figure 4.5.

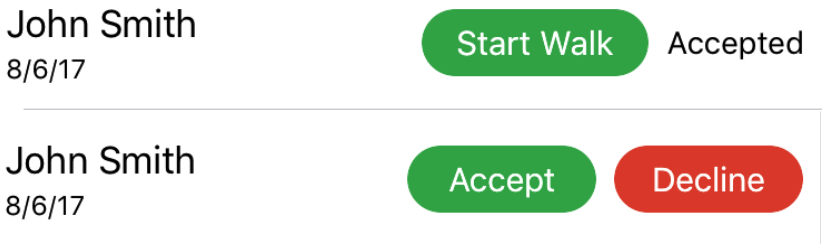



Figure 4.5: Two types of custom table view cells used to display invites – sent invite cell (top) and received invite cell (bottom).

### 4.10.1 Invite Notifications

When an invite is sent from within the app, a push notification is also sent to the device of each of the recipients. This was done using Apple's Push Notification Service (APNs) that enables data to be sent to a device in the form of a notification, given that they have the app installed.

Each iOS device is uniquely identified by a device token that is registered once a user subscribes for notifications within the app. Every time the app launches, we check whether this device token has been stored locally. If not, this may imply that the user is opening the app for the first time or has only just subscribed for notifications. In this case, we store the device token and make a request in the background to the API (`GET /users/register/:token`) to store the device token in the user's document in the database.

In the back-end, when a `POST /users/invite` request is made to send an invite an array is populated with the device tokens of all the recipients, if they have one. I then used the node-apn framework to interface with the APNs directly through Javascript. A notification is constructed and sent to each recipient, resulting in the notification received in Figure 4.6. Tapping on the notification will open the app where the user can respond to the invite.

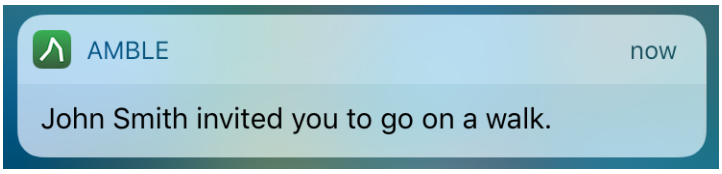



Figure 4.6: A example of a notification received when a user invites you to go on a walk.

## 4.11 Challenges Faced

Though the majority of the implementation was implemented as intended, there are some parts that posed a challenge due to its complexity or a third-party bug. This section discusses which

aspects of the implementation posed problems, and how these problems were avoided.

### 4.11.1 Using iteration in an asynchronous environment

Calls to the database from the API are, by nature, asynchronous. When a request is made to an endpoint, the database is queried and its callback function called after a certain period of time. A response is then sent to the client from within this callback function. However, when making asynchronous calls to the database inside an iterative loop, problems begin to arise. Since the requests are asynchronous, the loop will not wait for the response and immediately begin its next iteration. A response is then be sent back to the client straight after the loop has completed, possibly before any database queries have returned.

An example of this problem that surfaced in the API is when sending an invitation in the `POST /users/invite` endpoint. The list of recipients is iterated over, with each user ID looked up in the database to make sure it is valid – if it is, the user is added to a new array of valid users. The problem described above would cause this array to be empty after the execution of the loop due to the database lookup not having enough time to respond.

To fix this, we need to convert the for loop to execute asynchronously. I used the async framework, which provides a number of asynchronous versions of synchronous functions. The function that I required was `forEach()`, providing a way to iterate over an array in parallel. Each element in the array is given a callback function to be called when computation of that element has finished – in this case when the database query has returned. A main callback function is also specified, which is called when all callbacks from the elements have been received. For sending an invite, the main callback function creates an invite using the valid users array, which is now guaranteed to be populated due to the asynchronous iteration.

### 4.11.2 Travis CI Errors

There were problems when using the continuous integration tool Travis CI that caused builds to fail, primarily when testing the mobile app. To run iOS tests from the command line, you can use the `xcodebuild` command with parameters to specify the codebase being tested and the version of simulator to run the tests on. The command occasionally fails with no error message when running on Travis, meaning that tests were unable to be executed.

I discovered that the error appears to be well-known, with an issue raised about it on the Travis CI GitHub repository [29]. Following some of the suggestions discussed on the issue page, the most-used option seemed to be to use the `travis_retry` command to retry the test command three times. Since the error was so sporadic, this greatly reduced the chances of a build failing with this error.

## 4.12 Summary

The implementation segment of the project was by far the largest, with each feature requiring code to be written in both the front and back-end. The structure of the API and skeleton application were first created, followed by API communication and user authentication. More feature-specific aspects of the implementation were then completed following the requirements set out in Section 3.2. These included tracking and saving walks using `CLLocationManager`, displaying points of interest on a map by querying the OpenPlaques API, and setting up a system to invite others to go on a walk.

# Chapter 5

# Evaluation

Evaluation must be conducted throughout the project to assess both the technical quantitative aspects of the project so that the project functions correctly and the qualitative aspects, such as the design and ease-of-use of the produced application.

Additional evaluation is then performed near the completion of the project in order to assess how well the project's objectives were achieved and whether the project can be considered a success.

## 5.1 Software Validation

Throughout the project, I ensured that each feature of the application was well tested to minimise bugs that could arise later on in the project. When implementing a feature in the project, the back-end element was normally completed and tested first to guarantee that when the front-end implementation was taking place I would be working with a fully-working element from the back-end. Testing was performed using the continuous integration tool Travis CI chosen in Section 2.4.2. This meant that tests were run automatically when commits were made to Git, allowing me to see at exactly what point any tests failed.

### 5.1.1 API Testing

The purpose of API testing is to ensure that all endpoints function correctly and handle any erroneous input as expected. Mocha [30] was the main testing framework used to test the API. It supports asynchronous callbacks, which is necessary for making calls to the API in test cases. Each test case is identified by a unique string to differentiate between each case and recognise which test cases have failed.

To make the actual API requests in a test case, the Supertest framework was used, allowing for assertions of HTTP requests. The structure of one of the test cases can be seen in Listing 5.4. The `describe()` and `it()` functions specify the structure of the tests and uniquely identify each test case respectively, with the latter providing a `done` function (line 3) to be called once the test case has finished executing its request. The HTTP request is made on line 4, where `request` is a reference to the Supertest dependency, specifying the HTTP method and path (line 5). Subsequent functions are then called on this request to make assertions on elements of the response using the `expect()` function (lines 6-11). In the example shown, these assertions include checking the response is of JSON type, checking that the HTTP response code is **200 OK** and checking the number of users returned is zero (since this particular request searches for a user with an invalid name).

```
1  describe('GET /search/:userInfo', function() {
2    describe('Valid user search', function () {
3      it("should return empty array with name not found", function(done) {
4        request(app)
5          .get(uriPrefix + '/search/invalid name')
6          .expect('Content-Type', /json/)
7          .expect(function(res) {
8            res.body.success.should.be.equal(true);
9            res.body.users.should.have.length(0);
10         })
11         .expect(200, done);
12     });
13   });
14 });
```

Listing 5.4: Structure of a test case for the API

The file that these test cases are written in is listed in the project's configuration file, so that this test file is automatically executed when the default `test` command is run. When commits are made to Git, Travis CI then automatically runs the the `test` command, indicating which test cases passed. If a test case failed, I was able to pinpoint the exact part of the endpoint that was affected due to the the unique names I gave to each test case. This was invaluable in speeding up the error fixing process.

### 5.1.2 Mobile App Testing

Testing the front-end mobile app was not performed in as much detail as the back-end, mainly because of the prioritisation of user testing and concentration on the implementation at hand. With that being said, some parts of the logic and UI of the application were tested thoroughly. To

create a behaviour-driven development testing environment and provide English-like assertions, the Quick and Nimble frameworks were used respectively, so as to match the way in which the API was tested.

Calls to the API were tested to ensure that responses were handled properly in `APIManager`. In order to prevent unwanted requests to the API during testing, some HTTP requests were stubbed using the Mockingjay framework and mocked responses were returned to the app instead. For example, when making a valid request to register a new user, the actual API request is not made to prevent a superfluous record from being created in the database. In this case, a mocked response is returned containing a **200 OK** status code and the emulated successful response that would have been returned from the server. For invalid requests, no database records are ever created and so HTTP requests do not need to be mocked – instead the actual error response returned from the API can be used in a test case.

To make sure that different components within the app worked together correctly, integration tests (or UI tests as they are known in iOS) were written. UI tests execute a sequence of operations within a simulator and use assertions to make sure each operation executes correctly.

Thorough testing of the walk tracking feature was conducted to make sure that walk routes were accurate and points of interest were correctly displayed on the map. While testing by actually walking in the real world was extremely important, I was not restricted to this. Xcode provides a feature to simulate your location while running the app, either in the iOS simulator or on a real device. There are a number of default routes defined in Xcode to simulate a location route around California, however these were not so useful as not many points of interest were found there.

You can also define your own routes to use if you wish. The routes are stored as a list of coordinates in a GPX file that Xcode can then parse. I discovered a website [31] that converts a route from Google Maps into a GPX file. I was able to use this tool to create routes in London where there are a large number of plaques, which was extremely useful for testing.

## 5.2 User Testing

I aimed to provide users with versions of the application as early as possible and often throughout the project so as to gain feedback frequently and iterate the application multiple times based on this feedback. The feedback that was received from users ranged from bug fixes in the application to subjective views on features of the app that could be improved or changed.

### 5.2.1 Bug Fixes

The following section lists some of the bugs that were found in my code based on feedback from testing my application in the real world, either by myself or from other users that I provided the app to. For each error that was found, the reason that this error occurred and the solution that fixed the error is also listed.

**Error:** the app crashes when making any network request with no internet connection.

- **Reason:** every time a network request is made in `APIManager`, the status code of the response is obtained to see if an error has been returned. However, if there is no internet connection, there is no response from the request and so the status code is empty. This results in the Swift equivalent of a null pointer error, causing the app to crash.
- **Solution:** the status code of the response is only used if the response itself is not null, otherwise a default network error is returned.

**Error:** the app crashes when creating an account after scrolling the text fields off screen.

- **Reason:** table view cells in iOS are reusable, meaning the cells and their associated data that are not on the screen are not in memory and are recreated when they reappear on the screen. When pressing the button to create an account, the data the user entered from the text fields in the table view cells is retrieved and passed to the `register()` function in `APIManager`. If one or more of the cells is not on the screen, its data does not exist due to its reusability and so a null pointer error occurs when trying to obtain the value passed into `register()`.
- **Solution:** instead of retrieving the data from cells when pressing the register button, a global class array was used to store the data from the cells as the user is entering it. This means that even if some cells are not on the screen when registering, the data is still stored previously and can be passed to the `register()` function without any danger of any parameters being null.

**Error:** when tracking a particularly long walk, the image produced contains rendering issues where the map route is blurred, as shown in Figure 5.1.

- **Reason:** the walk route image was produced by rendering an image of the visible map view on the screen using Core Graphics – a framework used in iOS to perform 2D drawing. To do this, the map view needs to be zoomed out so that the entire walk route fits in the view. In doing so, the polyline used to draw the walk route on the map, discussed in Section 4.5.3, did not always render properly for reasons that I was not able to discover.

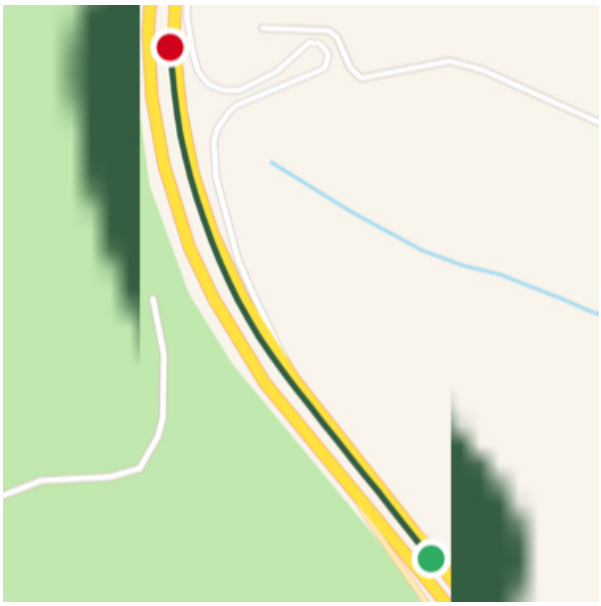


Figure 5.1: A bug in the app causing a walk's route to not render properly.

- **Solution:** the `MKMapSnapShotter` class, part of the MapKit framework, was used to render an image of a map rather than using Core Graphics. By specifying a coordinate region and various other options, an image is rendered of that area of a map. The map route still had to be rendered separately onto the map afterwards using Core Graphics, but I found this was the best solution at the time to avoid the rendering issues presented by the previous method.

### 5.2.2 Final Survey

When the implementation phase of the project was completed, I conducted a survey about the app as a way to try and quantify the qualitative aspects of the project that needed to be evaluated. These qualitative aspects include the ease-of-use of the application, its design and how useful each feature is to the user. The full results of this survey can be seen in Appendix C, with a summary of results discussed below. Any question where the user was expected to rate a feature's usefulness, a linear scale from 1 to 5 was given to choose from, with 5 being the most useful and 1 not useful at all.

The survey showed that both design and ease-of-use of the application appealed to the user, with no responses for either category giving a negative response (less than 3). The main questions asking the user whether they were encouraged to walk more when using the app gave mainly positive results, with 73% of people giving a score of 3 or 4. The rating of features gave mixed responses, with the controversial features being tracking walks and walk invitations. Features to view points of interest and view walks on your profile provided the best responses

in the survey, with all users giving a score of at least 3.

An interesting point to discuss is the opinion of users who had used previous walking or fitness apps compared to those who hadn't. From the graph in Figure 5.2, we can see that the users who have used previous fitness applications gave lower scores for the tracking walks and statistics features of my app. This is understandable since the apps that these users have used, the most popular being Strava, are dedicated fitness and running apps that have taken years to develop whereas the tracking of walks in my application is only one element.

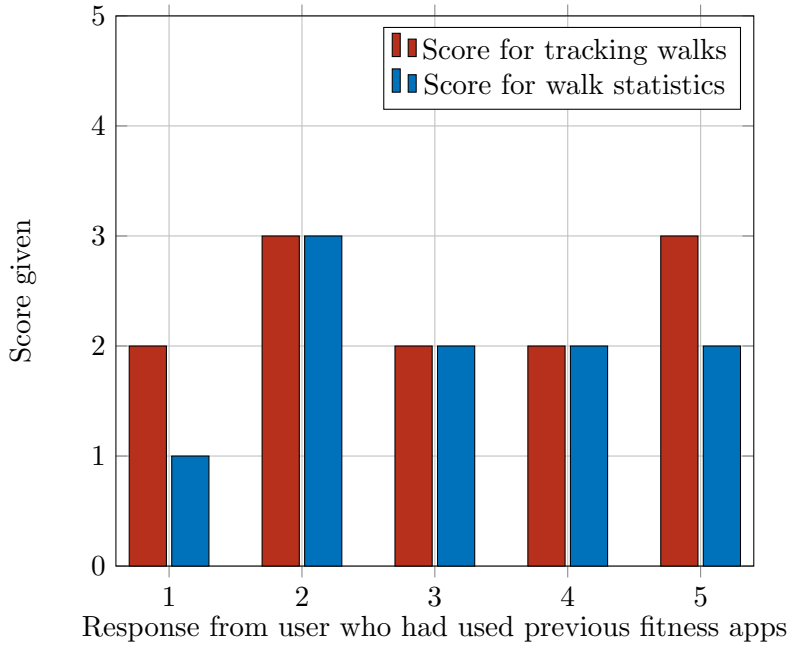



Figure 5.2: Graph showing the scores for the feature ratings of tracking walks and viewing walk statistics, given by users who had previously used walking or fitness apps.

Additional comments were left by users at the end of the survey. The main points written were that other apps were better for fitness and provided more detailed statistics, and other points of interest should be used instead of just historical plaques. I also conducted hallway testing, where I asked strangers to give their opinion on my application having no prior knowledge of it. The results from hallway testing agreed mainly agreed with the results of the survey, with people liking the design of the app and the main concept, but some stating that some other points of interest should be used and more achievements added.

I concluded from this survey that I have created an application that has a sleek design and is very easy to use, with multiple features that users found useful. Having said this, I also found that there were a few limitations in some features in the app that need to be addressed in order to cater for a wider audience.

## 5.3 Objective Reflection

One way to gauge whether the project can be considered a success is to reflect back on the broad objectives proposed in Section 1.2. The objectives are listed below, each with an explanation as to whether it was achieved as a result of the quantitative and qualitative evaluation discussed in the previous two sections.

**Obj 1 Encourage walking:** the basic features aimed to achieve this objective, namely inviting users to go on a walk and gamification, were implemented. User testing yielded that the tracking walks feature was maybe not up to scratch compared to existing fitness applications, which is understandable. The gamification aspect was popular, but more achievements need to be added in order to encourage the user to walk more. Based on this, I can conclude that although the basic features were implemented, this objective was not fully achieved.

**Obj 2 Help discover the world:** the points of interest feature was implemented as I had planned and so I would say this objective was mainly achieved. The survey results showed that the plaques displayed on the map when tracking a walk was one of the most popular features in the app. A few users were not satisfied with just plaques being shown and would like other points of interest to be used, which is something that would be implemented in the future.

**Obj 3 Test and evaluate with real users:** throughout the project the app was given to known and unknown users to test and evaluate the app. From the final survey, the majority of users have stated that the app encouraged them to walk and exercise more. Based on this, I can constitute this objective as a success.

### 5.3.1 Project Management

To reflect on how well I managed my time during the project I compared the original project timeline proposed at the start of the project to the actual time taken to complete each task. The comparison between the two can be seen in Table 5.1.

I had planned to implement a great deal of the project, including setting up the back-end and skeleton app, before the exams at the end of the second term (week of $20^{\text{th}}$ March) so as to provide myself with a platform to continue the project after this date. In reality however, due to the combination of the workload I had during this time and the complexity of the back-end, I did not implement as much as I had hoped.

I did not anticipate how long the initial phase of the API implementation would take me. I had no experience in Javascript or any back-end development before the start of the project and so there was a much steeper learning curve to that of the front-end development. I researched

PROPOSED

| Activity | February | | | March | | | | April | | | | May | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 20 | 27 | 6 | 13 | 20 | 27 | 3 | 10 | 17 | 24 | 1 | 8 | 15 | 22 | 29 |
| Skeleton app | | | | | REVISION | EXAMS | | | | | | | | | | |
| Set up web server | | | | | | | | | | | | | | | | |
| Set up database | | | | | | | | | | | | | | | | |
| Login system | | | | | | | | | | | | | | | | |
| Track walks | | | | | | | | | | | | | | | | |
| User profile | | | | | | | | | | | | | | | | |
| Invite users for walk | | | | | | | | | | | | | | | | |
| Gamification | | | | | | | | | | | | | | | | |
| Extensions | | | | | | | | | | | | | | | | |

ACTUAL

| Activity | February | | | March | | | | April | | | | May | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 13 | 20 | 27 | 6 | 13 | 20 | 27 | 3 | 10 | 17 | 24 | 1 | 8 | 15 | 22 | 29 |
| Skeleton app | | | | | REVISION | EXAMS | | | | | | | | | | |
| Set up web server | | | | | | | | | | | | | | | | |
| Set up database | | | | | | | | | | | | | | | | |
| Login system | | | | | | | | | | | | | | | | |
| Track walks | | | | | | | | | | | | | | | | |
| User profile | | | | | | | | | | | | | | | | |
| Invite users for walk | | | | | | | | | | | | | | | | |
| Gamification | | | | | | | | | | | | | | | | |
| Extensions | | | | | | | | | | | | | | | | |

Table 5.1: Comparison between the proposed schedule for the project (top) and the actual schedule (bottom).

numerous articles and tutorials to try and gain some insight into what tools were commonly used and how these tools were used to create a functioning API. The tutorial I found most helpful was from SlatePeak [32], which helped me define the structure of the API using an Express router and find out the tools used to implement token-based authentication in Node.js.

On the other hand, some of the front-end features actually took a shorter amount of time than what I had planned. For example, inviting users to go on a walk actually only took around two weeks to implement rather than the three weeks I had outlined in Table 5.1, meaning that I gained some time back from the time lost implementing the API setup.

I had made sure to allow for extra time at the end of the implementation phase so that the problems listed above did not hinder the project and I was able to complete my intended features in time, albeit not having enough time to implement the extensions I had planned before the project. These were formulated into future extensions, which are listed in detail in Section 6.2.

## 5.4 Summary

Based on the evaluation conducted in this chapter, I believe I have created a fully functioning and well tested application that achieved most of my initial objectives. The final application does however have many limitations that were noticed by both myself and other users during testing.

We have found that the RESTful style of the API provided an elegant way to interact with the database. The categorised structure that was defined meant that each endpoint in the API had a single functionality to read from or write to the database, and once the base structure was implemented it was very easy to add or remove endpoints.

We have also found that JSON used in HTTP requests to communicate between the mobile app and the API worked well, owing to the flexibility and structure of the notation. JSON could be parsed and constructed by both the front and back-end, allowing for easy data transfer.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusion

The main aim of this project was to create a means to encourage people to walk more and help people discover more about the area around them. In that sense, based on the feedback received from users who tested and used the application, we have achieved what we set out to achieve.

A number of methods were researched and employed in the project in an attempt to motivate the user to exercise more, including gamification and exercising in groups. Gamification was implemented in the application by allowing achievements to be gained for walking further and more regularly, while group exercising was implemented by encouraging users to walk with one another using walk invitations.

Back-end development, including Javascript and the Node.js environment, were completely new to me before the start of the project and therefore required a lot more time to learn than other parts of the project. Following research into the types of technologies that needed to be used in the back-end, as well as a few tutorials to help with API design and authentication, I was able to create an elegant and well structured API using Javascript that served the mobile application.

Throughout the course of this project, we have undertaken the following:

1. Researched existing fitness applications and technologies to determine which features are most popular and what technologies would be most useful to me (Chapter 2).
2. Designed and implemented a connected, fully-functioning back-end Node.js API and a front-end iOS mobile app that allows a user to record their walks, view points of interest near them, gain achievements to increase their score and invite one another to go on a

walk (Chapters 3 & 4).

3. Tested each component using automated testing services and evaluated the project with real users to gauge their opinion on design, ease-of-use and how useful the application is (Chapter 5).

We have produced an application that provides a number of innovative features relating to promoting exercise and discovering new places, including viewing local points of interest and inviting users to go on a walk. Although some of the features need polishing and the app does still have its limitations, what I have created provides a good stepping stone to create a complete social walking app.

## 6.2 Future Work

Listed below are some of the future extensions that I would like to implement in the application given more time. Some of the extensions are based on current features that can be extended and others are new ideas that I thought of throughout the project.

### 6.2.1 Live walk tracking system

Due to time constraints, the proposed feature to invite users on a walk was not fully implemented. The current system only allows the walk to be tracked by the invitation sender, while the recipients cannot view the current walk until it has completed and is visible on their profile.

The proposed extension plans to show all the users of a shared walk the walk tracking view at the same time, so that all users are able to view walk statistics and discover points of interest around them. To make sure that all users were viewing the same map at the same time, data would need to be streamed from the invitation sender to all of the recipients.

### 6.2.2 Add photos to walk

While the current implementation only allowing for a walk route and its statistics to be saved, a step forward would be allowing the user to take photos of points of interest along their walk. These photos would then appear on the map once the walk had been saved.

### 6.2.3 Popular and nearby walks

A new section of the app could be created that shows both popular and nearby walks created by other users. Walks will therefore obviously need to be able to be rated, possibly using a

like/dislike system. Pre-made sets of well-known walks could also be shown in this section – the London Loop being a good example of one.

### 6.2.4 Favourite points of interest

After using the application a lot, I realised there were points of interest viewed during a walk that I wanted to refer back to later. A favourites system will enable the user to save a number of their favourite or interesting places while tracking a walk. These favourites will then be visible in a new section of the app even after the walk has finished.

### 6.2.5 Activity feed

While users can invite each other to go on a walk together, there is no current system to connect or view each other's profile. This could be done by allowing users to add each other as friends. Another section of the app could then be used to display a recent activity feed to show users what walks their friends have recorded recently.

# Bibliography


[1] S Kautiainen, L Koivusilta, T Lintonen, S M Virtanen, and A Rimpela. Use of information and communication technology and prevalence of overweight and obesity among adolescents. *International Journal of Obesity*, 29(8):925–933, aug 2005. ISSN 0307-0565. doi: 10.1038/sj.ijo.0802994. URL `http://www.nature.com/ijo/journal/v29/n8/pdf/0802994a.pdf`. Accessed 23/01/2017.

[2] Mitchell A. Lazar. How Obesity Causes Diabetes: Not a Tall Tale. *Science*, 307(5708), 2005. Accessed 25/01/2017.

[3] Cameron Lister, Joshua H West, Ben Cannon, Tyler Sax, and David Brodegard. Just a fad? Gamification in health and fitness apps. *Journal of Medical Internet Research*, 16(8): e9, aug 2014. ISSN 14388871. doi: 10.2196/games.3413. URL `http://games.jmir.org/article/viewFile/games_v2i2e9/2`. Accessed 23/01/2017.

[4] Thomas G. Plante, Laura Coscarelli, and Maire Ford. Does Exercising with Another Enhance the Stress-Reducing Benefits of Exercise? *International Journal of Stress Management*, 8(3):201–213, 2001. ISSN 10725245. doi: 10.1023/A:1011339025532. URL `https://www.psychologytoday.com/files/attachments/34033/exercise-another.pdf`. Accessed 26/01/2017.

[5] MapMyFitness Inc. Walking Maps and Walking Route Planner for Fitness — MapMyWalk. URL `http://www.mapmywalk.com/`. Accessed 30/01/2017.

[6] Strava Inc. Strava — Run and Cycling Tracking on the Social Network for Athletes. URL `https://www.strava.com/`. Accessed 31/01/2017.

[7] Lets Walk App. Lets Walk App. URL `http://www.letswalkapp.com/`. Accessed 31/01/2017.

[8] Google Inc. Google Maps - Navigation & Transit on the App Store, . URL `https://itunes.apple.com/us/app/google-maps-navigation-transit/id585027354?mt=8`. Accessed 31/01/2017.

[9] Citymapper. Citymapper - The Ultimate Transport App. URL `https://citymapper.com/`. Accessed 31/01/2017.

[10] Habitica. Habitica — Your Life the Role Playing Game, . URL `https://habitica.com/static/front`. Accessed 08/02/2017.

[11] Habitica. How it Works, . URL `https://habitica.com/static/features`. Accessed 08/02/2017.

[12] Intuit Inc. Mint: Money Manager, Bill Pay, Credit Score, Budgeting Investing, . URL `https://www.mint.com/`. Accessed 02/02/2017.

[13] Intuit Inc. Mint: Budget, Budgeting Goals, . URL `https://www.mint.com/how-mint-works/budgets`. Accessed 02/02/2017.

[14] Google Inc. Google Places API — Google Developers, . URL `https://developers.google.com/places/`. Accessed 06/02/2017.

[15] Google Inc. Displaying Attributions — Google Places API for iOS — Google Developers, . URL `https://developers.google.com/places/ios-api/attributions`. Accessed 05/02/2017.

[16] Apple Inc. MKLocalSearch - MapKit — Apple Developer Documentation, . URL `https://developer.apple.com/reference/mapkit/mklocalsearch`. Accessed 06/02/2017.

[17] Kevin Selwyn. kevinselwyn/pokestop: Find nearby Pokestops. URL `https://github.com/kevinselwyn/pokestop`. Accessed 02/02/2017.

[18] English Heritage Trust. English Heritage Home Page — English Heritage. URL `http://www.english-heritage.org.uk/`. Accessed 17/06/2017.

[19] Open Plaques. Open Plaques - linking historical people and places - Open Plaques. URL `http://openplaques.org/`. Accessed 19/06/2017.

[20] Apple Inc. Getting the User's Location, . URL `https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/LocationAwarenessPG/CoreLocation/CoreLocation.html`. Accessed 03/02/2017.

[21] Apple Inc. MapKit — Apple Developer Documentation, . URL `https://developer.apple.com/reference/mapkit`. Accessed 05/02/2017.

[22] Node.js Foundation. Node.js. URL `https://nodejs.org/en/`. Accessed 22/05/2017.

[23] Mongoose. Mongoose ODM v4.10.2. URL `http://mongoosejs.com/`. Accessed 24/05/2017.

[24] Heroku. Cloud Application Platform - Heroku, 2016. URL `https://www.heroku.com/`. Accessed 19/06/2017.

[25] Apple Inc. Design Principles - Overview - iOS Human Interface Guidelines, . URL `https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/`. Accessed 31/05/2017.

[26] StrongLoop and IBM. Express - Node.js web application framework. URL `http://expressjs.com/`. Accessed 04/06/2017.

[27] Jared Hanson. Passport. URL `http://passportjs.org/`. Accessed 05/06/2017.

[28] GeoJSON. GeoJSON. URL `http://geojson.org/`. Accessed 07/06/2017.

[29] Chris Ballinger. xcodebuild exited with 65. URL `https://github.com/travis-ci/travis-ci/issues/6675`. Accessed 25/05/2017.

[30] Mocha. Mocha - the fun, simple, flexible JavaScript test framework. URL `https://mochajs.org/`. Accessed 11/06/2017.

[31] Sverrir Sigmundarson. Maps to GPX Converter for location aware development. URL `https://mapstogpx.com/mobiledev.php`. Accessed 17/06/2017.

[32] Joshua Slate. Creating a Simple Node/Express API Authentication System with Passport and JWT. URL `http://blog.slatepeak.com/creating-a-simple-node-express-api-authentication-system-with-passport-and-jwt/`. Accessed 18/06/2017.

# Appendix A

# Existing applications matrices

The full matrices for the existing applications are shown below. Table A.1 contains all the existing fitness and walking applications, while Table A.2 contains existing journey planners.

| **Features** | | **MapMyWalk** | **Strava** | **Let's Walk** | **Pokémon Go** |
|---|---|---|---|---|---|
| Design (2) | Clean, uncluttered design | ✗ | ✓ | ✗ | ✓ |
| | Nice colour scheme | ✓ | ✓ | ✗ | ✓ |
| Ease of use (3) | All functions of app work correctly | ✓ | ✓ | ✓ | ✓ |
| | App is not slow/clunky | ✓ | ✓ | ✗ | ✓ |
| | Features accessible within 3 clicks | ✓ | ✓ | ✗ | ✓ |
| Tracking location (2) | Accurately tracks location while walking | ✓ | ✓ | ✓ | ✓ |
| | Records information about number of steps, distance travelled, etc. | ✓ | ✓ | ✓ | ✗ |
| Navigation (4) | Journey view while walking | ✓ | ✓ | ✓ | ✓ |
| | Provides accurate navigation directions | ✓ | ✗ | ✗ | ✗ |
| | Gives information about points of interest near user | ✗ | ✗ | ✗ | ✓ |
| | Able to take photos during walk | ✓ | ✗ | ✗ | ✗ |
| Social interaction (5) | Able to publish walks completed to profile | ✓ | ✓ | ✓ | ✗ |
| | Leaderboard of most popular walks | ✗ | ✗ | ✗ | ✗ |
| | Able to invite other users to join walks | ✗ | ✗ | ✗ | ✗ |
| | Each user has a score based on km walked, day streaks, etc. | ✗ | ✗ | ✗ | ✗ |
| | Users can add other users as friends | ✓ | ✓ | ✓ | ✗ |
| Total | | 11 | 10 | 6 | 8 |

Table A.1: Matrix for existing fitness/walking applications

| Features | | Google Maps | Citymapper |
|---|---|---|---|
| Design (2) | Clean, uncluttered design | ✓ | ✓ |
| | Nice colour scheme | ✓ | ✓ |
| Ease of use (3) | All functions of app work correctly | ✓ | ✓ |
| | App is not slow/clunky | ✓ | ✓ |
| | Features accessible within 3 clicks | ✓ | ✓ |
| Tracking location (2) | Accurately tracks location while walking | ✓ | ✓ |
| | Records information about number of steps, distance travelled, etc. | ✗ | ✗ |
| Navigation (4) | Journey view while walking | ✓ | ✓ |
| | Provides accurate navigation directions | ✗ | ✓ |
| | Gives information about points of interest near user | ✗ | ✓ |
| | Able to take photos during walk | ✗ | ✗ |
| Social interaction (5) | Able to publish walks completed to profile | ✗ | ✗ |
| | Leaderboard of most popular walks | ✗ | ✗ |
| | Able to invite other users to join walks | ✗ | ✗ |
| | Each user has a score based on km walked, day streaks, etc. | ✗ | ✗ |
| | Users can add other users as friends | ✗ | ✗ |
| Total | | 7 | 9 |

Table A.2: Matrix for existing journey planners

# Appendix B

# User Guide

## B.1 Login screen

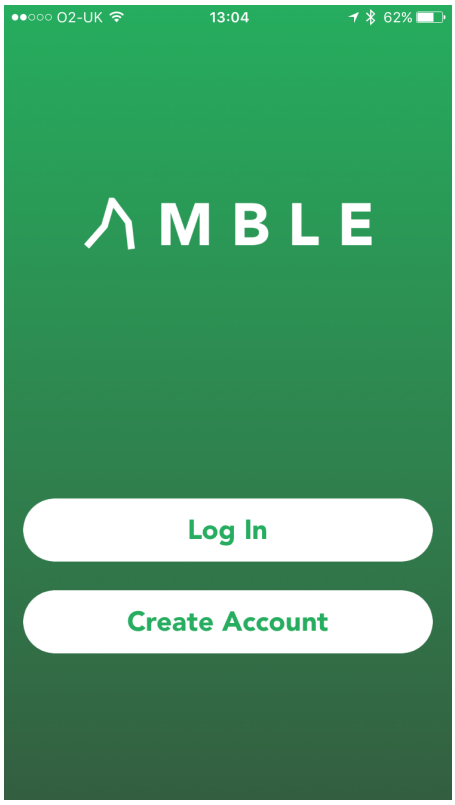



Figure B.1: Initial launch screen shown when launching the app for the first time or when not logged in.

When you launch the application for the first time, you are greeted with the login initial launch screen shown in Figure B.1. From here, you can either login to an existing account or create a new one.

Once you choose to login or register, you are taken to the screens displayed in Figure B.2. The view on the left prompts you to log in with your username and password, while the view on the right asks you to enter additional details to register an account including your full name and email address.

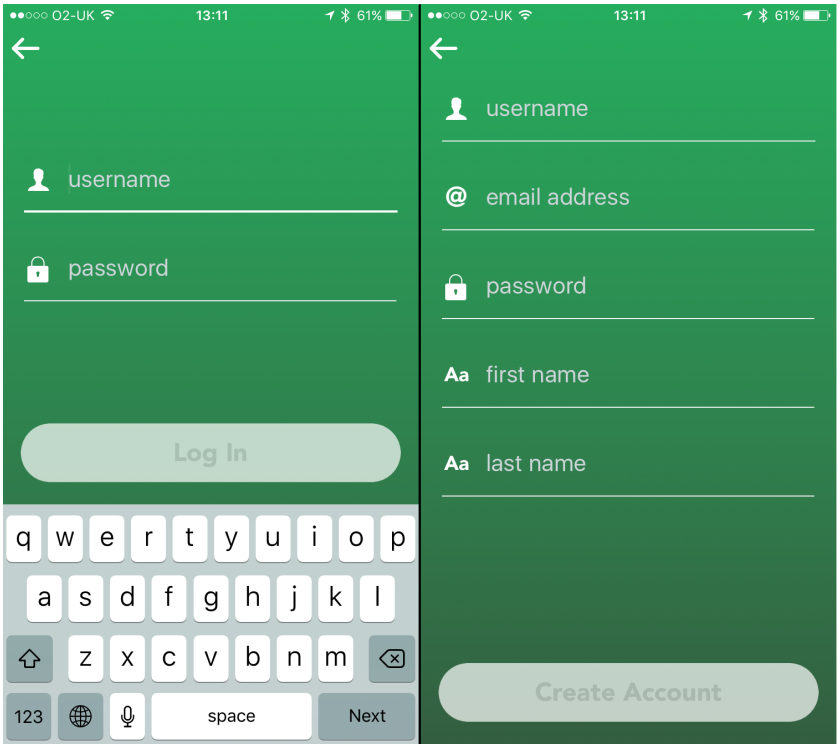



Figure B.2: Login screen of the app (left) and register screen (right).

## B.2 Track walks

Once you have authenticated successfully, you are taken to the main screen of the app shown in Figure B.3, where a map is displayed and walks can be tracked. Here, the app is split into three sections – the first of which is tracking a walk. As long as you are logged in, you will automatically be taken to this screen when you launch the app.

The arrow button in the bottom-left of the screen allows you to toggle between tracking modes – tapping it once will zoom to your location on the map, and tapping it again will add a compass that will rotate the map as you move your device.

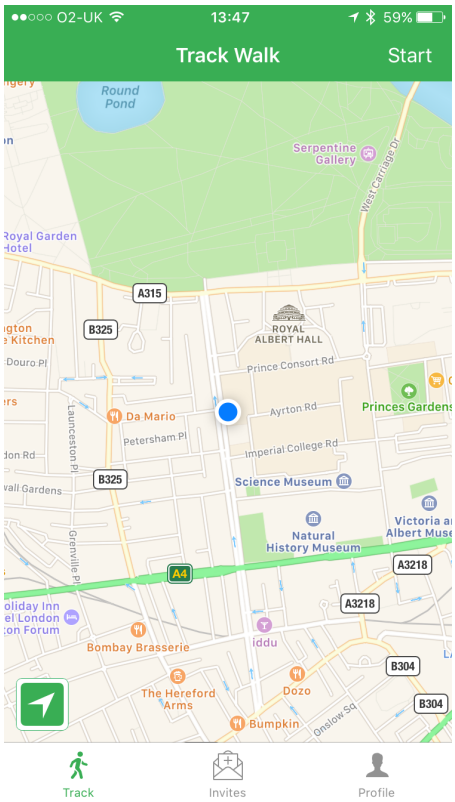



Figure B.3: The main screen of the app to track walks, shown after logging in or when launching the app already logged in.

To start tracking a walk, you can tap the *Start* button in the top-right. The app then brings up the statistics view with a timer of the walk duration, the distance travelled during the walk and the number of steps taken. A line is drawn along your route as you begin to walk. Points of interest will also appear around you as you walk along, shown as red pins on the map. The full track walk view can be seen in Figure B.4.

Tapping on a pin on the map during a walk will result in a popup with a short description and image. To get further information about the point of interest, you can tap the $i$ button. This will bring up a screen with more details about the plaque, including its inscription and related person(s). Tapping on a person will bring up their Wikipedia page (if they have one). This process is shown in Figure B.5.

To save a walk, tap the *End* button in the top-right. This will give you a confirmation message and, once confirmed, will prompt you to enter a name for the walk. Once you enter a name and save the walk, the entire walk will be displayed as shown in Figure B.6. Here a map is displayed of the entire walk route as well as the walk statistics and achievements, along with the number of points that you gained, listed below.

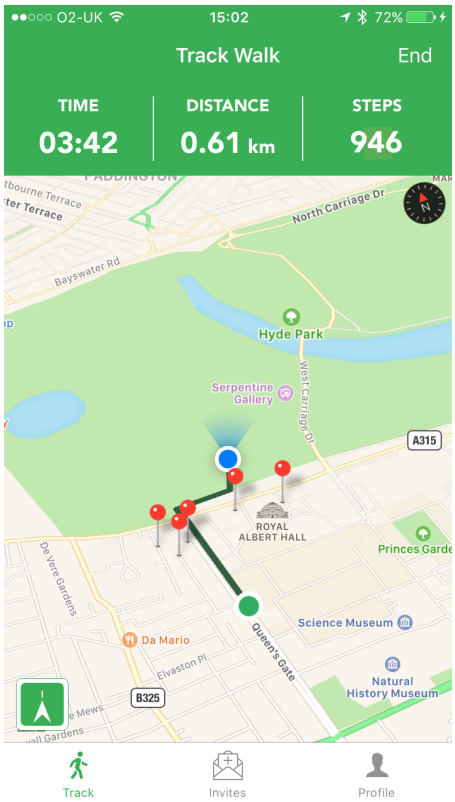



Figure B.4: The screen displayed while tracking a walk, showing the walk statistics at the top of the screen, a route on the map of where you walk and points of interest displayed around your location as red pins.

## B.3 Invitations

The second tab in the application is designed to send and respond to walk invitations. The view initially shows the invites you have sent, with each invite containing a label to show whether the invite is *Pending* or *Accepted*. If a sent invite is marked as accepted, a *Start Walk* button will appear that will switch to the track walk screen and begin tracking a group walk.

By using the toggle at the top of the screen, you can switch to view your received invites. Any pending invitations will be marked with a badge on the tab and a badge on the app's icon. *Accept* and *Decline* buttons are available – tapping decline will delete the invite and tapping accept will mark it as accepted. Both views can be seen in Figure B.7.

To invite other users on a walk, tap the + icon in the top-right of the menu bar. This will take you to the screen shown in Figure B.8, with a search bar to search for users by their username, name or email. Once a user is selected from the list of search results, it will appear in the *Selected Users* section. From here you can either perform another search to add more user(s) or tap the *Invite* button in the top-right of the screen to send the invite. You are then

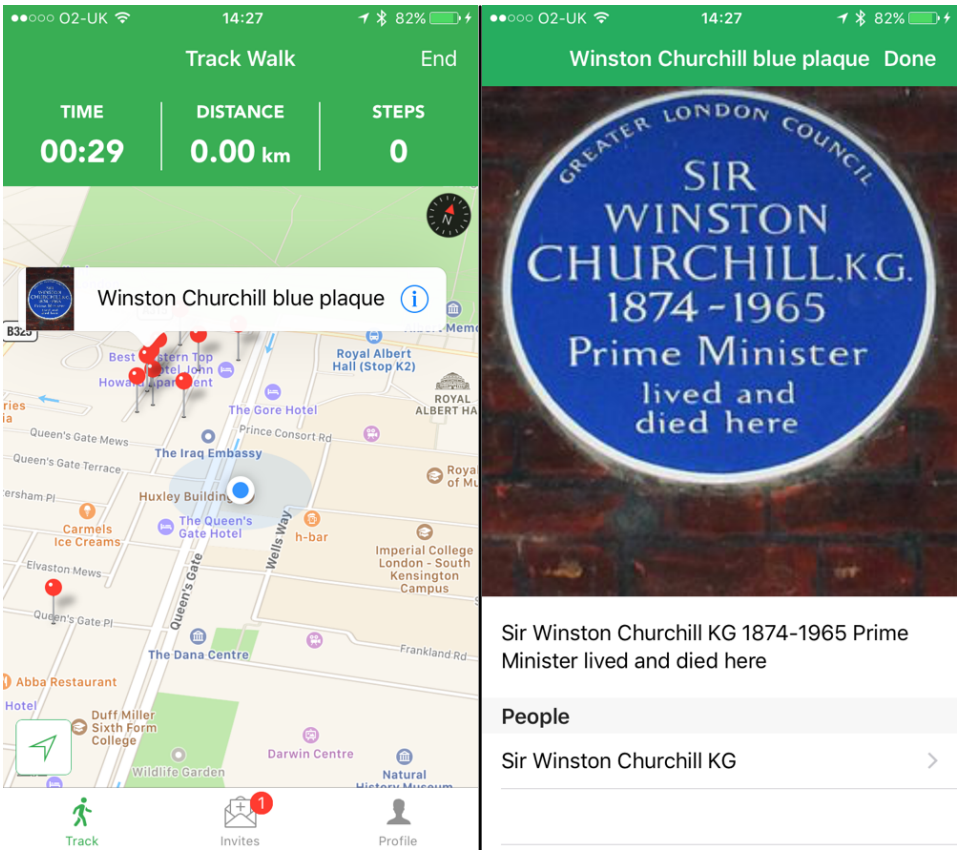



Figure B.5: Process of how to view points of interest in the app, with the popup appearing on the map (left) and a more detailed view once selected (right).

returned to the sent invites screen.

## B.4 Profile

The third tab of the application is your profile view, shown in Figure B.9. This view displays your user statistics – specifically your total number of points (or score), the total distance you have walked and your total number of steps taken.

A list of all the walks you have tracked, including group walks, are then shown in a grid formation below. Tapping on a walk will present you with the walk detail view discussed previously (Figure B.6).

There is also a settings screen accessible via the cog icon on the top-right of the profile view. This gives you the option to change your desired unit of distance and log out, which takes you back to the initial launch screen (Figure B.1).

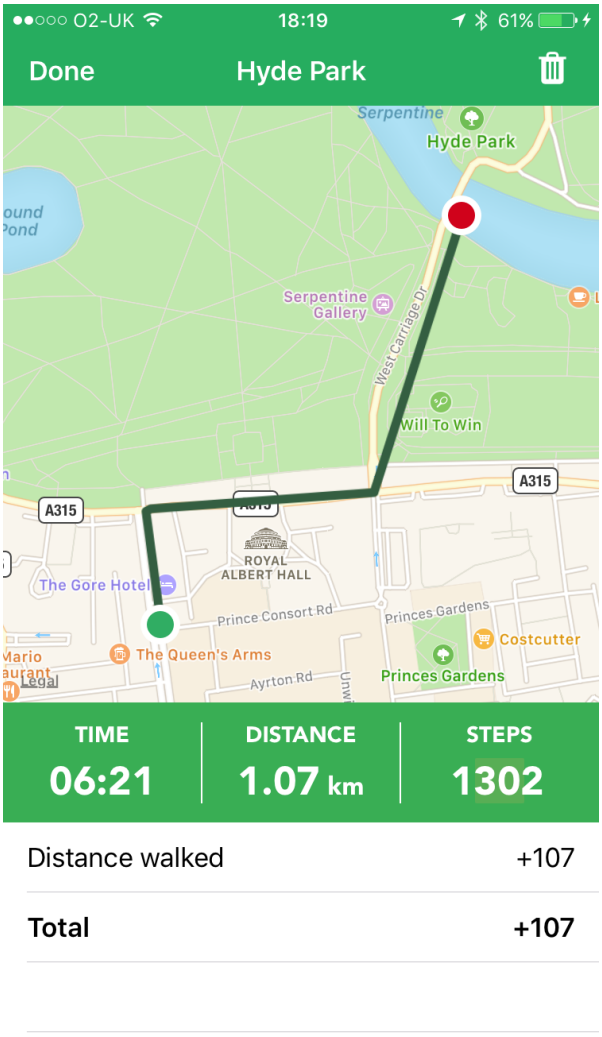



Figure B.6: The walk detail view containing the route of the walk, its statistics and achievements. This view is shown after the walk has been saved (or when selected on your profile).

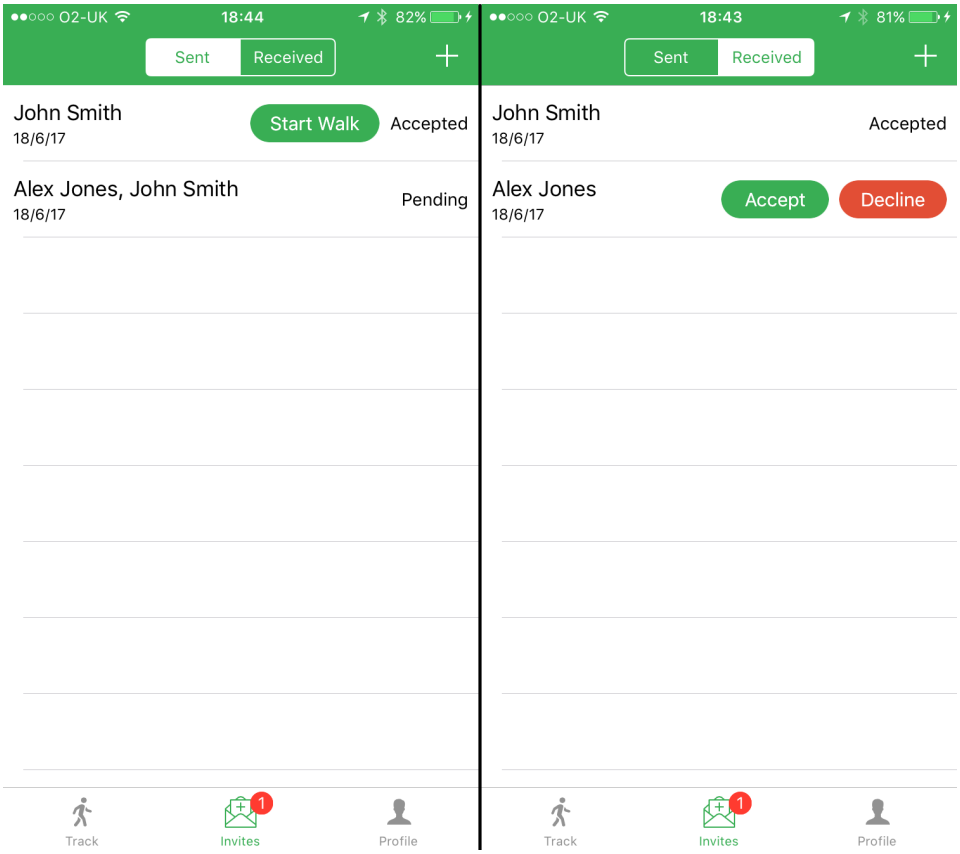



Figure B.7: Invites screen in the app, showing sent invites (left) and received invites (right).

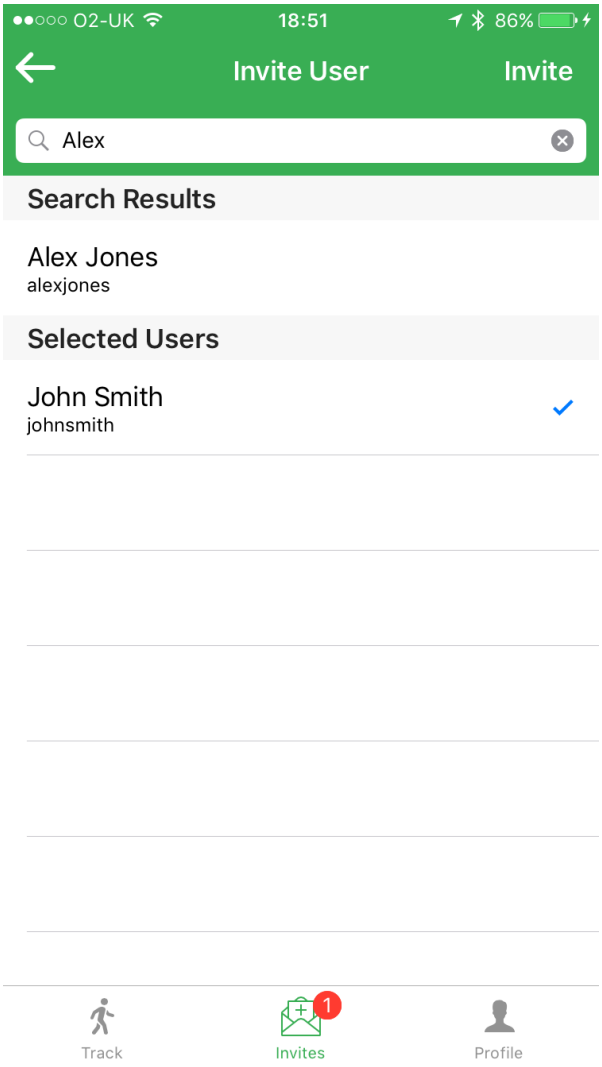



Figure B.8: Screen to send an invitation to one or more users.

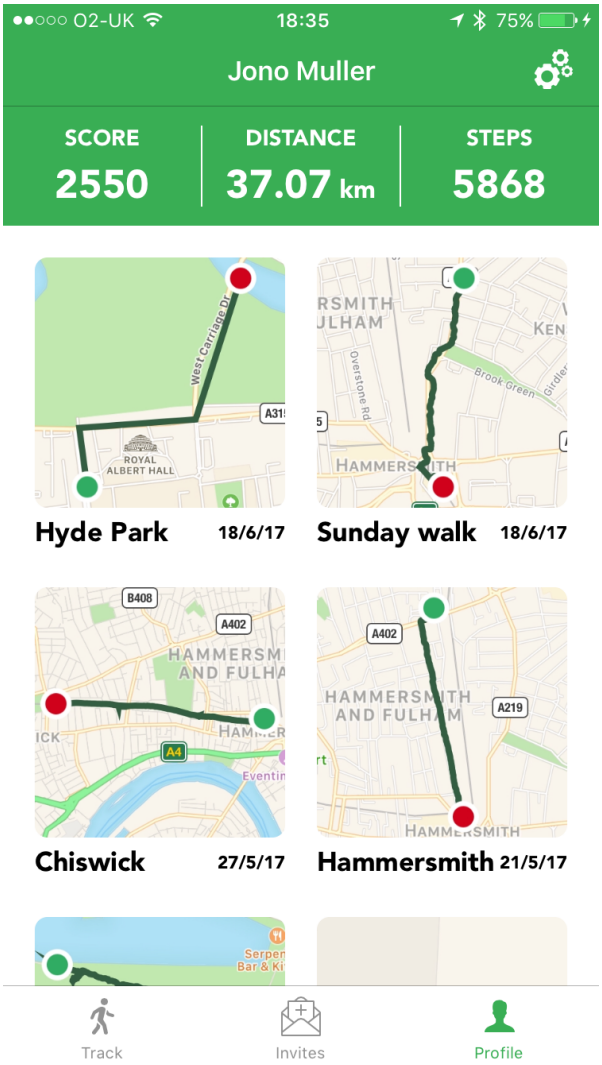
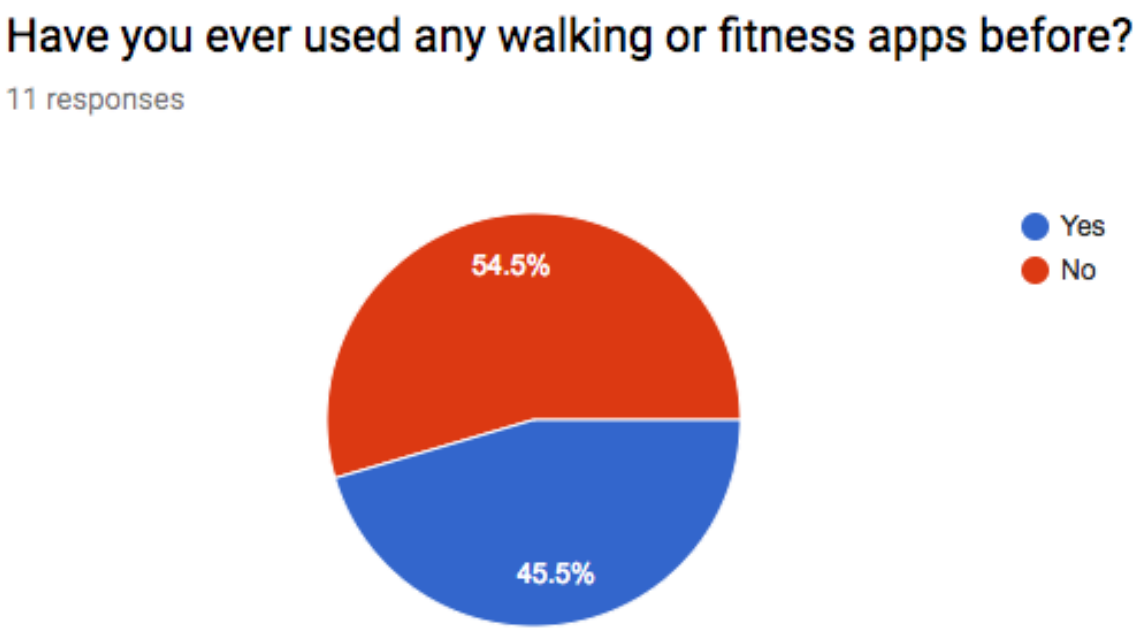



Figure B.9: Profile view in the app, showing your statistics and tracked walks.

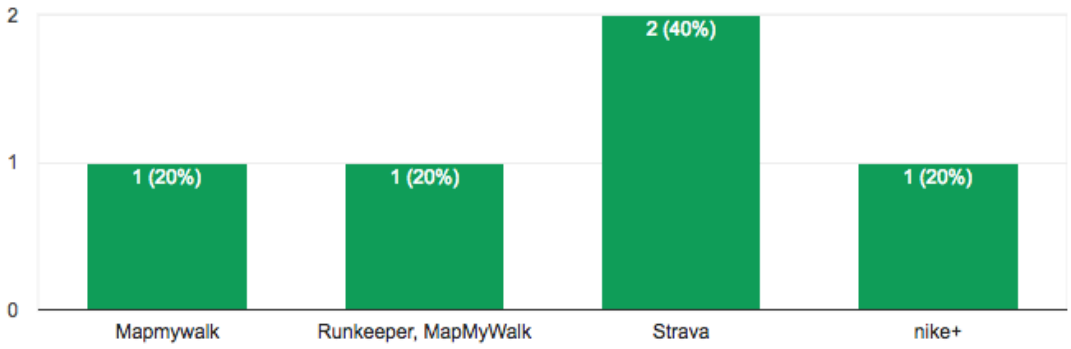
# Appendix C

# Final Survey Results



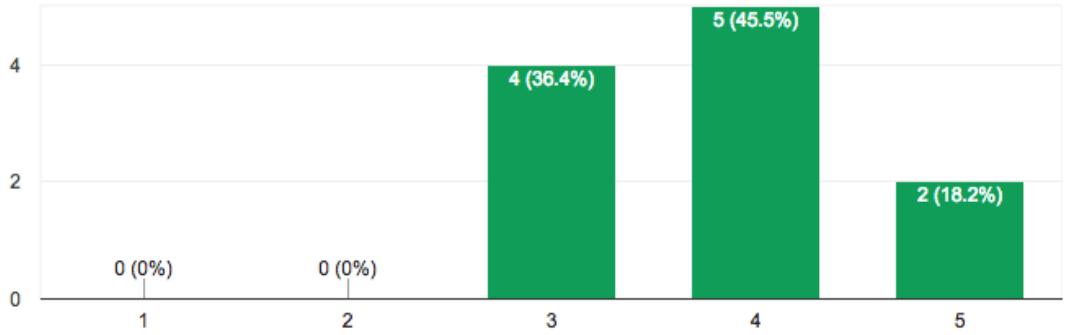
## If so, which one(s)?

5 responses



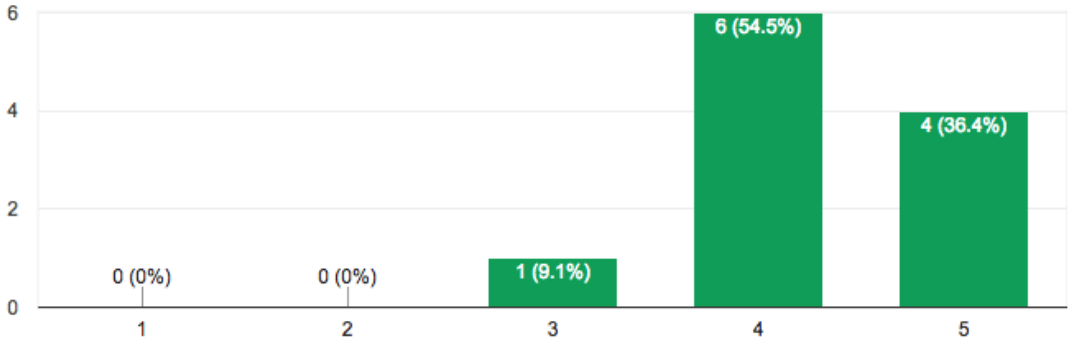

## How do you rate the overall design of the app?

11 responses



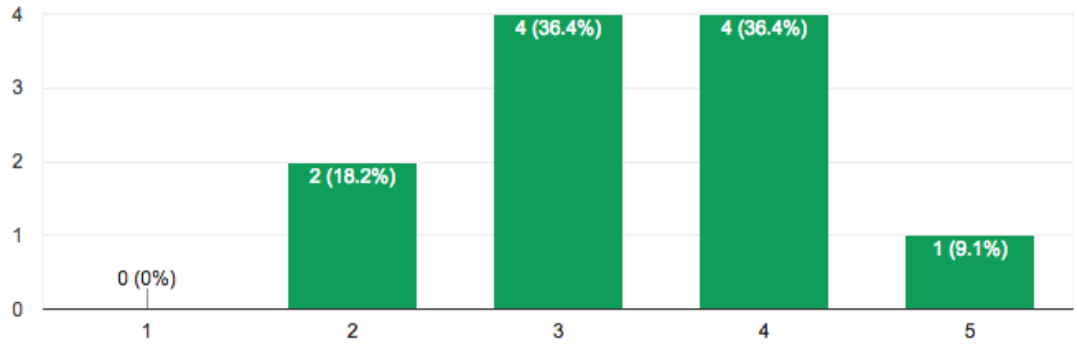

## How easy was the app to use?

11 responses



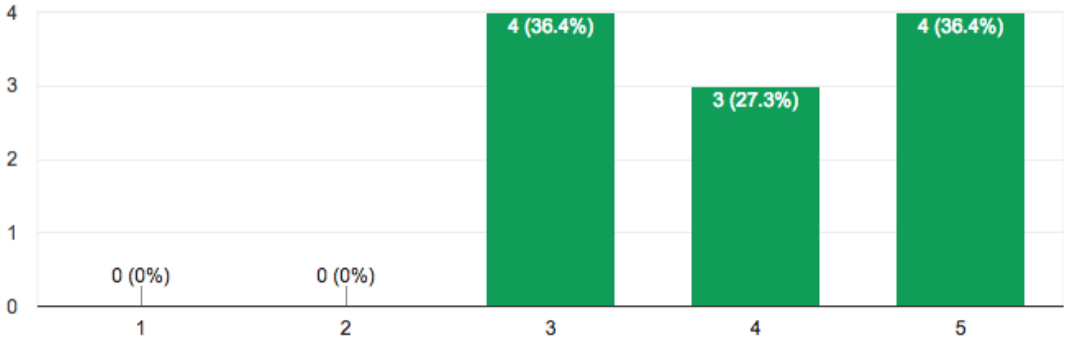
How useful was the app to encourage you to make you walk more?

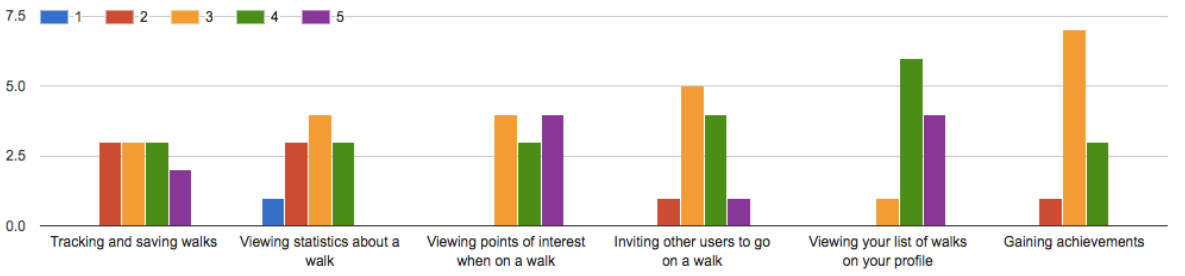
11 responses




How useful was the app to allow you to find out more about your area when tracking a walk?

11 responses



## Please add any other feedback or improvements you have.

6 responses

| |
|---|
| Points of interest work well |
| Looks good for walking but Strava is better for fitness. |
| I don't think it is clear what achievements you can get until you achieve them |
| Adding other places around you other than just plaques would be good. |
| Other apps I've used have calories burned, avg. pace, etc. for workouts |
| I really like the design and colour scheme |