Crowdsourced prediction of Cardiff bus delays (IOS)

FINAL REPORT

Ioana Surdu-Bob
1545977

CM3203 ONE SEMESTER INDIVIDUAL PROJECT
SUPERVISOR – MARTIN CAMINADA
MODERATOR – NATASHA EDWARDS

ABSTRACT

Nowadays, bus departure times are not being predicted in Cardiff. Unfortunately, the unreliability of these buses (Change.org, 2018) affects peoples' time schedules and might result in some using other forms of transport. Having predictable delays available ahead of time (hours to days), should be a viable solution to solving this problem.

This report will describe my approach for creating a mobile application that crowdsources the registration of actual departure times, and then process the delays for every bus stop in order to predict future delays. The result is composed of a mobile application (front-end) created in the Swift programming language, a back-end server that processes the delays, created in Ruby on Rails, and a Postgres database that stores it. To access external data, multiple APIs were used (Transport API and Postcodes.io).

The working prototype shows that bus delays can be crowdsourced and recorded correctly, which can give accurate predictions useful to commuters.

Further work could perfect the algorithm, test it more thoroughly, and result in the application going on the market. This application would improve the experience of using public transport by providing more accurate departure times because of the computed expected delays. Because of that, bus companies could be interested in partnering with us, as it might indirectly increase their revenue. If more people would use public transport because of increased efficiency, there would be less traffic which should increase road safety (less accidents), and also resulting in reduced pollution.

Below are the links to the source codes of the application presented below:

Mobile front-end part: https://github.com/IoanaBob/PublicTransport

Back-end part: https://github.com/IoanaBob/PublicTransportBackend

Server location: https://gentle-falls-45144.herokuapp.com/

CONTENTS

Abstract 1

Introduction 5

Problem 5

Alternative solutions and their limitations 5

Solution 6

Viability on the market 6

Background 7

IOS Development 7

APIs 7

Relational Databases 8

Working Environment 8

Approach 9

Aims and Goals 9

Functional Requirements 9

Non-Functional Requirements 10

Mobile development technologies 10

Candidates 10

Chosen technology 10

Backend development technologies 11

Candidates 11

Chosen technology 11

Database and Server 11

PostgreSQL 11

Heroku 12

External APIs used 12

Transport API 12

Postcodes.io 12

Planning 13

Milestones 13

Gantt chart 13

Use cases 13
Use case diagram 14
UC 1: Search bus stop (by current location) 14
UC 2: Search bus stop (by postcode) 15
UC 3: Search timetable 15
UC 4: View timetable 16
UC 5: Save timetable 16
UC 6: Saved timetables list 16
Implementation 17
Algorithms 17
Bus departure time detection 17
Delay detection 20
Expected delay calculation 20
System Architecture 20
Class Diagrams 20
Network diagram 23
Database model 24
Interface design 25
Navigation 25
Data entry 26
Error handling 28
Visual design 29
Evaluation and Results 30
Testing location updates 31
Testing mobile functionality 31
Testing backend functionality 32
Manual testing 33
Acceptance testing 34
Functional Requirements 34
Non-Functional Requirements 34
Future Work 35

Security 35
Architecture 35
Optimization 35
Interface design 36
Testing 36
Conclusions 37
Reflection On Learning 38
References 39
Appendix 41

INTRODUCTION

PROBLEM

Delays are one of the main disadvantages when using public transport. However, where applications (like Google Traffic) are available to keep track of delays for motorists, relatively little help is available for public transport users, especially for predicting the likelihood of any major delays.

To illustrate why the problem matters, suppose one is taking a local bus service to connect to another mode of transport (say, to the railway station, coach station or airport). If the bus is late, one might miss the connection, which is especially a problem if the ticket for this connection is non-refundable. In the absence of any information on the punctuality of the local bus service, the best one can do is to prepare for the worse and leave home way in advance. This of course is very inefficient and leaves public transport at a disadvantage compared to other forms of transport (e.g. car or taxi).

Moreover, because of the lack of reliability, commuters might prefer not using the bus if it might affect their schedule. They could opt for private transport, which is costly, but more reliable and convenient. If there would be a method to provide expected delays, buses could become more reliable and therefore more people would prefer using them. This could save commuters' money and generate more income for the bus company used. If people would use cars less and public transport more, this would reduce pollution and traffic jams. Having less cars on the road should also result in less car accidents, therefore keeping citizens safer.

ALTERNATIVE SOLUTIONS AND THEIR LIMITATIONS

An alternative solution would be to add GPS trackers to buses and receive real time locations in an application. This concept exists and works well in Geneva, Switzerland (Ronga, 2018), where commuters use an application (called TDG) to find out when the bus/tram will arrive. There are also companies that provide tracking devices (Trackimo, 2018), which are used for any type of transport (trucks, private school buses etc.). In Wales, there has been recent funding for providing real time information (bbc.co.uk, 2018), but there's no sign of future work on delay prediction, or even publishing the delay data online for third parties to use. There are no records of average delays on buses on the government website. There is only some information about London public transport, where delays are quite low because public transport arrives so frequently, and it states that 98.4% of the planned routes were covered (Transport For London, 2018). However, a route being covered means the bus arrives to all bus stops in the route, but not necessarily on time.

GPS tracking gives accurate information only minutes before the bus will arrive (while the bus is already in transit, following the course of the bus line a person is searching for). One would not be able to plan ahead for taking the bus to the airport, for example, because at the time the delay information appears, it's too late to prevent arriving to the airport late. Therefore, an alternative would be predicting delays from departure times.

Assuming buses will be tracked by GPS in a few years, actual delays cannot be accessed publicly (e.g. from an API[1]). However, actual arriving times could be recorded at no extra cost, by using the GPS of commuters' phones. Over time, actual arriving times could be crowdsourced from public transport users.

SOLUTION

Given the current limitations, my suggested way of dealing with this is to implement a smartphone app that records the actual departure times of public transport users and stores them in a database. This database (with historic delays) can then serve as a basis for prediction of any future delays, to the same smartphone users. The more data (departure times in different days) is recorded, the more accurate the prediction can be. If, for example, a bus comes 5 minutes late in the morning for an entire month, we can easily predict that it will be late in the future, too. Prediction should be the best solution for planning ahead.

The application registers when users get into the bus (the name of the bus stop, bus line and time). The aim is to automatically detect this information using the GPS of the phone, which is used to determine the proximity of a bus stop. Once the speed of the phone exceeds walking distance, the user is most likely to be inside the bus. A back-end server will receive, process and store the information sent from the mobile app. The same iPhone app allows users to view expected times and expected delays for a certain bus stop and line, using the interface. The user can query the nearest bus stations of a given location or of the current location, and view timetables (using existing APIs) along with expected delays.

VIABILITY ON THE MARKET

In Cardiff (and in UK generally), there are already mobile applications for public transport. An example is the Cardiff Bus app (Apple, 2018), where you can purchase tickets and show them to the driver. Personally, I only use the mobile application and, while I am in the bus, I see a big portion of commuters using it, too. This means commuters have been already introduced to mobile apps used to improve the public transport experience, and they would therefore embrace a new mobile application easier than a few years ago.

In my opinion, this application could be profitable in two ways. On one hand, it could be developed for an individual bus company. Since it would provide a better experience using public transport (shorter wait times), more users would be keener on using the service, and therefore generate revenue for the company. On the other hand, it could generate direct revenue through ads, which would be profitable only if the application would be widely used. Therefore, if one plans for this application to go on the market, research should be made in both directions.

If buses will end up being tracked by GPS, an alternative plan would be to use more accurate, unique data coming from buses instead of phones, and use the delay detection algorithm I created to generate expected delays. Even if in the future there will be new developments, part of this application could still be used on the market.

[1] Application Programming Interface

BACKGROUND

Given the project requirements, it was clear my project would consist in an iPhone app connected to external APIs. In this section, I will try to roughly explain the environments I worked in, and how do they differ to other forms of development.

IOS DEVELOPMENT

Disregarding the programming language iOS development is done in, it is very different to web development, and can be quite different to Android development, too. As I have worked on both mobile and web development prior to this project, I have realized that generally, a web application can take much less time to develop, especially as a beginner. For example, there are web frameworks that give code structure and shortcuts (background "magic"), such as Django or Ruby on Rails, which speeds up development. In mobile development, you use what you are provided with (iOS/Android SDKs[2]) and generally have the freedom to choose your project structure, which results in more time being lost. Also, while moving through pages in HTML is quite easy, in mobile development some sort of navigation is used (in iOS – navigation controller, tab bar controller etc.). iPhones also do not include a back button, so navigation must be set up to go back to the previous page. While in web development we look at a whole stack of processes that go from a database/ server request to generating the DOM[3], mobile development can be mainly compared to the front-end part of web development (e.g. Angular part of a web project). Since the processing power of a phone is much smaller, heavy data processing is not an option.

Developing for iOS devices compared to Android is also different. Android is more open (open source software, choice of hardware etc.) while iOS is all controlled by Apple, therefore making it more restrictive (strict rules and conventions, can only be developed on a Mac). Also, since Android caters to more devices, it also must be supported on different hardware and many more versions (Sinicki, 2016). While checking the flagship phones every year, you can notice how Apple devices have less performant hardware (e.g. iPhone 6s has 2GB RAM while Samsung Galaxy S6 which came out earlier has 3GB RAM) (Samsung, 2018). The reason Apple products still perform the same is because of software optimization. Apple hardware is made to work on Apple software, while Android software is made to work on any Android hardware. That's why acceptance on the App Store is much harder to obtain.

APIS

Nowadays, APIs become more and more common, especially in mobile development. Because of the limited processing power of a phone, processing must be done somewhere else. Even in web development, most processing is done on a server, not on the users' computer when accessing a website, so it makes sense the "heavy lifting" should be done on a powerful machine. APIs connect the machine that does the processing (the server) to the rest of the application (Gazarov, 2016). "At some point or another, most

[2] Software Development Kits
[3] Document Object Model (W3.org, 2018)

large companies have built APIs for their customers, or for internal use" (Gazarov, 2016). In this project I used APIs built by others to gather external data, but I am also going to create an internal one, using my data.

Relational Databases

Relational databases have been around for many years – 48, to be exact (Techtarget.com, 2018). They are collections of data items structured in tables with attributes. Relational databases create relations between different types of data, which can be searched through using SQL[4]. Nowadays, mobile applications tend to use different types of NoSQL databases, which provide the availability that relational databases don't give, but compromise on consistency. However, many companies still tend to prefer the reliability and the consistency of the relational model. The size should not be a problem, as the theoretical table limit is of 64TB (Matthew, 2005).

Working Environment

Since iOS development can be only done on a Mac machine and I don't own one, setting up a good working environment was quite difficult and required some planning. Development can only be done in XCode, the IDE[5] provided by Apple (Apple.com, 2018). Starting up a project is quite easy but testing it on a phone is not. Before taking this project, I had to make sure I could develop the application in the laboratories and test it there. It ended up being a difficult process as XCode is not quickly updated in the school. At the time I tried to create the project, only XCode 8 was available, but it did not support the latest version of the programming language I was going to use (Swift 4). I did not have the permission to upgrade it myself, therefore I had to wait for the school to do it. In the end, I managed to deploy on an iPad mini I had on hand. However, I did not have an iPhone to test on yet, and I knew it was essential for testing as it uses the location and the movement of the phone. Therefore, I bought a refurbished iPhone and sold the phone I had at the time.

Testing on phone also requires signing some certificates with a developer account, which prove the machine I am deploying from is a Mac, and the software written is for iOS. Fortunately, Apple now provides free developer accounts, with some limitations, and only for testing. If the developer wants to publish on the app store or easily share the app on multiple devices, she must have a paid developer account (99 USD/year). But the purpose of this application is to create a prototype, or MVP[6], for which a free developer account is sufficient.

The backend environment was easier to set up, as I could do it from any computer. As the language I used (Ruby) works much better on Linux, I used an Ubuntu-based operating system, called Elementary OS (Wikipedia.org, 2018). Creating a running blank project was much less of a hassle than on Mac.

[4] Structured Query language (Techtarget.com, 2018)
[5] Integrated Development Environment
[6] Minimum Viable Product

To keep my code backed up and have it organized, I chose to use version control for both of my applications (front-end and back-end). My preferred option was Git on GitHub, as we have used it in prior projects for the University and I generally consider it my portfolio, where other people can see my work.

APPROACH

AIMS AND GOALS

We aimed for this project to include:

- The development of an iPhone app that registers when user got into the bus (the name of the bus stop, bus line, terminal and time). The aim is to automatically detect this information using the GPS of the phone, which is used to determine the proximity at a bus stop. Once the speed of the phone exceeds walking distance, the user is most likely to be inside the bus.
- The development of a back-end server that receives, processes and stores the information. Delays should be linked to expected arrival times.
- The same iPhone app should allow users to view expected times and expected delays for a certain bus stop and line, using the interface. Ideally, the user should get options of what to query. He should be able to query nearest bus stations to a given location or current location, and view timetables (using existing APIs) along with expected delays.

Functional Requirements

Crowdsourced data capturing

Every device which has the application installed should feed data to the server. Every time a user takes the bus (determined by the proximity to the bus station and the speed), the application will record the expected and actual departure time of the bus, the direction (terminal) and bus line. Whenever convenient, the information will be sent to the server. The more users take the bus, the more data we have, resulting in a more accurate prediction of future delays.

Anonymity

The only user information available should be a unique identifier linked to their device.

Server Availability

The database should always be online, accessible on an IP address.

Delay Estimation

If one took a bus at similar times of the day/week and it was recorded by the application, the back-end should process historical data and calculate the estimated delay for a future route. If there is no relevant data available, no delay should be suggested.

Nearby bus stop retrieval by current location

By using the current location of the phone, the application should retrieve the closest bus stops to the location, using existing APIs (transport API).

Nearby bus stop retrieval by postcode

By using postcode input, the application should retrieve the closest bus stops to the postcode location.

Bus station timetable

When inputting a bus station, a time and a date, the application should retrieve a bus timetable for the station, starting at the date given, along with estimated delays (covered earlier).

Non-Functional Requirements

Usability

User should navigate through pages seamlessly, without knowing about the crowdsourced functionality of the application. The user will be able to access timetables with expected delays within a few clicks.

Reliability

Timetable information should be displayed even if there is no delay data available for a certain date/time. In the beginning, the route coverage will be low, therefore there will be a high chance of not having historical data.

Performance

Users should retrieve responses to any request in less than a second. Only most relevant results should be shown, so that it takes less time to process.

MOBILE DEVELOPMENT TECHNOLOGIES

Candidates

Mobile development can be done natively or using cross-platform frameworks that allow developers to create mobile apps for both Android and iOS. Personally, I have tried both. I used React Native (cross platform) and Swift (native iOS) to create identical applications, which gave me a good overview of their differences. On one hand, cross platform development gives quicker results, as there is no need for two teams of developers (one for iOS and one for Android). However, it compromises on the quality of the product. I found Swift much easier to pick up, because it is an object-oriented language inspired from both Java and JavaScript. React Native, on the other hand, employs a component-based architecture that is not useful until the application grows. Also, as I have touched earlier, Android software consumes more resources. Having software made for both also results in more resources being consumed on an iOS device. When putting the two applications side by side, I have found that the React Native application would heat up the phone while being on, with no tasks in progress. Further tests proved that the React Native app consumed much more resources.

Chosen technology

Given my findings, of course my chosen technology was Swift. Because Apple developers are being paid to develop this language (it's not open source), the error handling is much better, and everything is well documented. Also, XCode is the most well-thought IDE I have tried. Because of these reasons, I find development in Swift much more pleasant, and the result is better code quality. Also, I think Swift is the best option as it is always the first to receive the fastest updates, which means the newest gestures, libraries and UI tools can be used.

BACKEND DEVELOPMENT TECHNOLOGIES

Candidates

I have mentioned earlier that companies also create APIs internally. To make large project easier to manage, it is common to have a back-end and a front-end team decoupled. Generally, the front-end teams receive data from the back-end teams using APIs. The back-end generates the information required and creates endpoints from which the front-end can access this information. For my application, the mobile would be the front-end, which should use my back-end API to send and receive information.

For the backend technologies, I thought of two options: Elixir with the Phoenix framework, and Ruby with the Ruby on Rails framework. Compared to everything I tried before, Phoenix and Ruby on Rails are the frameworks I liked most, as the syntax promotes easy to write, readable code. However, Elixir and Ruby are completely different programming languages. Ruby is object-oriented, has extensive documentation and is widely used. Elixir is a functional programming language, very fast and good for concurrency. However, development time with Elixir/Phoenix is longer, so one must think how important the difference between processing time and coding speed is.

Chosen technology

Even though becoming more skilled in functional programming was appealing to me, I chose to go for Ruby on Rails, because it is more mature. Ruby on Rails favors convention over configuration (Quora.com, 2018), which means some smart people created good conventions which, if you follow, you will be more productive. Ruby lets you do many things in different ways, but conventions help developers from different parts of the world understand each other's code. I personally don't think I could create better conventions at this age, so I think using Ruby on Rails helps me create quality code even if I am not as skilled as a senior developer.

Moreover, Ruby on Rails 5 provides support for creating API's. The structure is there and rendering JSON[7] (the text format of the response) is very straight forward. This is a plus, as Phoenix does not provide this.

DATABASE AND SERVER

For the database and the server, there weren't many other reasons besides the fact that they work well with what I have chosen already (Ruby on Rails). I will still touch up on alternatives, though.

PostgreSQL

PostgreSQL is one of the most widely used relational databases (along with MySQL, Oracle etc.). It has "over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance" (Postgresql.org, 2018). Differences between relational databases are not huge, as they are all have SQL type querying. Generally, Ruby on Rails applications use either PostgreSQL or MySQL, and I went for PostgreSQL because it's known to be reliable and fast for large databases.

[7] JavaScript Object Notation (Json.org, 2018)

Compared to NoSQL, I find the relational model more structured. The trend with mobile applications is to use some sort of NoSQL (for example, Firebase, MongoDB or Cloud Kit). However, my database would connect to my back-end. I preferred relational over NoSQL because of the consistency. If the database is consistent, less error checking is needed, as I am always sure all the necessary fields always exist. In terms of speed, I was not worried as queries are done in milliseconds and would not be too slow even if everyone used the application in Cardiff.

Heroku

For hosting the server, I had two options: to do it with my hardware (a computer/raspberry pi) or host it on the cloud. The latter option was my preferred one, as it can be free, there is less set up and there is less chance it would crash (because skilled people manage these servers). There are multiple options for cloud hosting out there, but one of the most popular, which happens to work very well with Ruby on Rails, is Heroku. Heroku is different from others, because it provides the platform (web interface to manage the server) and the service (hosting on cloud), which makes it very easy to use (Heroku.com, 2018). Since this project's focus was the algorithm and the user interface, I considered I should do this part as easy as possible, and Heroku required very little knowledge about operations from my part.

External APIs used

Transport API

An essential block for building this project was having the bus timetables in Cardiff. It was obvious that copying every timetable from the Cardiff bus website was too time consuming. Therefore, I searched for an API from which I can query timetables for certain bus stops. My first idea was to use Google Maps API, as these bus stops were linked to locations (latitude and longitude coordinates). It seemed possible to retrieve bus stops, but I could not find much about timetables during my research. Therefore, I looked for alternatives and I found Transport API, "the most comprehensive transport data platform for the UK" (Transportapi.com, 2018). In the beginning, I thought I would use Transport API for timetables, and Google Maps API for location detection, but after making a few queries, I realized Transport API provided everything I needed. It would not make sense to use two API's, as both Transport and Google Maps API cost money after a set number of requests.

Postcodes.io

This API was used for a feature that I planned as optional in the beginning, so it was not necessary to create the project. However, I thought it would be useful if the user can search bus stops nearby a set location (by postcode) in addition to current location. In some cases, one might want to check when to leave work even if they are still home, and with this feature a user could check nearby stops to the workplace's postcode. To implement this, I needed the coordinates (latitude and longitude) of the postcode, and this is what Postcodes.io provides (Postcodes.io, 2018). What was appealing to me was that, compared to other similar APIs, it is completely free.

PLANNING

In my initial plan, my tasks were quite detailed, and even though it helped me understand my tasks clearly, sometimes I could not finish all in the week they were planned. Therefore, I decided to also set some milestones and have a rougher plan to follow, structured in a Gantt chart.

Milestones

- Have real departure times recorded in the phone locally by week 4;
- Have delays computed from real departure times by week 7;
- Have an online server working by week 8;
- Be able to query the timetable by week 8;
- Finish basic functionality in the phone by week 9;
- Finish at least two optional tasks by week 10;
- Have a draft report by the end of week 11.

Gantt chart

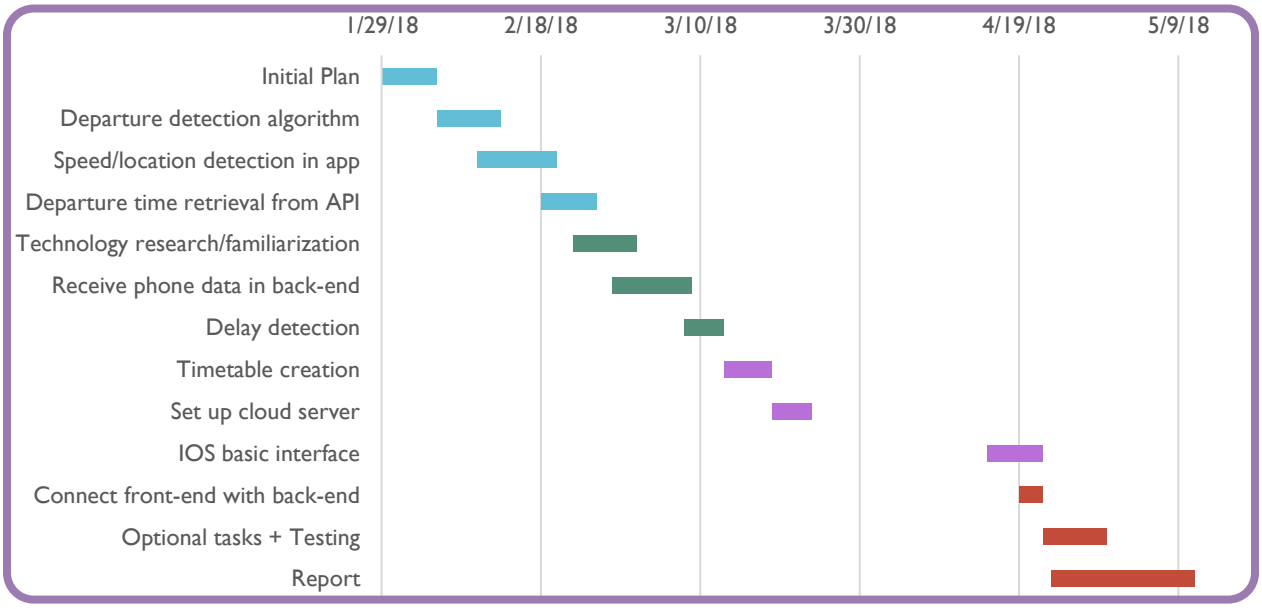

FIGURE 1

USE CASES

Use cases represent “a commonly used business analysis technique that captures requirements for a software application” (Bridging-the-gap.com, 2018). I think they are vital for explaining how the user is expected to interact with the system. However, if I was working in a team doing Agile development (methodology based on incremental planning and improvement), I would have used user stories. Compared to use cases, user stories require less planning, therefore leaving the option to decide on details on the spot (which might be beneficial if the team understood the requirements well).

Since I have no team to dispute my ideas with, I decided to go for use cases, as they provide a better overview of the system in this report.

Use case diagram

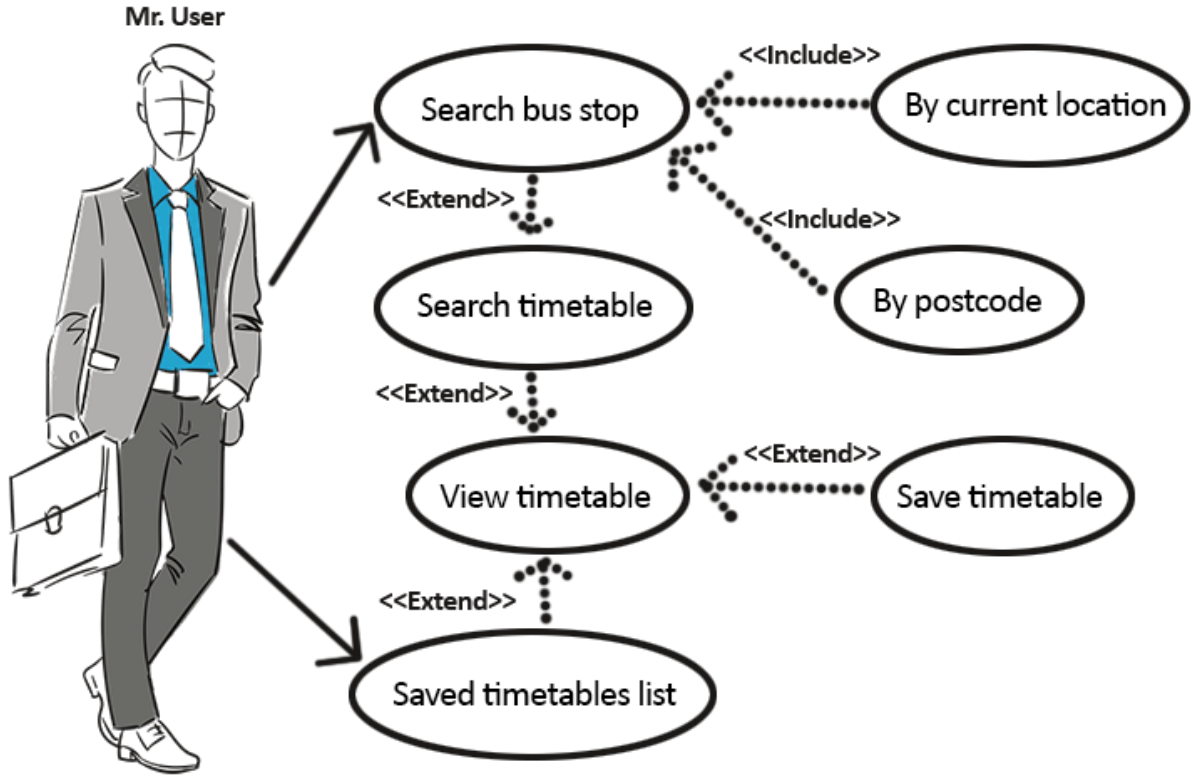

FIGURE 2

UC 1: Search bus stop (by current location)

Description	The user can select to search for bus stops nearby its current location.
Actors	Public transport user/commuter
Preconditions	Application is open
Postconditions	"Search timetable" page open (UC 3)

Basic flow

1. Click on "Search" tab (if not there already).
2. Turn on switch with "Use my current location" label.
3. Click on "Find" button.

Alternative flow – no nearby bus stops found

1. Click on "Search" tab (if not there already).
2. Turn on switch with "Use my current location" label.
3. Click on "Find" button.

4. Receive popup mentioning there are no nearby bus stops.
5. Use UC 2 to achieve the postcondition or exit.

UC 2: Search bus stop (by postcode)

Description	The user can select to search for bus stops nearby a given postcode.
Actors	Public transport user/commuter
Preconditions	Application is open
Postconditions	"Search timetable" page open (UC 3)

Basic flow

1. Click on "Search" tab (if not there already).
2. Input desired postcode.
3. Click on "Find" button.

Alternative flow – postcode invalid

1. Click on "Search" tab (if not there already).
2. Input desired postcode.
3. Click on "Find" button.
4. Receive popup mentioning postcode is invalid.
5. Retry or exit.

UC 3: Search timetable

Description	A page where the user inputs the necessary information for finding a timetable. Bus stop, date, time must be selected. There is an optional field to also search for a particular bus line.
Actors	Public transport user/commuter
Preconditions	User has found nearby bus stops to a location successfully (UC 1/UC 2)
Postconditions	"View timetable" page open (UC 4)

Basic flow

1. Select a nearby bus stop using the picker.
2. Select a date using the date picker.
3. Select a time using the time picker.
4. Click on "Find" button.

Alternative flow – add bus line search

1. Select a nearby bus stop using the picker.
2. Select a date using the date picker.
3. Select a time using the time picker.
4. Turn on switch with "For bus line" label.
5. Input bus line
6. Click on "Find" button.

UC 4: View timetable

Description	A page containing a list of timetable items. Each cell should contain bus line, bus terminal, aimed departure and expected delay, along with a reliability icon. A bar at the end of the table should explain the reliability icon, and a heart icon should be present on the top navigation bar.
Actors	Public transport user/commuter
Preconditions	User has found nearby bus stops to a location successfully (UC 1/UC 2)
Postconditions	None

Basic flow

1. Read the information in the table to find out the desired departure time, along with the delay.

Alternative flow – Reliability icon information

1. Scroll to the end of the table
2. Click on the information icon on the bar explaining reliability icons.
3. Read the explanation.
4. Click on “OK” button.

UC 5: Save timetable

Description	An action present on the timetable page (UC 4) that allows the user to save a timetable he might access often. The button is present on the right side of the top navigation bar, as a heat. If the heart is red, it’s been saved, and if the heart is empty with a black border, it’s not saved. Clicking on the heart will changed the saved status.
Actors	Public transport user/commuter
Preconditions	On the “View timetable” page (UC 4)
Postconditions	Saved status of the timetable changed

Basic flow

1. Click on the heart icon.
2. If it was red, it will turn into an empty heart with a black outline or vice versa. As a result, the saved status is changed (saved if it became red, not saved otherwise).

UC 6: Saved timetables list

Description	A page containing a list of saved timetable searches. Each cell should contain the bus stop name, time, day of week, and optionally a bus line.
Actors	Public transport user/commuter
Preconditions	Application is open
Postconditions	On the “View timetable” page (UC 4)

Basic flow

1. Click on the “Saved” tab.
2. Click on the desired saved search to see its timetable.

IMPLEMENTATION

ALGORITHMS

Bus departure time detection

To detect delays, we need to detect the actual bus departure times, compared to the timetable's departure times. To achieve that, we use the user's phone's location and speed. When using the iOS standard location library, CLLocationManager, the developer can start and stop receiving location updates (Apple.com, 2018). Every location received is a list of multiple locations, which means iOS generates automatically the speed from the difference in time and distance between the last location recorded and its previous one. Using this library, I set up my application to receive location updates (which also include speed updates) with an accuracy of 5 meters, even if the application is not open (runs in the background). The code written to create location updates can be found in the appendix section 1.

Filtering and saving locations and bus stops

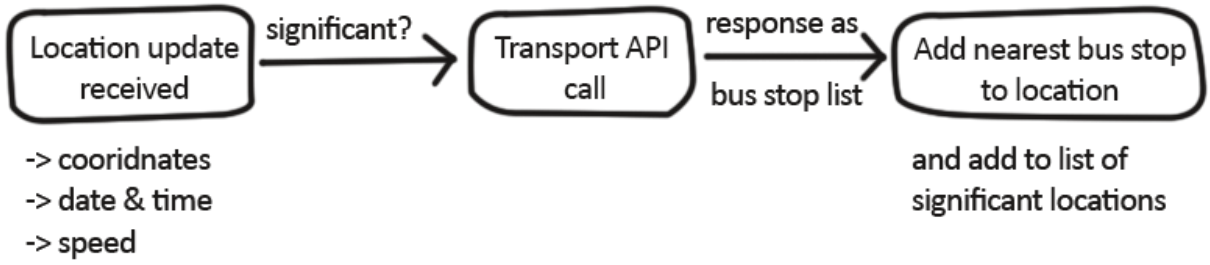

FIGURE 3

Once received, the updates can be saved and processed using my algorithm. To avoid over processing, and to reduce phone resource usage, I saved only "significant locations" for later processing. The conditions for selecting a location as significant are:

- There is no other location being processed when the new one will start being processed (thread lock);
- It is not identical to the last saved location;
- If the distance to the nearest bus stop is more than 200 meters – the time difference between the current and last location must be at least 40 seconds (if walking speed) or 20 seconds (if driving speed)
- Otherwise if the distance is less than 200 meters – the time difference must be at least 20 seconds (if walking speed) or 10 seconds (if driving speed)

When a location passes the requirements (marked as significant), an API call is made to the Transport API to retrieve the nearest bus stop. Then the location is added to the list along with all the details of its nearest bus stop. The code written to filter and save locations and bus stops can be found in the appendix section 2.

Detecting user is entering the bus

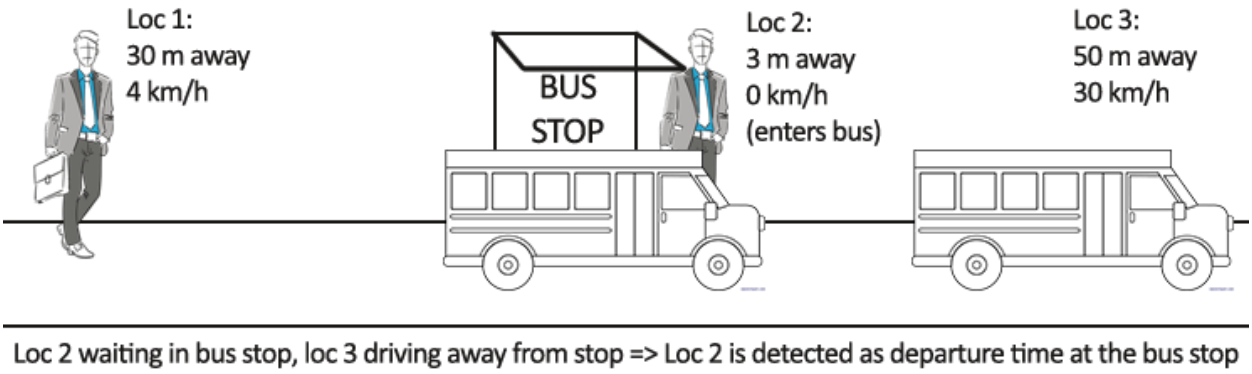

FIGURE 4

We detect the user enters a bus if he is at least 25 meters away from a bus stop, moving with walking speed (less than 15 km/h). A location with these properties must be recorded. Then, if the next location recorded is more than 25 meters away from the bus stop, moving with driving speed (more than 15 km/h), we assume the user took the bus at the nearest bus stop to the first location. If the user waits a while for the bus to come, multiple locations might be recorded, with undefined speed (noted as -1). The speed can only be undefined if the user is not moving, therefore we consider that case, too. If there are multiple recordings of the user being nearby a bus stop, the one with the latest timestamp (closest to the departure time) will be selected. We consider both standing and walking valid as "waiting in station", because the user might either walk fast to catch the bus, move inside the bus to find a seat, or simply move around the bus stop while waiting.

This part detects the departure time, which is the closest timestamp recorded before the bus left. It is recorded even if the user is in the bus, for every stop the route goes through.

Detecting user is leaving the bus

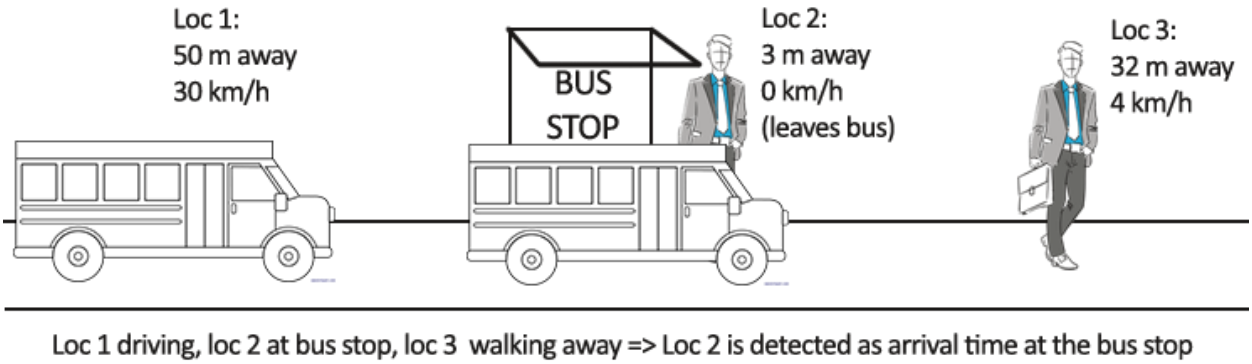

FIGURE 5

When leaving the bus, only the arrival time can be detected, because after leaving the bus, the user will walk away from the bus stop. The arrival time is calculated inversely compared to the departure time. It

is more accurate though, because the algorithm knows at what location the last departure was. So, instead of looking if one recorded location has driving speed and the next one recorded is near a bus stop, the algorithm looks at all the locations recorded since the last departure time.

The algorithm will check if any of the previous locations have driving speed, while the current location has walking speed (or not moving). This is to avoid the scenariOS when there is traffic and the bus might drive very slow just before it arrives to the station. If there is traffic, vehicles usually travel in sprints, therefore I am not worried that a driving speed won't be recorded. However, if through a whole station it can't be recorded, the algorithm can't be sure that the user was still in the bus, so it won't record any arrival. Since the algorithm records all arrival and departure times, the chance none would be recorded is quite low.

Selecting and saving departure times

As explained above, this algorithm is designed to detect not only when the bus departs, but also when it arrives, for every bus stop the user goes through in his journey. In the UK, there is generally not a long wait between arriving to and leaving the station (usually seconds), so I think it is worth recording the arrival time, even if it might be a bit earlier than the actual departure time.

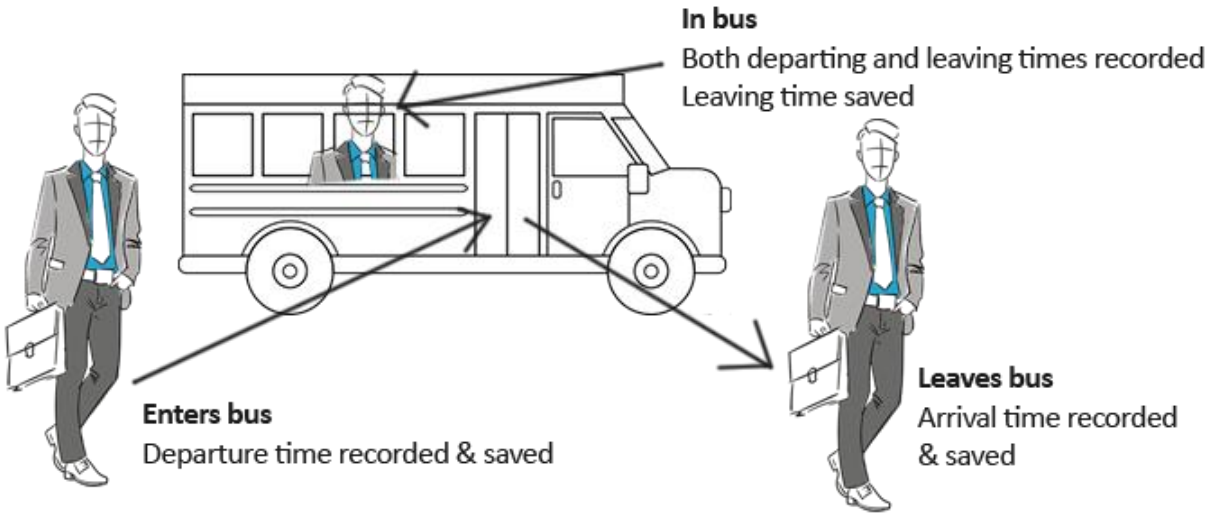

FIGURE 6

At this moment, the algorithm will have a list of both arriving and leaving times. If both departing and arrival times are present, only the departing time is saved. This last part of the processing takes all the locations previously selected, and once again, filters by deleting the arrival times that are irrelevant (code can be found in appendix section 3). Finally, the processed list of locations is sent to the back-end I created, using API calls. Two POST requests are made asynchronously, for adding both the bus stop and location information.

Delay detection

The delays are detected in my back-end application, created with Ruby on Rails and Postgres. When the mobile application sends locations and bus stops by making API calls, the back-end adds delays to locations automatically.

Firstly, to retrieve expected departure times, we use Transport API to retrieve a timetable for a selected time, it date and bus stop. The response API is saved and used for processing the delays. To include all scenariOS, I created the algorithm so that detects both negative or positive delays. For every item in the timetable, it checks if the delay (expected minus actual departure time) is smaller than any previous delay from the timetable (code can be found in appendix section 4). Then it adds to the database the delay, aimed departure time and the bus line from which the delay was processed. This is how we make a link between a location recorded and a bus line.

Expected delay calculation

This last algorithm generates the expected delays seen on the interface when a timetable is queried. It will average all delays of the locations that match with a timetable entry.

For each item in the timetable, we query all locations that have the same bus line and time. We also consider weekdays and weekends as different, so, for example, if the timetable query is made for Tuesday, we take all the locations recorded in weekdays (Monday to Friday). If there are no locations where the bus was expected to leave at the exact time, the algorithm will query for all times between one hour before the expected time and one hour after (code can be found in appendix section 5). This is because usually the average between the delays around the time when the bus leaves is usually similar to the actual time. For example, since rush hours are 2-3 hours in the morning and evening and at those times it is expected to be heavy traffic, we assume that a bus departing at 8 AM is usually as late all buses departing from 7 AM to 9 AM. However, if there are little records in the database, this would not be as accurate. To avoid confusions, I decided to let the user know if the detected delay is highly accurate or not. We do that by calculating the number of relevant locations we found and sending it to the front-end, along with the expected delays. In the end, we add to every timetable item these fields: “delay”, “record_count”, “expected_departure_date”, “expected_departure_time”. Afterwards, the modified response is sent to the front-end.

System Architecture

Class Diagrams

This section explains the architecture on which the algorithms described above run. The back-end and front-end have some common classes (with common attributes) because the data generated in the back-end needs to be used in the front-end and vice versa. While some classes generate data and send it, others are just blueprints of the response structure.

To structure both the back-end and front-end, I decided to use MVC (Model-View-Controller), a popular design pattern for both web and mobile development. MVC splits the application in three parts, each one

of them having different roles. The model should handle everything data-related (querying and processing), the view should display the information to the user (through tables, navigation or others), and the controller should tie the other two together. The model should contain reusable functions, while the controller should handle processing that is only needed to display the data as required in the view.

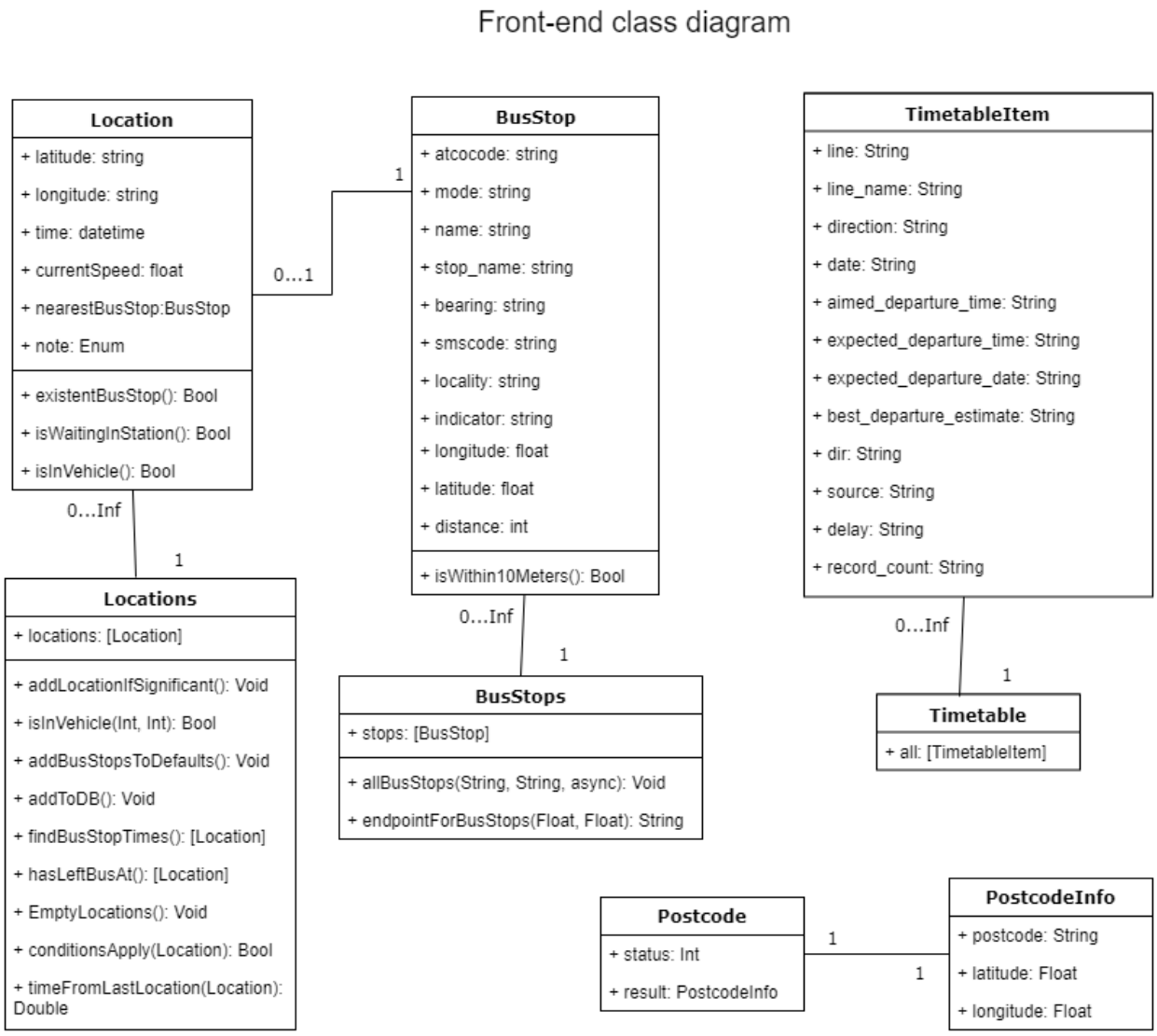

FIGURE 7

In iOS development, the split between the view and controller is quite faint. Views are created using the Storyboard, an interactive tool where developers can create pages the user will see and use. Then, to send information to the views, ViewControllers are used. A ViewController can only correspond to one view, and they can access elements inside the view (such as labels, tableViews, inputs etc.) using IBAction. For the front-end class diagram (figure 7), I decided to only include the models, as the controllers are too close to the views, and the numerous functions used to access the phone screen do not picture the functionality this application does (which is generating and sending locations).

Swift also comes with its own views and controllers, in its “UIKit” and “Foundation” libraries, which can be used to implement custom views and controllers. In my project I am using views such as “UIButton”, “UILabel” and “UIView”, and controllers such as “UIViewController” and “CLLocationManager”.

Models are based on entities/objects that have data stored inside. In general, in mobile applications, data is retrieved from an API or a JSON file. For the front-end class diagram, “BusStop”, “BusStops”, “PostcodeInfo” and “Postcode” classes are footprints of the JSON responses coming from external APIs (Transport API and Postcodes.io). “Timetable” and “TimetableItem” are footprints of the JSON responses coming from my internal back-end application. The singular along with plural class pattern can be observed here, and this is because most JSON responses are nested (sometimes more than one nesting). When we retrieve a list of items and we don’t have a database to keep the state, we need to have a class/structure that allows us to hold multiple items that are related. And that’s why most plural classes just have one attribute, which is an array of the singular related class. For Postcode, it was nested with only one element inside, and unfortunately, we had to follow the structure of the API when decoding from JSON to a Swift class. However, if I would have made that API, I wouldn’t have nested the latitude and longitude in “result”.

Back-end class diagram

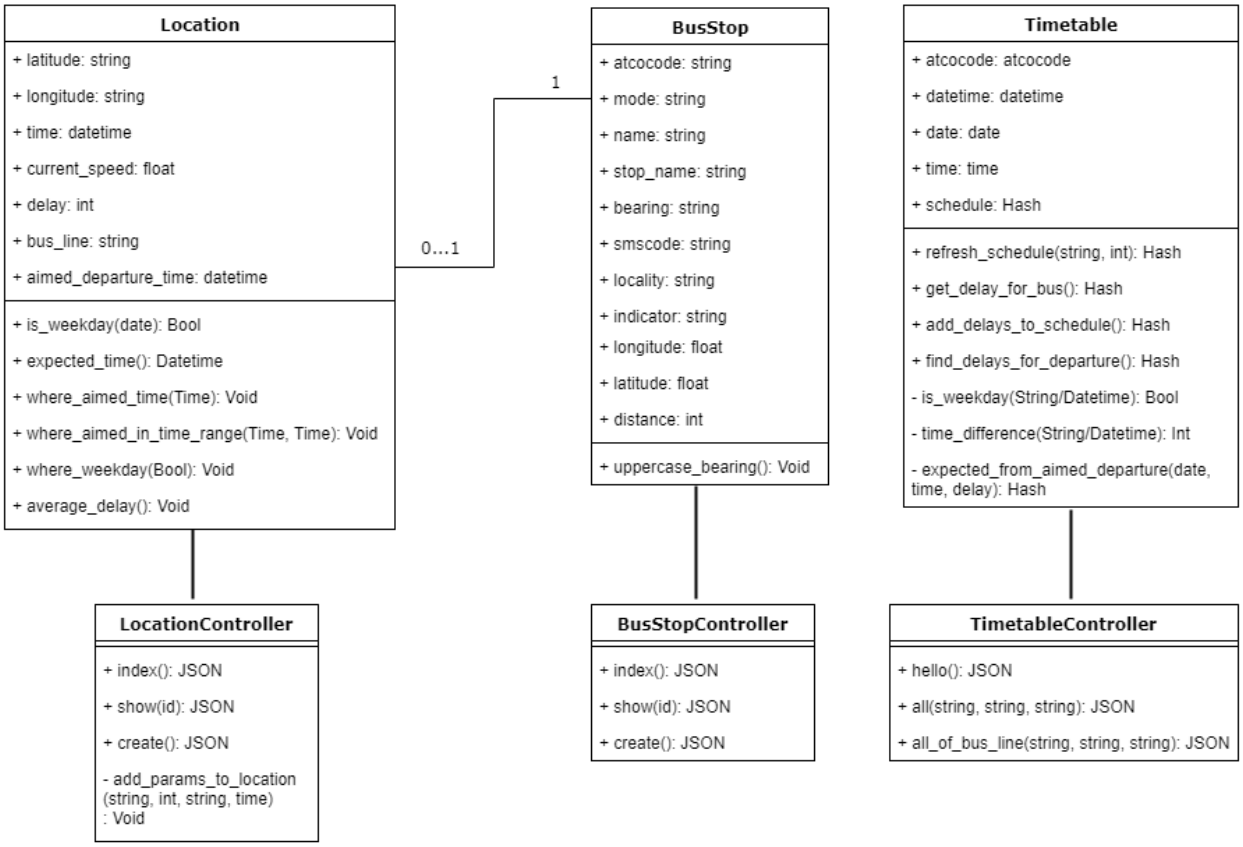

FIGURE 8

In web development and specifically Ruby on Rails, models, controllers and views are well defined with a clear split between them. For that reason, I added controllers in the back-end class diagram for explaining how the data is sent to and retrieved from the front-end. In Ruby on Rails the convention is that every controller represents one model, and every function in the controller represents a route (and a view except for POST/PATCH requests). However, you can access any model from any controller.

Models are representations of the database. They contain attributes according to the schema of a table and contain functions that process data from the database. The "Location" model contains short functions that can build up a query (WHERE statements, for example), such as "where_aimed_time(Time)". For my use case, I needed a model that, instead of retrieving data from the database, would retrieve data from an API according to some parameters. "Timetable" receives data from Transport API and provides functions to process it. However, it is not inheriting any Rails class, it is purely a Ruby class.

Routes are endpoints on the web a user (or a machine) can access, such as "http://website.com/login". For every route path the API provides, there is a function in a controller that processes information before sending it as a response. For example, the "all" function in "TimetableController" retrieves the timetable for a bus line, date and time. It will use functions in the "Timetable" model to generate it, and then render it.

In traditional web development, views would be what the user sees, so HTML and JavaScript. But, because the back-end application is an API, we don't need to write any HTML. The views will be JSONs, created from hashes and rendered automatically by rails. Controllers will have a "render JSON" line at the end of each of their functions to let Rails know what to render, instead of rendering views. For this reason, there is no need to have any file in the "views" folder.

In Ruby on Rails, the database structure (called schema) is saved within the project. Therefore, any developer can run `rails db:migrate` in the terminal to add the database to their computer. The schema file can be found at `db/schema.rb` inside the project and its content is added in appendix section 7.

Network diagram

The diagram below (figure 9) is further explaining how different parts of the system are connected. The front-end Swift application is a different source of data coming from every user with the application installed. Different processing is done on every phone, such as generating locations, accessing the Transport API for gathering bus stop data and Postcodes.io to retrieve latitude and longitude of a postcode. The mobile application does not directly retrieve data from the database. Having credentials for accessing the database in every phone is very unsecure as the user might somehow access the code. Possibly for this reason, Swift does not provide any in-built library to access databases. Therefore, locations are sent to the database through POST requests to the back-end API. If the request passes validations, it is saved.

The back-end API has direct access to the database, and both are hosted on the Cloud, using Heroku. It can receive or send data, using GET/POST requests. It will receive bus stop and location data, while it will send timetable data. When saving locations, the server will automatically process delays and when retrieving timetables, it will automatically calculate expected delays.

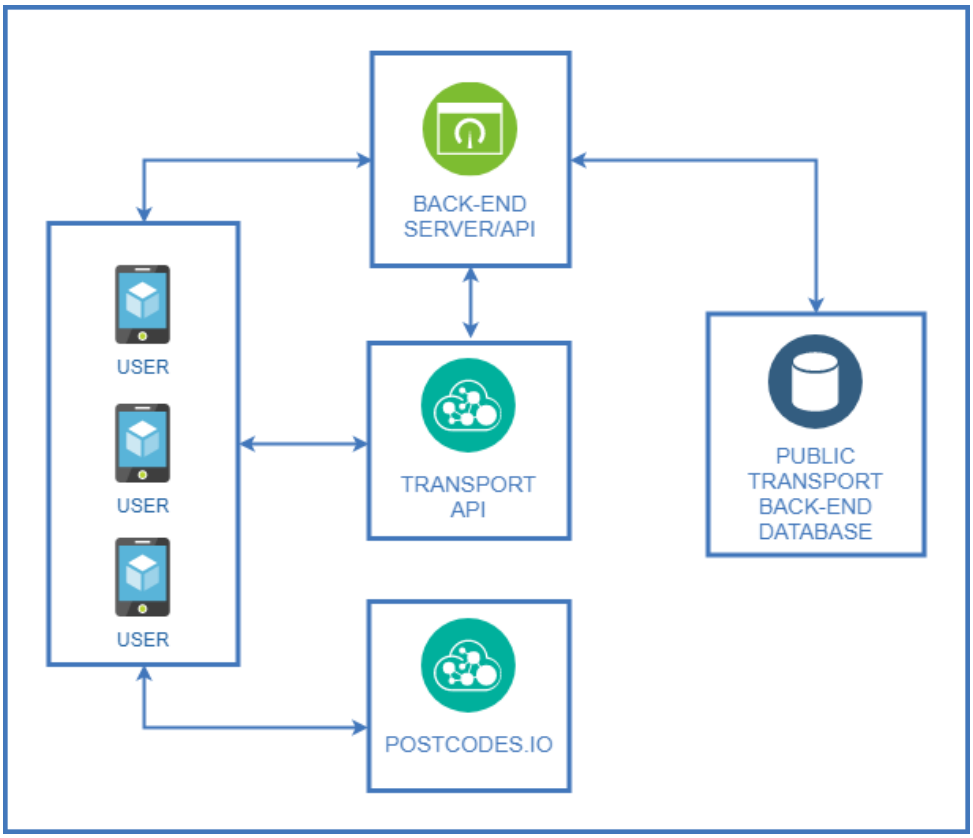

FIGURE 9

Database model

As we are using a lot of external data, the internal database does not need to be very complex. "Bus_stops" has the same attributes as the JSON response that retrieves a list of bus stops. This information is extracted as we need to generate delays for timetables of bus stops. "Locations" contains both data generated on phones and on the server. "latitude", "longitude", "time" and "current_speed" are received from the phone, and, using these fields and external API requests, "delay", "bus_line" and "aimed_departure_time" are generated and saved into the database.

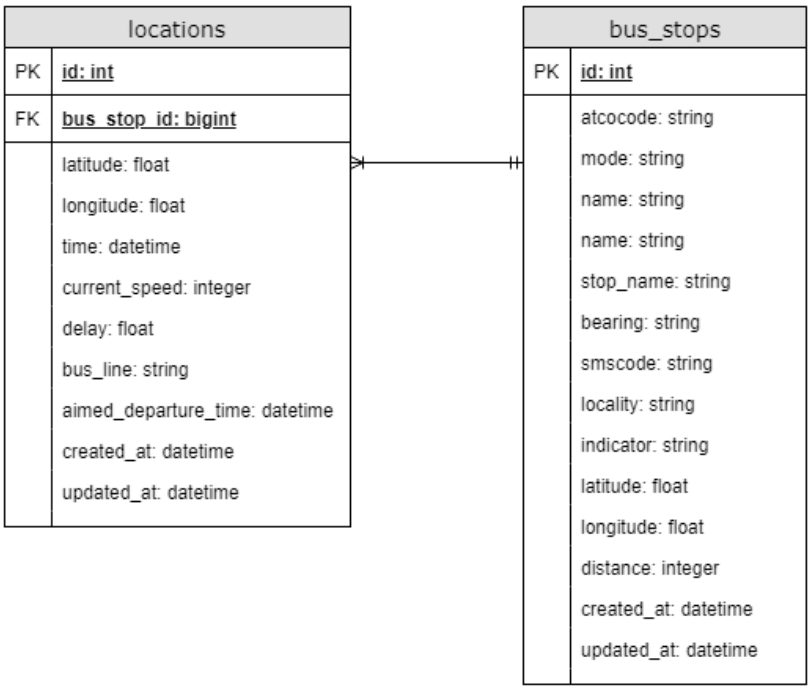

FIGURE 10

I initially designed “bus_stops” to have a one to many relationship with “locations”. However, as I was mirroring API requests, every location created has only one bus stop. Even though at the moment the one to many relationship is not needed, it still acts like a one to one relationship and leaves space for further work on refactoring to a better solution of saving data.

INTERFACE DESIGN

Interface behavior was explained above in the “Approach” section, using use cases. Therefore, this section will focus on guidelines followed and usability, as the features were explained already.

Nowadays, iOS applications generally have simple, sleek designs. “Early iPhone interfaces were full of custom graphics until iOS 7 established a flatter, more minimal design. Many apps followed suit to maintain consistency between the OS and app experiences” (Kaufman, 2016). The mobile application I created tries to follow a sleek design too, along with many other guidelines created by Apple.

Navigation

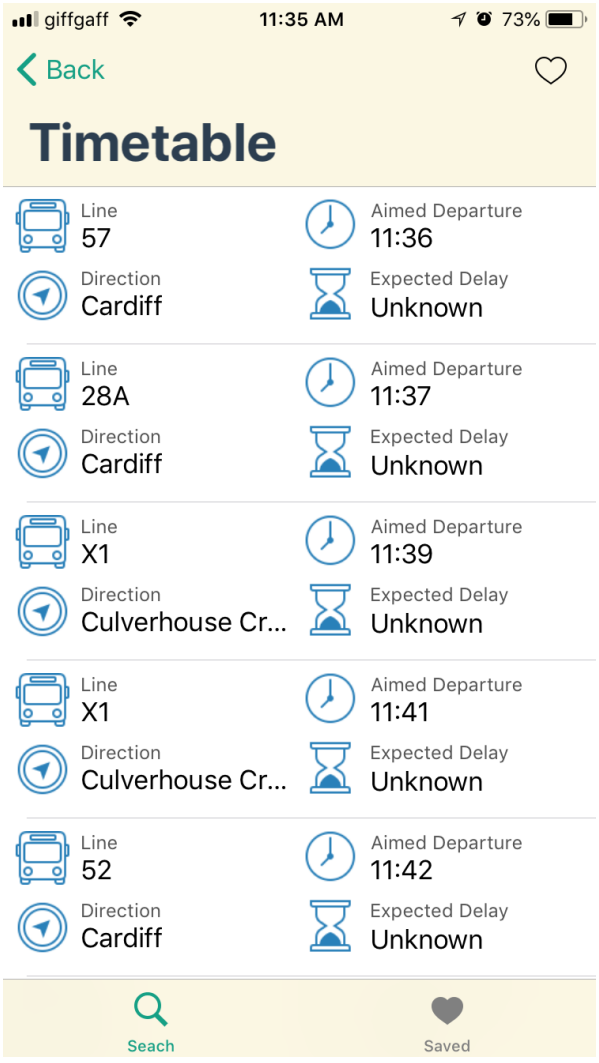

FIGURE 11

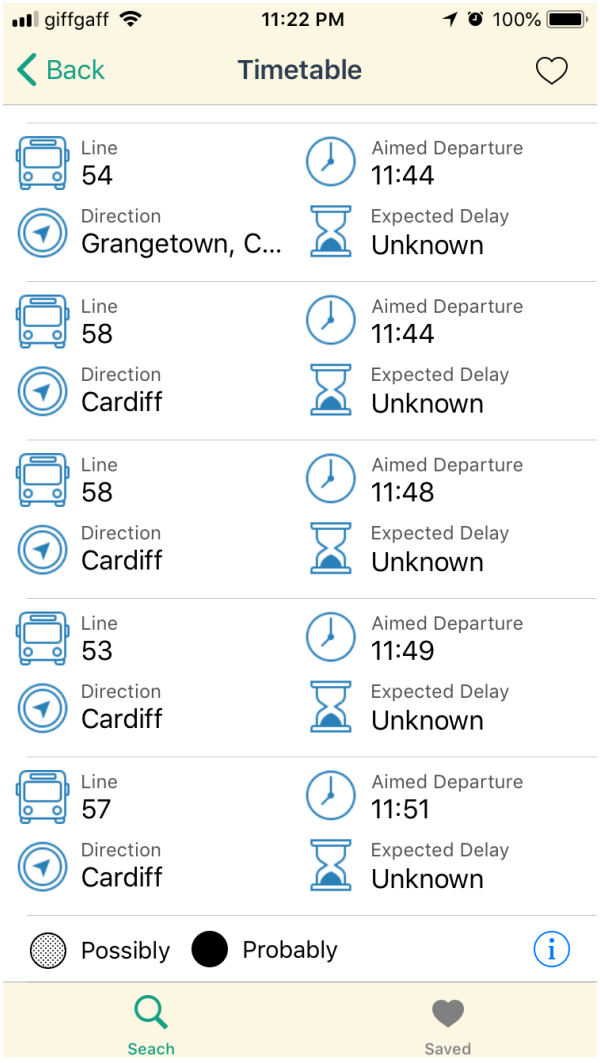

FIGURE 12

The application structure in terms of navigation is very important, as it determines in how many steps the user can achieve a goal. Apple suggests in their human interface guidelines to “design an information structure that makes it fast and easy to get to content”, that require a “minimum amount of taps” (Apple.com, 2018). The iOS SDK offers multiple options in terms of navigation, and sometimes it is suggested to mix and match navigation models to fit the application purpose (Kaufman, 2016).

For my application, I considered that flat navigation is most intuitive, as I had two ways the user could achieve the goal, which is reading the timetable of a bus stop. The user can access the timetable either by search, or by selecting an already saved timetable. To support this, I used both tab bars and navigation bars. The tab has the highest priority, and each tab has its own navigation bar. If a user goes into a deeper page, where the navigation bar shows the option to go back, and then clicks on a second tab, the state of the navigation on the first tab remains.

The images above (figures 11 and 12) picture both types of navigation I am using. If interaction has been done on the previous page to navigate to a different page, a button that lets the user navigate back is shown on the top navigation bar. The page title is also shown, but it is not required. Apple suggests to “consider showing the title of the current view in the navigation bar” (Apple.com, 2018), which was my choice, too.

The large title styles I am using have been introduced recently, with iOS 11. Apple recommends using them to “provide extra emphasis on context” and suggests using them to “clarify the active tab and inform the user when they've scrolled to the top” (Apple.com, 2018). Both suggestions are applied for my interface, as I am using tabs, and when switching to a new tab the big title lets the user know what page they are on. The scrolling is also present on the timetable page (as shown on figures 11 and 12), and at the top of the page, the title appears big, but when the user scrolls down, the title minimizes.

In terms of tabs, they should be used “strictly for navigation” and to “avoid having too many” (up to five) (Apple.com, 2018), which are guidelines I followed. Also, tab bar icons should be “visually consistent and balanced”. I have achieved that by having tab icons of the same size, and while the search icon is consistent with Apple’s standard search icon, the saved icon is consistent with the icon that marks timetables as saved.

Data entry

Inputs are very important, as this application’s main purpose is to search for information. Multiple types of inputs are used to search for timetables. Figures 13, 14, and 15 present use cases 1 and 2, where the user can search for a nearby bus stop or near a postcode. The “Search Bus Stop” view presents a postcode input and a switch that if set as “on”, the postcode input becomes disabled and it will use the user’s current location. If the user inputs a postcode, the phone will retrieve the latitude and longitude of the postcode (using Postcodes.io), and otherwise it will retrieve device’s latitude and longitude. If the postcode is invalid, an error will appear with an error message, and the field will be cleared. A “Disabled” placeholder will also appear when the user switches the usage of current location on, which notifies of the field status, and that the user cannot input a postcode if they are using the phone’s location.

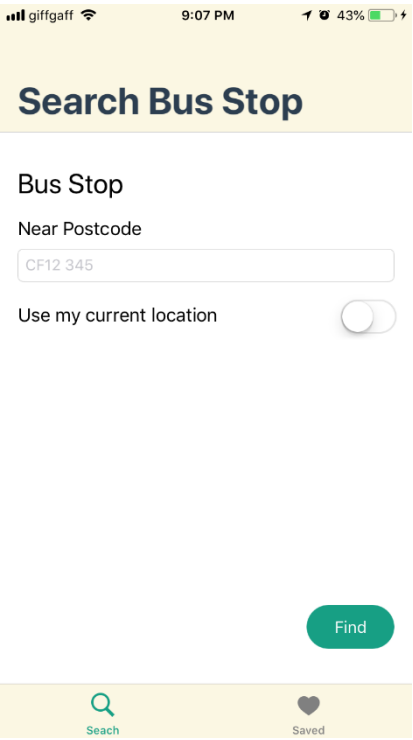

FIGURE 13

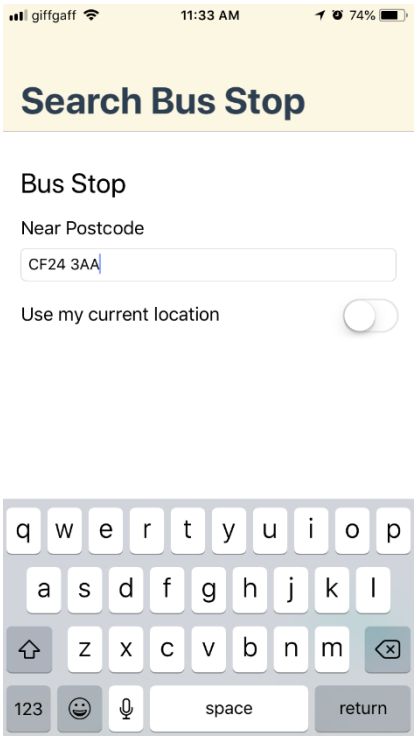

FIGURE 14

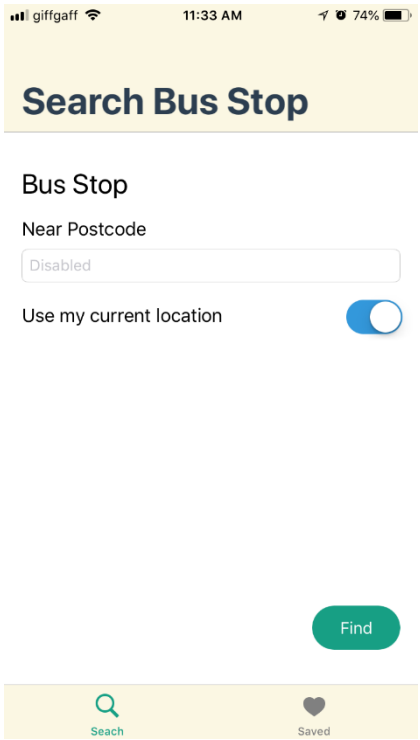

FIGURE 15

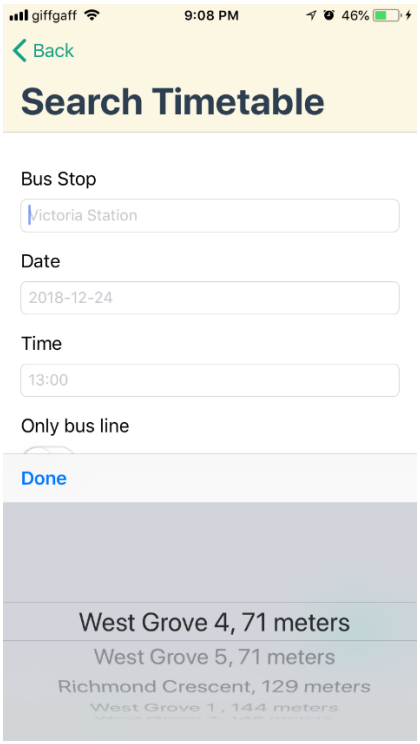

FIGURE 16

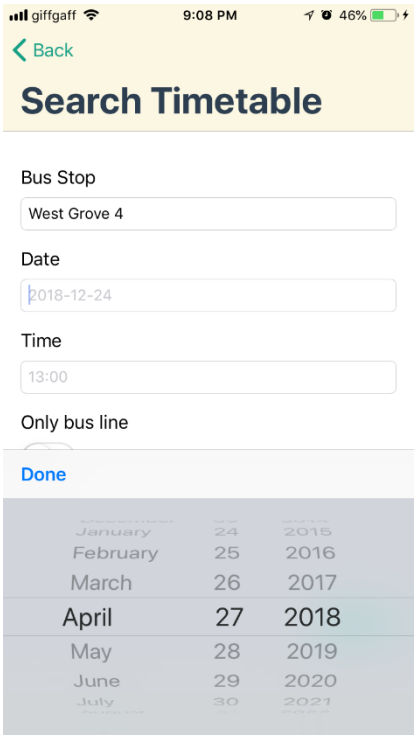

FIGURE 17

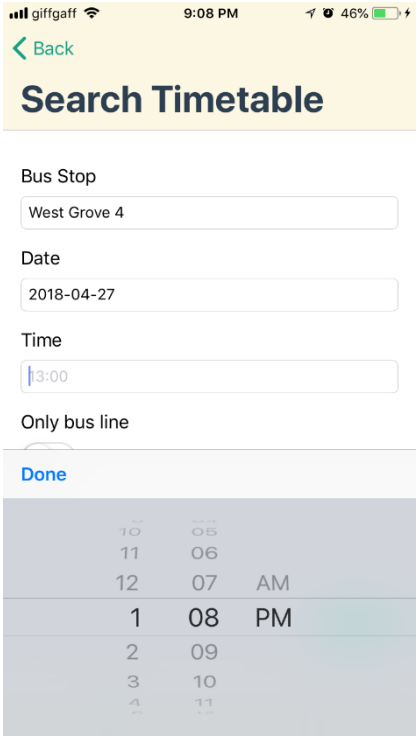

FIGURE 18

Nowadays, the convention on iOS app design is to not use labels, because the user should know what a field is after completing is. However, I agree more with the Android style, which includes labels. I believe it is important to present explicit information. Since this application should cater to all ages, elderly people might not understand what a field represents clearly or might not remember what the input was. Therefore, I think it's more explicit to use both labels of what the field input should be, and placeholders of an example of information that should be input. An iOS guideline is to "provide reasonable default values", as it "minimizes decision making" (Apple.com, 2018), which is achieved through placeholders.

For the "Search Timetable" page (figure 16, 17, 18), I am not using text input fields, which is following the rule of presenting choices if possible, to "make data entry as efficient as possible" (Apple.com, 2018). For the "bus stop" field I am using a picker view, for "Date" I am using a date picker view and for "Time" a time picker view. To minimize the space used on the screen, I am creating these views as popovers, so that when a user clicks on the text field, a picker appears. This also creates focus on the current field. These pickers are standard for iOS devices, provided by apple, therefore are consistent with iOS applications. For example, when setting up a new alarm on iPhones, time input is done through a time picker.

The "Only bus line" switch lets the user query the timetable for a bus line, which is optional. The switch is a good option for avoiding confusion when having a "bus line" field always present. If the user wants to search for a bus line, he will turn the switch on and a text input field will appear. If the user switches "only bus line" off, the field will disappear.

Error handling

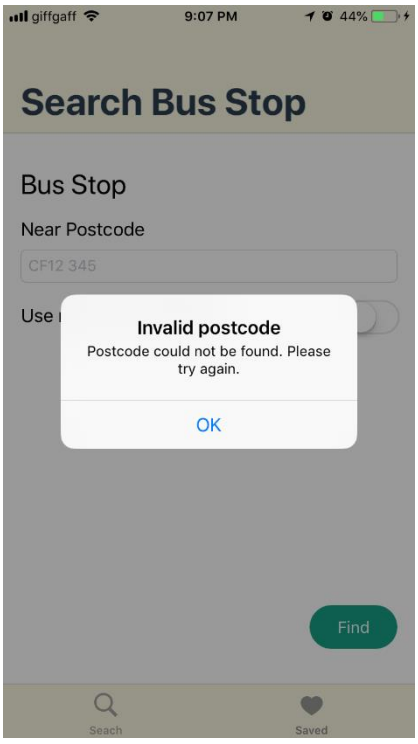

FIGURE 19

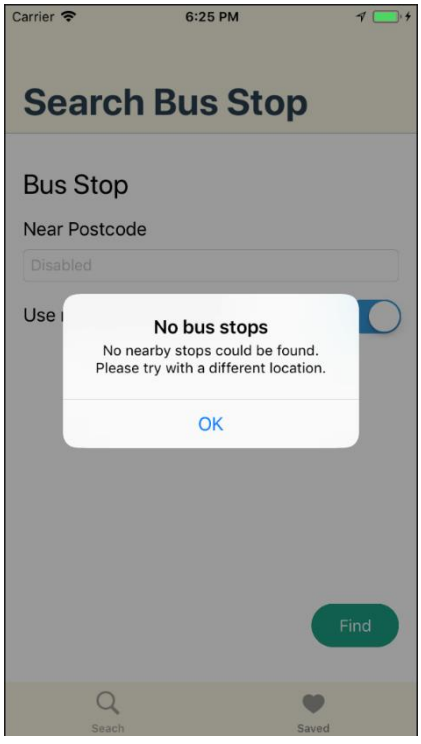

FIGURE 20

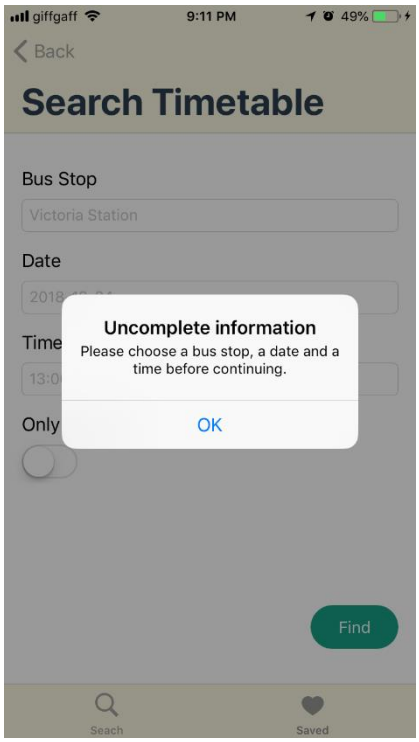

FIGURE 21

While completing forms, errors can arise. In most cases, it's because input is invalid, and this must come from the server, as a response. To notify the user about the errors, I have decided to use alerts (figures 19, 20, 21), as they are quite easy to implement and deliver the message. It is not recommended to use alerts too often, but they can be used in situations like "notifying people about problems" (Apple.com, 2018). However, if I had more time, I would have made more research for alternatives, and maybe chosen to use alerts only for severe errors, as they disrupt the flow.

On the "Search Bus Stop" view, I am showing an alert if the postcode is invalid (figure 19), or if there are no nearby bus stops when using the current location (for example, if the user tries to use the current location while not in the UK) (figure 20). On the "Search Timetable" view, an alert will pop if the fields that contain the selected picker item are empty (figure 21). It is suggested to "write short, descriptive, multiword alert titles" (Bridging-the-gap.com, 2018), which I think I have achieved. The user will clearly understand the problem from the alert title, even if he does not read the alert description before he closes.

I am also using an alert for an explanation on the "Timetable" view (figure 22). I am not sure this is the best option, but two sentences would have been too little to open a full new view. I will discuss my ideas on improving this in the "Future work" section. However, at the moment, if the user clicks on the information button on the last cell of the timetable (that describes the meaning of the half empty and filled circles), an alert will pop up to shortly explain what this means.

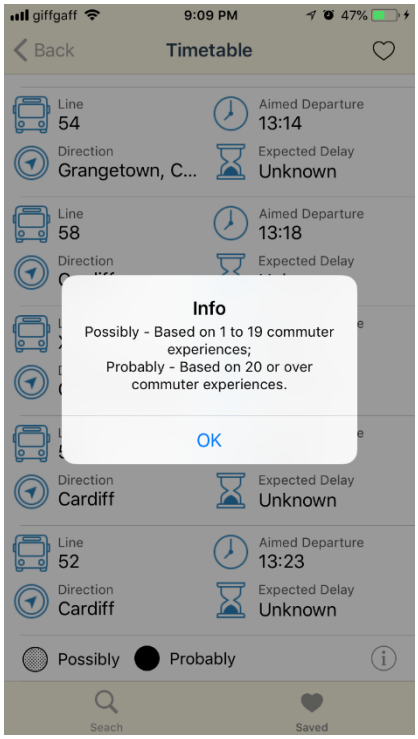

FIGURE 22

Visual design

Good aesthetics/design on an application is always a plus and can make the user feel better while using it. Also, having a dedicated color scheme is part of the branding of the application, which can represent a business. "Great apps express unique brand identity through smart font, color, and image decisions. Provide enough branding to give people context in your app, but not so much that it becomes a distraction" (Apple.com, 2018). This goes hand in hand of Apple's decision to employ minimal design. I think I have managed to follow both guidelines in my application.

Apple recommends to "choose a limited color palette" and to "use complementary colors" throughout the app. My color scheme is quite simple. It includes blue and teal, with light yellow background on the navigation and tab bars. I am also using a very dark tone of the teal on the large titles, as I think it's less harsh than black. All the text, icon and button colors (teal and blue) are complementary with the yellow used in the background, which makes the application visually appealing. Also, the text and icon colors are easily readable on both white and light yellow. It is also suggested to "consider choosing a key color to indicate interactivity". My key interactivity color is teal, which is used on search buttons and on tab bars.

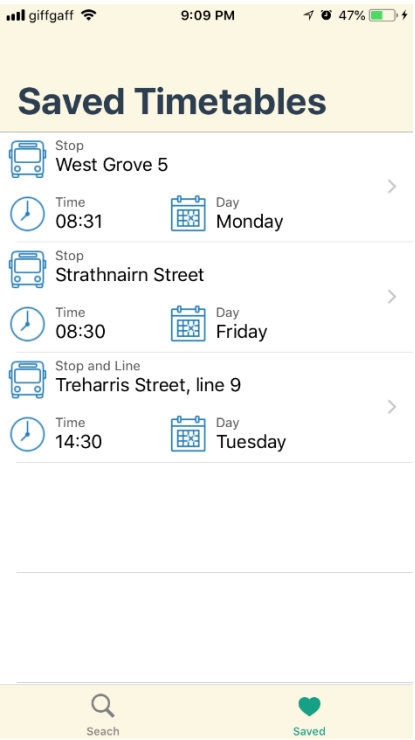

FIGURE 23

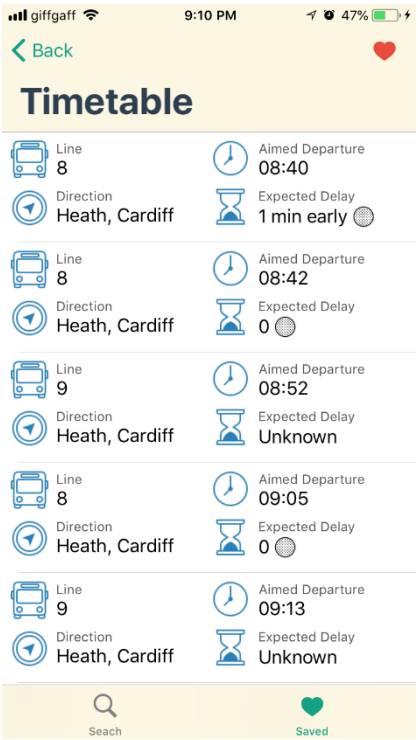

FIGURE 24

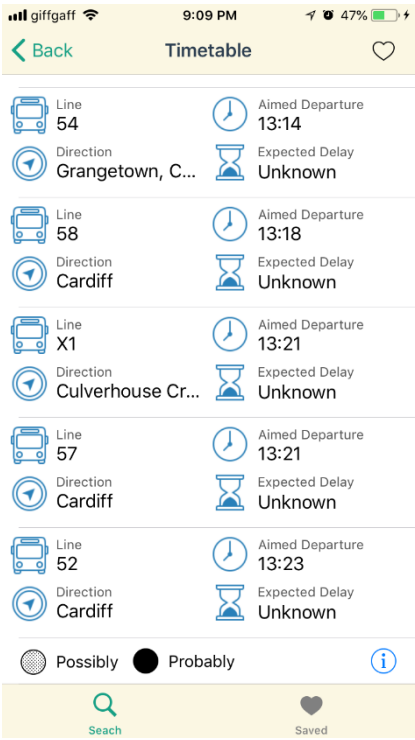

FIGURE 25

The icons are used to visualize the meaning of text for every timetable item (figures 23, 24, 25). They are used on the "Timetable" and "Saved Timetables" pages, and all have a blue color. In my opinion, not only they are intuitive and makes the user understand the information even before reading the label, but they also give a good look to the application.

In terms of layout, the application should be usable on any phone (different screen sizes, resolutions), and both orientations (portrait and landscape) (Apple.com, 2018). To achieve this, auto layout is used. When creating views, Swift lets you add constraints that specify how different items should adapt to the screen size. For example, for the first field on a page, I set left constraint to 8px, top constraint to 16px, and right and bottom constraints adaptive to the screen size. In the beginning, constraints can be quite confusing, but XCode does a great job in telling you where constraints don't match. It is also recommended to "ad-here to the safe area and layout margins defined by UIKit" (Apple.com, 2018). For iPhones, it is 8px to the edges of the screen, which I have respected. Also, to make the application look consistent, I am only using multiples of 8 as margins, which makes spaces proportional. For example, I am using 16 as top constraint when I am adding a title, or between fields, which is 8*2.

EVALUATION AND RESULTS

Testing the final product works is easy in theory: take the bus and see if actual departure times are recorded and sent to the database. However, taking the bus many times was a waste of time and money for me, as I rarely take the bus while in Cardiff. However, using other testing methods I managed to test different parts of the application separately, without taking the bus, and as a result, departure times were

recorded the first time I tested the application in real life, while taking the bus. In this section I will explain how I managed to properly test at the computer an application which its purpose is to gather data while moving around.

TESTING LOCATION UPDATES

Location updates are the base of the algorithms I built, therefore they had to work before everything else. I had a big problem while testing location updates, because iOS development can be only done on a Mac computer, in XCode. When testing using the simulator, the real location coordinates are not available (it retrieves virtual coordinates). Because of that, I could also not write automated tests to assure the code works. I could only see the location when I had an iPhone plugged in into a Mac machine. This is how I could see all the logs/requests in the terminal, and check if the correct location is extracted. But, to record movement, the device must move towards a direction for an amount of time. If I'd move the phone around, it would not see any difference between the locations, and even if it would record locations because of movement, it would get confused and mark the speed as undefined (-1). As I do not own a Mac computer, testing was particularly hard, because I could not move in a straight direction in the labs, with the phone plugged in to the computer.

To check the API requests are successful and that the application records movement, my only solution was creating a controller and a view to display the information that is recorded. This would be present in the mobile application UI for me to view while not plugged in to a Mac machine. I created a table that I can refresh while I am walking and see for every location recorded by the computer the nearest bus stop, the distance to the nearest bus stop, the time and the speed. The process of creating the view took a few days, which is quite a long time for testing a small part of the functionality, but it was necessary for building the algorithm. However, it was successful, and it was the only option using the resources I had at the time.

TESTING MOBILE FUNCTIONALITY

Automatic tests can be written for everything else other than the location updates I have mentioned before and asynchronous API calls. Because of that, we need to provide the testing framework fake information that looks like the responses the mobile would get from these sources. This concept is called "stubbing" (Stackexchange.com, 2018).

When you stub, you assume the response of the API call (or the location update) is always correct. This data is coming from sources the developer cannot control, and hopefully they are tested before being released to the market. Data extraction is decoupled into separate functions, so everything else regarding the algorithm can be tested.

In my application, I make a call to the Transport API every time a location update that is relevant comes in. After making the API call, the information gathered is saved into the Location class. Therefore, I assume location updates come in and the API calls give appropriate responses, and I add the information in the class manually. For testing the algorithm that detects bus deprture times works, I am creating different scenariOS of locations, like the locations that would be recorded when the bus arrives to the station

(which require high speed far away from the station, then low speed close to the station). This is a similar approach to the Factory design pattern, adapted to my needs. These scenariOS that consist in appending certain locations are saved into functions, and then can be called in tests appropriately. For example, when checking if the user was in vehicle, we add two scenariOS: “addWalking()” and “addEnteringBus()”. Using this technique, I tested what would happen if parts of the data was present, or if everything was present and the algorithm would gather information successfully. I consider this part was integration testing, as it tested how multiple blocks of the application are working together towards a goal.

I also tested separately the functions built, using unit tests. Most of the controller functions cannot be tested using unit tests, as they connect directly to the UI, which cannot be accessed in unit tests. However, everything related to the algorithms is written in the models, which I managed to test fully. I used the same style of tests, but in this case just checking the response of one function. In all the tests I wrote the response is tested against expected response using assertion (assert true, false, equal etc.).

TESTING BACKEND FUNCTIONALITY

In Ruby on Rails, the application could be tested in more detail, because the MVC pattern is stricter, therefore different parts of the code are more decoupled. Because of that, I found automatic testing even more useful than for mobile. Because useful tests could be written for every function, I decided to use the test-driven development (TDD) approach. When using TDD, tests must be written before the code, and tests must be run before running the program (Wikipedia.org, 2018). This helps the programmer think of all the edge cases he might have missed if he did not do TDD. Also, TDD is widely used in many companies (I used this technique during my past internship) and knowing its benefits might be a plus for employers.

Among with TDD, I also used behavior driven development (BDD), which favors testing the behavior of the system rather than the functionality, so that when, for example, output structure changes, there is no need to change the tests, too (Codeutopia.net, 2017). “Unit Testing gives you the what. Test-Driven Development gives you the when. Behavior Driven-Development gives you the how. Although you can use each individually, you should combine them for best results as they complement each other very nicely” (Codeutopia.net, 2017). In my application, all three types of testing are present, and are used in conjunction when necessary. As the application grew, I found fixing bugs and adding new features more manageable, because instead of testing the change manually, I would just run all the tests even before opening the server, and it would let me know which parts of the application would not work as expected. Sometimes, I would receive errors in other parts of the application I would not have tested manually.

Ruby on Rails comes with a testing framework called Minitest. However, it is not as complete as other alternatives, and it does not support BDD. Most companies that use Ruby on Rails for development use RSpec, and I decided to learn it and use it too, as it is an important requirement for employees. Also, I think it is a powerful testing framework that allows the developer to write human readable tests, that could be understood by employees not specialized in computer science. It works by creating nested grouping of tests, and each category has a human readable description. When running a test, the computer will run all the commands from each nested level until the test itself. It will also combine the phrases coming from each level. For example, when the first level is described by “get location by id” and the second by

"when the id does not exist", a test description located inside would be "Describe get location by id when the id does not exist it is expected to raise error RecordNotFound". A person doing acceptance testing could easily read this style of tests and understand in what state the testing is. Also, RSpec provides a tool to generate the sentences coming from these tests so they can be forwarded to other people that don't have access to the codebase.

For making tests shorter and easier to write, I used the factory design pattern, with the "FactoryBot" library (gem). This allowed me to easily create factories of my models ("Location" and "BusStop") by only providing a sample value for each attribute. Then, whenever I need to create a location, I can use FactoryBot to create a new location using the attributes I provided, which saves time and makes the code look cleaner:

```
FactoryBot.create :location
```

Tests are also separated in different files. Each model and each controller have their own designated test files. This is a convention created by RSpec, and it creates a structure that every developer knows to follow, which means others will be able to maintain my code easily.

The list of the tests I have written in my back-end application can be found in the appendix section 6, generated from test descriptions using RSpec's tool.

MANUAL TESTING

Manually testing the algorithm did not bring any issues, as the tests I wrote using the techniques I described above were sufficient to cover all the edge cases. When I took the bus for the first time while having the application on, I managed to log correct location updates, calculate how close the phone was to the bus station, sending the departure times to the back-end, and calculate the delays.

After I checked everything was recorded correctly in the database, I searched for the timetable for a bus stop I stopped at, and I retrieved the correct information. The correct bus line was detected, too (I took bus line 8, and the correct line and direction were detected).

When looking at the data recorded, I noticed not all the bus stops were detected on the way, therefore I changed the accuracy of the algorithm, because I was blocking too many location updates. I don't think the algorithm will always perfectly detect information for every bus stop the user goes through, but everything sent is accurate. The accuracy could be maximized even further, but as a result more API calls would be made (which cost money) and more location updates will be processed in the background (which drain the user's phone battery). Therefore, I am trying to find a balance between receiving relevant information and not affecting the user.

I have also shown the interface on my phone to some friends. They gave their opinion purely on the interface, as I did not have resources (money) to let them user test on their phones. Their opinion was positive, saying that it "looked good" and it was "easy to use".

ACCEPTANCE TESTING

In this section I will see how many of the initial requirements I managed to complete. I will include both functional and non-functional requirements.

Functional Requirements

Crowdsourced data – PASS

- Devices with the application installed feed data to the server.
- The application records actual departure times
- Plus: Data is recorded even when the application is not open

Anonymity - PASS

- No user information is sent when location updates are sent. There is no log in, and no personal information is asked when opening the application.

Server Availability - PASS

- Database is always online, accessible on an IP address.
- Can be accessed online through: https://gentle-falls-45144.herokuapp.com/

Delay Estimation – PASS

- Delay is generated
- Delay is estimated using the generated delays; if there is no relevant delay recorded, the delay will be described as "Unknown" on the interface.

Nearby bus stop retrieval by current location - PASS

- Closest bus stops to current location are retrieved on the interface using Transport API

Nearby bus stop retrieval by postcode - PASS

- The user can search the closest bus stops to a postcode they input

Bus station timetable - PASS

- Timetable starting at the requested date is retrieved, and contains bus lines, departure times and estimated delays.

Non-Functional Requirements

Usability - PASS

- User is not aware of the crowdsourced functionality
- User needs to navigate through 2 pages before retrieving the information requested.

Reliability - PASS

- Timetable information is displayed even if there is not relevant information about it in the database (historical data).

Performance - FAIL

- User does not always retrieve information in less than a second. If the server is awake, it passes, but because I am using a free tier, the server is usually on "idle", and it takes around 10 seconds to retrieve it in that case. If a paid service would be used, this requirement would pass.

FUTURE WORK

The application I created is just at the prototype stage now. Three months were not enough to refactor in all the areas I would have liked. In this section I am describing a few areas which I would have liked to improve.

SECURITY

Currently, a big security issue is that anyone can access the endpoints in the back-end. Therefore, if they had the link and the parameters needed, they could create a POST request to add new locations/bus stops.

While using my application, these post requests are only made automatically, when the phone reaches a number of locations and he processes them. The user cannot click anywhere that would result into making a POST request, so only a phone could send them. Therefore, a good solution would be to research how to identify that a request is made from an iPhone, using some variables that are not accessible to the iPhone user. For example, the UDID is a unique identifier for iPhones, but if the user can access it, he might be able to create POST requests manually using their iPhone UDID.

ARCHITECTURE

The database the application is working on is mirroring the APIs I am using. However, these APIs have a structure that is based on NoSQL architecture. The fact that now every bus stop has only one location means that there will be repeating bus stops. To avoid that, the distance to the closest bus stop should exist in the "locations" table, and this would allow one bus stop to have many locations. The code that saves these entries in the database should be rewritten so that a new bus stop is added only if it does not exist, and the location is connected appropriately to a unique bus stop.

Also, if the application becomes widely used, there is the chance that multiple people taking the bus have the application installed, and therefore recording data. I already created a unique identifier for locations and bus stops, which means a new item will not be created if a very similar entry already exists. However, this might not work if the bus is waiting in station for a long time, so I would make more testing and research to make sure duplicates are not being saved.

OPTIMIZATION

Currently, API calls are made every time the user wants to see a timetable. However, if the timetable does not change often, it would be useful to store/cache data that was queried before or is queried often. When a timetable is retrieved on the interface, the internal back-end API is called, which calls the external Transport API and calculates the delays using the internal database. Even though it's not that slow (queries are very quick in Postgres), calling an external API when not necessary wastes time. Also, there is a limited

amount of free Transport API calls daily, so when the application would be widely used, this would also save money.

The easiest improvement to be made is to cache all the information of the timetables saved by the user. It requires creating a separate "timetable" page in the UI, which instead of querying data from the API, it will query it from phone storage. Most of the time, people only need to check timetables for a few routes (for example, bus to/from home), which they would save for easy access. If a person checks the same timetables daily, caching the saved ones will drastically reduce the amount of requests made for searching timetables. Having information saved locally also results in instant display on the interface (no loading times). However, we need to make sure that timetables won't change, because if they do, we risk giving the user incorrect information.

Caching could be also done in the back-end, but if we want to save a lot of information to reduce the number of API calls, a lot of storage space is needed. Also, the same issue where timetables might change remains, so more research needs to be done on this subject.

INTERFACE DESIGN

There are some user interface features I would have liked to add if I had more time. One would have been onboarding slides. Apple recommends having them, too. I would have liked to create 3-4 slides that explain what the application can do, which the user only sees when they first open the application. This can be useful for new iOS users that are not used to the iOS guidelines because they did not use many iOS applications previously. Also, clearly explaining how the "saved" feature works and the meaning behind the half/filled circles on the timetable place would give a better experience from the beginning, as the user does not have to figure out anything.

I would also create a loading view for the "Timetable" page. Now, when a timetable is being searched for, the user sees the timetable page empty before the timetable loads. It would be better for the user experience to have a page containing a loading animated image while the data is gathered. Unfortunately, small things that should come out of the box are not easily implemented, therefore I did not have time to add this feature.

Another issue is that I am using an alert to explain what the half and fully filled circles are on the "Timetable" page. Having a full new page for two sentences was not appropriate, so what I think a better solution could be is having a modal that pops on the bottom side of the page, that also acts as an overlay, just like an alert. This modal should be closed by sliding it down (a gesture). This modal style is less harsh than an alert, and it was chosen by our designer for a similar purpose for the application I worked on during my internship. However, I know creating this modal takes a lot of effort, and I did not consider it a priority.

TESTING

Even though unit and integration tests were covered, testing should be done in other areas before an application goes into production. The last stage of automated tests to consider are UI tests. However, they usually take a lot of time, because while running them the computer must access the interface itself. Because of that, one test should include multiple scenariOS that have to be thought out carefully. UI tests

can be written in Swift, and a framework is created for easily recreating the actions in code. But they usually take longer to write because they must be thought out well, so that tests will still pass if minimal UI changes are made.

The final type of testing which gives insight on how the application will behave on the market is user testing. I was not able to perform user testing because of some limitations: free Apple developer account and free tier on Transport API. Because I am using a free Apple developer account, the application can only be installed on the phone using XCode, through cable connection between the Mac and the iPhone. Moreover, the certificates which are signed when the application is installed on the phone expire in 6 days. If the user wants to have the application running for more, they must plug in the iPhone to my computer (with XCode open) again, to renew the certificates. This is not feasible. If the user cannot have the application constantly running in the background, not enough location data can be saved.

Even if some volunteers would be ok with coming in and renewing their certificates every few days, there would be too many queries to the Transport API, so the usage will be above the threshold in under a day's time. When I took the bus for a day for testing purposes, I managed to reach the threshold alone.

CONCLUSIONS

In conclusion, the application is a working prototype, but it could be perfected in some places, like those I mentioned in the "Further work" section. In the initial plan, I prioritized some tasks, but also had some "optional" ones, which are not vital for the application to run. I managed to complete all the mandatory tasks, even though I had limited resources. The application crowdsources actual departure times, sends them through the database, processes delays, and the user can request to view timetables along with expected delays using the interface.

I also completed optional tasks, such as the feature of saving timetables and the possibility of recording and processing locations in the background. Searching by postcode was also an optional task. Also, writing automated tests that cover most of the codebase was not mentioned in the initial plan, but they are a big plus from a software development perspective. This application could be easily improved and maintained by other developers, because they could develop new features without having the fear of breaking legacy parts of the application they don't know about.

Moreover, I made sure to follow conventions and use well known design patterns. I tried to write code that is readable, using techniques that software development teams use so that team members understand each other's' code. For example, I tried to name variables and functions explicitly, so that others could understand the code even without reading comments.

Given the results, I think I managed to complete this project successfully, because I achieved all the aims, and added more features on the side. I have managed to create two applications and was responsible of the whole stack (from the server to the user interface), and not only I created working algorithms, but also a user interface that is intuitive and easy to use.

REFLECTION ON LEARNING

During this project, I faced many challenges and learned how to solve new problems. Even though I had previous experience with both web and mobile applications, I did not have experience with making them work together. I did not create APIs or request them from mobile. I also did not work with variables retrieved using phone hardware - GPS and location updates.

Working with location updates was hard, because I could only test them while the phone was moving. Finding workarounds to test the app outside the labs was a new experience, which made me think out of the box. I am sure this experience would help me in a similar situation thorough my career.

Since I only worked with GET requests with external APIs, it took me a while to understand that the mobile phone should post the information through the API and not directly to the database. It was quite different from posting through a form because the information comes from a file (JSON). I learned how to manually test POST requests using Postman, how to give written errors as response, throw an error status and how to write custom validations to only accept correct input from the back-end. These are all valuable skills that a back-end developer should have, and I am glad I had the chance to solve these problems alone.

However, RSpec, the BDD testing framework, had the longest learning curve for me. It is very different from unit testing because it tests the behavior rather than the functionality. Also, RSpec is a framework itself, even if it exists within another framework (Ruby on Rails), so it comes with its own conventions (domain specific conventions). Since I wanted to write valuable tests, they required more thinking and more work to achieve the code coverage I wanted. In total, it took me more time to write the back-end tests than to write the code. Even though I struggled, I saw the benefits at the end of the project. Because I trusted the tests I've written were good, I was generally running all the tests before running the server. Sometimes I'd find out I affected other parts of the application with my change, and if I didn't have the tests to show me that, I would have had more bugs. I believe using RSpec increased my code quality.

Using Heroku was also a new experience for me. I have never done operations work by myself before. However, it was easier than I expected, and it overcame my fear of DevOps[8]. I learned how to deploy on the Cloud and to manage the server and the database.

Overall, even though I read before about many of the technologies I used, it was the first time I created a mobile application with a server, from top to bottom. Connecting all the parts was challenging, but it was fun.

As this project is open source, I wanted to be proud of the code I wrote. I tried my best to write good code, using current standards, conventions and design patterns, and using widely used techniques such as TDD and BDD. This project is now a representation of how I would work for a real client, and I am not afraid to show it to employers or future colleagues. I have a lot more to learn, but I have done my best to show that I am trying to be up to date and be part of the software developer community.

[8] Development and Operations

REFERENCES

Apple.com, 2018. *Apple.com.* [Online]
Available at: https://developer.apple.com/documentation/corelocation/cllocationmanager
[Accessed 22 April 2018].

Apple.com, 2018. *Human Interface Guidelines.* [Online]
Available at: https://developer.apple.com/ios/human-interface-guidelines
[Accessed 26 April 2018].

Apple, 2018. *Apple Store.* [Online]
Available at: https://itunes.apple.com/gb/app/cardiff-bus/id670910962?mt=8
[Accessed 17 April 2018].

bbc.co.uk, 2018. *Cash for real-time information on 'vital' bus services.* [Online]
Available at: http://www.bbc.co.uk/news/uk-wales-politics-43978374
[Accessed 6 May 2018].

Bridging-the-gap.com, 2018. *Bridging-the-gap.com.* [Online]
Available at: http://www.bridging-the-gap.com/what-is-a-use-case/
[Accessed 21 April 2018].

Change.org, 2018. *Change.org.* [Online]
Available at: https://www.change.org/p/cardiff-bus-petition-to-call-on-cardiff-bus-to-improve-the-28a-bus-service
[Accessed 17 April 2018].

Codeutopia.net, 2017. *What's the difference between Unit Testing, TDD and BDD?.* [Online]
Available at: https://codeutopia.net/blog/2015/03/01/unit-testing-tdd-and-bdd/
[Accessed 29 April 2018].

Gazarov, P., 2016. *What is an API? In English, please..* [Online]
Available at: https://medium.freecodecamp.org/what-is-an-api-in-english-please-b880a3214a82
[Accessed 18 April 2018].

Heroku.com, 2018. *Heroku.com.* [Online]
Available at: https://www.heroku.com/
[Accessed 20 April 2018].

Json.org, 2018. *Json.org.* [Online]
Available at: https://www.json.org/
[Accessed 19 April 2018].

Kaufman, J., 2016. *Principles of Mobile App Design,* s.l.: Apteligent.

Matthew, N., 2005. Beginning databases with PostgreSQL. In: *Beginning databases with PostgreSQL.* s.l.:Apress.

Postcodes.io, 2018. *Postcodes.io.* [Online]
Available at: http://postcodes.io/
[Accessed 20 April 2018].

Postgresql.org, 2018. *Postgresql.org.* [Online]
Available at: https://www.postgresql.org/
[Accessed 20 April 2018].

Quora.com, 2018. *Quora.com.* [Online]
Available at: https://www.quora.com/What-does-it-mean-that-Ruby-on-Rails-favors-convention-over-configuration
[Accessed 19 April 2018].

Ronga, P., 2018. *Tdg.ch.* [Online]
Available at: https://m.tdg.ch/articles/58a413d5ab5c37152c000001
[Accessed 17 April 2018].

Samsung, 2018. *Samsung.* [Online]
Available at: http://www.samsung.com/uk/smartphones/galaxy-s6-g920f/SM-G920FZKABTU/
[Accessed 18 April 2018].

Sinicki, A., 2016. *Android Authority.* [Online]
Available at: https://www.androidauthority.com/developing-for-android-vs-ios-697304/
[Accessed 18 April 2018].

Stackexchange.com, 2018. *Stackexchange.com.* [Online]
Available at: https://softwareengineering.stackexchange.com/questions/271720/what-does-stubbing-mean-in-programming
[Accessed 28 April 2018].

Trackimo, 2018. *Trackimo.* [Online]
Available at: https://store.trackimo.com/
[Accessed 17 April 2018].

Transport For London, 2018. *Transport for London.* [Online]
Available at: https://tfl.gov.uk/corporate/publications-and-reports/buses-performance-data
[Accessed 17 April 2018].

Transportapi.com, 2018. *Transport API.* [Online]
Available at: https://www.transportapi.com/
[Accessed 20 April 2018].

W3.org, 2018. *W3.org.* [Online]
Available at: https://www.w3.org/TR/DOM-Level-2-Core/introduction.html
[Accessed 17 April 2018].

Wikipedia.org, 2018. *Test-driven development.* [Online]
Available at: https://en.wikipedia.org/wiki/Test-driven_development
[Accessed 29 April 2018].

APPENDIX

1. DETERMINE LOCATION

```
func determineCurrentLocation() {
        locationManager = CLLocationManager()
        locationManager.delegate = self as CLLocationManagerDelegate
        locationManager.desiredAccuracy = kCLLocationAccuracyBest
        locationManager.distanceFilter = 5
        locationManager.allowsBackgroundLocationUpdates = true

        locationManager.requestAlwaysAuthorization()

        if CLLocationManager.locationServicesEnabled() {
            locationManager.startUpdatingLocation()
            //locationManager.startUpdatingHeading()
        }
    }

    func locationManager(_ manager: CLLocationManager, didUpdateLoca-
tions locations: [CLLocation]) {
        if locations.count == 0 {
            return
        }
        guard let lastloc = locations.last else { return }

        // create two inits for cleaner initializations - nearestbus,
note, delay not necessary
        let currentLocation = Location.init(lat: Float(lastloc.coordi-
nate.latitude),    long:    Float(lastloc.coordinate.longitude),    cur-
rentSpeed: Float((manager.location?.speed)!))
        savedLocations.addLocationIfSignificant(loc: currentLocation)
        if savedLocations.locations.count > 100 {
            savedLocations.addToDB()
        }
```

```
    }
```

2. FILTER AND REQUIRE NEAREST BUS STOP

```
func addLocationIfSignificant(loc: Location) {
        if conditionsApply(loc) {
            Variables.requestingNearestBusStops = true
            let lat = Float(loc.lat)
            let long =  Float(loc.long)
            // calls function using completion handler in order to add new
location
            BusStops.allBusStops(lat: lat, long: long) { (busStops, error) in
                if let error = error {
                    // got an error in getting the data
                    print(error)
                    return
                }
                guard let busStops = busStops else {
                    print("error getting all: result is nil")
                    return
                }
                if !busStops.stops.isEmpty || self.locations.isEmpty {
                    var mutatedLoc = loc
                    mutatedLoc.nearestBusStop = busStops.stops.first ?? nil
                    self.locations.append(mutatedLoc)
                    print(self.locations)
                    self.saveToDefaults(self.locations)
                }
            }
            Variables.requestingNearestBusStops = false
        }
    }

func conditionsApply(_ loc: Location) -> Bool {
        // thread lock
        guard Variables.requestingNearestBusStops == false else {return
false}
        // we always want to add the first location for future reference
        guard !locations.isEmpty else { return true }
        // the new location has the same coordinates as the one last
recorded
        if locations.last! == loc { return false }
        // conditions based on walking/driving distance
        if (loc.nearestBusStop?.distance ?? -1) > 200 {
```

```
            if (timeFromLastLocation(loc) > 40 && loc.currentSpeed <=
walkingSpeedTreshold) || (timeFromLastLocation(loc) > 20 && loc.cur-
rentSpeed > walkingSpeedTreshold) {
                return true
            }
        }
        else {
            if (timeFromLastLocation(loc) > 20 && loc.currentSpeed <=
walkingSpeedTreshold) || (timeFromLastLocation(loc) > 10 && loc.cur-
rentSpeed > walkingSpeedTreshold) {
                return true
            }
        }
        return false
    }
```

3. FIND BUS DEPARTURE TIMES

```
func findBusStopTimes() -> [Location] {
        var busStopTimes:[Location] = []
        let locs = self.hasLeftBusAt()
        for location in locs {
            // add to persistent memory
            switch location.note {
            case .leftStation:
                // take current location
                busStopTimes.append(location)
            case .arrivedStation:
                guard let nextLocation = locs.item(after: location) else
{ return busStopTimes }
                if nextLocation.note == .leftStation {
                    // do nothing
                    break
                }
                else {
                    // take current location
                    busStopTimes.append(location)
                }
            case .none: break // do nothing
            }
        }
        return busStopTimes
    }

    func hasLeftBusAt() -> [Location] {
```

```
        var inStationAtLocations:[Location] = []
        var lastStationIndex:Int = 0
        for var loc in locations {
            print("index = " + String(locations.index(of: loc)!) )
            guard var nextLoc = locations.item(after: loc) else {return
inStationAtLocations}
            let inVehicle = isInVehicle(index: locations.index(of: loc)!,
since: lastStationIndex)
            // after leaving the station, busses usually sprint there-
fore checking if the next location has driving speed
            if loc.isWaitingInStation() && nextLoc.isInVehicle() {
                loc.note = .leftStation
                lastStationIndex = locations.index(of: loc)!
                inStationAtLocations.append(loc)
            }
            // when arriving in station checks against all previous lo-
cations
            if inVehicle && nextLoc.isWaitingInStation() {
                nextLoc.note = .arrivedStation
                lastStationIndex = locations.index(of: loc)!
                inStationAtLocations.append(nextLoc)
            }
        }
        return inStationAtLocations
    }

    func emptyLocations() {
        if self.locations.isEmpty {
            self.locations = [self.locations.last!]
        }
    }
```

4. DELAY DETECTION

```
def get_delay_for_bus
    return if @schedule.empty?

    delay = Float::INFINITY
    bus_line = ""
    sch_time = ""
    dep_time = ""

    @schedule.each do |departure|
```

```
      sch_time = departure["date"] + " " + departure["aimed_depar-
ture_time"]

      # time difference has to be smaller than the minimum delay and
more than 5 minutes early
      # if it's more than 5 minutes early it's probably the previous bus
being late
      if time_difference(sch_time).abs < delay.abs && time_differ-
ence(sch_time) >= -5
        dep_time = sch_time
        delay = time_difference(sch_time).to_i
        # line_name or line??
        bus_line = departure["line_name"]
      end
    end

    {delay: delay, bus_line: bus_line, aimed_departure_time: dep_time}
  end
```

5. CALCULATE EXPECTED DELAY

```
def add_delays_to_schedule
    return if @schedule.empty?

    @schedule.each do |departure|
      delay_info = find_delay_for_departure(departure)

      unless delay_info.nil?
        departure["delay"] = delay_info[:delay].to_i
        departure["record_count"] = delay_info[:record_count]
        expected_dep = expected_from_aimed_departure(departure["date"],
departure["aimed_departure_time"], delay_info[:delay])
        departure["expected_departure_time"] = expected_dep[:time]
        departure["expected_departure_date"] = expected_dep[:date]
      else
        departure["delay"] = 0
        departure["record_count"] = 0
        departure["expected_departure_time"] = "unknown"
        departure["expected_departure_date"] = "unknown"
      end
    end
    @schedule
  end

  def find_delay_for_departure(departure)
```

```
    dep_time = departure["date"] + " " + departure["aimed_depar-
ture_time"]
    is_weekday = is_weekday(dep_time)
    locations = Location.where(bus_line: departure["line_name"])
    locations = locations.where_weekday(is_weekday)

    if locations.where_aimed_time(departure["aimed_depar-
ture_time"]).empty?
      starting = (departure["aimed_departure_time"].to_time -
1.hour).strftime("%H:%M")
      ending = (departure["aimed_departure_time"].to_time +
1.hour).strftime("%H:%M")

      locations = locations.where_aimed_in_time_range(starting, ending)
    else
      locations = locations.where_aimed_time(departure["aimed_depar-
ture_time"])
    end

    if locations.empty?
      nil
    else
      {delay: locations.average_delay, record_count: locations.count}
    end
  end

def expected_from_aimed_departure(aim_date, aim_time, delay)
    aimed_time = (aim_date + " " + aim_time).to_time
    expected_time = aimed_time + delay.minutes
    {date: expected_time.strftime("%Y-%m-%d"),
     time: expected_time.strftime("%H:%M")}
  end
```

6. RSPEC TESTS

```
BusStop
  Validations
    when valid
      should change `BusStop.count` by 1
    when invalid
      should raise ActiveRecord::RecordInvalid and change
`BusStop.count` by 0
    #atcocode
      when not present
        should not be valid
```

```
      when duplicate
        should not be valid
    #mode
      when not == bus
        should not be valid
      when == bus
        should be valid
    #name
      when not present
        should not be valid
      when less than 3 characters
        should not be valid
    #stop_name
      when not present
        should not be valid
      when less than 3 characters
        should not be valid
    #bearing
      when > 2 characters
        should not be valid
      when length == 1
        should be valid
      when includes character A..Z
        should not be valid
      when not capitalised
        should eq "SW"
    #longitude
      when not present
        should not be valid
      when not float
        should not be valid
    #latitude
      when not present
        should not be valid
      when not float
        should not be valid
    #distance
      when not present
        should not be valid
      when not integer
        should not be valid

Location
  Validations
    when invalid
```

```
      should raise ActiveRecord::RecordInvalid and change `Location.count` by 0
    #latitude
      when not float
        should not be valid
      when duplicate
        should not be valid
    #longitude
      when not float
        should not be valid
    #current_speed
      when not float
        should not be valid
    #time
      when not datetime
        should not be valid
      when later than current time
        should not be valid
  class function
    is_weekday
      when friday
        should equal true
      when saturday
        should equal false
    expected_time
      when delay positive
        should eq "2018-02-24 14:22:17"
      when delay negative
        should eq "2018-02-24 14:30:17"
  scope
    where_aimed_time
      when time = aimed time
        should have attributes {"id" => 252, "latitude" => 2.0, "longitude" => 1.5, "time" => 2018-02-27 14:25:17.000000000 +0000, "current_...0 +0000, "delay" => 5.0, "bus_line" => "9", "aimed_departure_time" => 2018-02-27 14:30:17.000000000 +0000}
      when time != aimed time
        should be blank
    aimed_in_time_range
      when location in time range
        should have attributes {"id" => 254, "latitude" => 2.0, "longitude" => 1.5, "time" => 2018-02-27 14:25:17.000000000 +0000, "current_...0 +0000, "delay" => 5.0, "bus_line" => "9", "aimed_departure_time" => 2018-02-27 14:30:17.000000000 +0000}
      when location not in time range
```

```
        should be blank
    where_weekday
      when choosing weekdays
        when valid = true
          should have attributes {"id" => 256, "latitude" => 2.0, "lon-
gitude" => 1.5, "time" => 2018-02-27 14:25:17.000000000 +0000, "cur-
rent_...0 +0000, "delay" => 5.0, "bus_line" => "9", "aimed_depar-
ture_time" => 2018-02-23 14:25:17.000000000 +0000}
        when valid = false
          should be blank
      when choosing weekends
        when valid == true
          should be blank
        when valid == false
          should have attributes {"id" => 259, "latitude" => 2.0, "lon-
gitude" => 1.5, "time" => 2018-02-24 14:25:17.000000000 +0000, "cur-
rent_...0 +0000, "delay" => 5.0, "bus_line" => "9", "aimed_depar-
ture_time" => 2018-02-24 14:25:17.000000000 +0000}
    avergae_delay
      when empty
        should be blank
      when two elements
        should eq 3.5

Timetable
  class function
    refresh_schedule
      when correct atcocode and time
        should eq []
    get_delay_for_bus
      when schedule empty
        should be nil
      when schedule has one element
        should   eq   {:delay=>-4,   :bus_line=>"9",   :aimed_depar-
ture_time=>"2018-02-27 14:54"}
      when closest departure is before
        should   eq   {:delay=>-4,   :bus_line=>"9",   :aimed_depar-
ture_time=>"2018-02-27 14:54"}
      when closest departure is after
        should   eq   {:delay=>2,   :bus_line=>"8",   :aimed_depar-
ture_time=>"2018-02-27 14:48"}
    find_delay_for_departure
      when there's no relevant location information
        should be nil
      when there is a location aimed to leave at the same time
```

```
        should eq {:delay=>5, :record_count=>1}
      when there is a location aimed to leave within +- 1 hour
        should eq {:delay=>5, :record_count=>1}
      when there is a location aimed to leave within +- 1 hour
        should eq {:delay=>5, :record_count=>1}
    add_delays_to_schedule
      when nothing in schedule
        should be blank
      when delay not found
        should include (a hash including {"expected_departure_time" =>
"unknown", "expected_departure_date" => "unknown"})
      when one item in schedule
        should include (a hash including {"expected_departure_time" =>
"14:30", "expected_departure_date" => "2018-02-27"})
      when multiple in schedule
        should include (a hash including {"expected_departure_time" =>
"14:30", "expected_departure_date" => "2018-02-27"})

BusStopController
  get list of bus stops
    when empty
      should be empty
    when not empty
      should include {"id" => 471, "atcocode" => "5710AWA10617", "mode"
=> "bus", "name" => "Treharris Street", "stop_name" => "Treharris Street",
"bearing" => "SE", "smscode" => "cdipaga", "locality" => "Roath", "in-
dicator" => "o/s", "longitude" => -3.16913, "latitude" => 51.48983,
"distance" => 61, "created_at" => "2018-05-07T15:33:40.343Z", "up-
dated_at" => "2018-05-07T15:33:40.343Z"}
  get bus stop by id
    when the id does not exist
      should raise ActiveRecord::RecordNotFound
    when the id exists
      should eq {"id"=>473, "atcocode"=>"5710AWA10617", "mode"=>"bus",
"name"=>"Treharris Street", "stop_name"=>"Treh... "distance"=>61, "cre-
ated_at"=>"2018-05-07T15:33:40.359Z",          "updated_at"=>"2018-05-
07T15:33:40.359Z"}
  post bus stop
    when doesn't validate
      should respond with status code :bad_request (400)
    when validations pass
      should return status :ok
      should change `BusStop.count` by 1

Location
```

```
  get list of locations
    when empty
      should be empty
    when not empty
      should include {"id" => 269, "latitude" => 2.0, "longitude" => 1.5,
"time" => "2018-02-27T14:25:17.000Z", "current_speed" => 1.5, "note" =>
1, "bus_stop_id" => 476, "created_at" => "2018-05-07T15:33:40.412Z",
"updated_at" => "2018-05-07T15:33:40.412Z", "delay" => 5.0, "bus_line"
=> "9", "aimed_departure_time" => "2018-02-27T14:30:17.000Z"}
  get location by id
    when the id does not exist
      should raise ActiveRecord::RecordNotFound
    when the id exists
      should   eq   {"id"=>271,   "latitude"=>2.0,   "longitude"=>1.5,
"time"=>"2018-02-27T14:25:17.000Z",                                "cur-
rent_speed"=>1...7T15:33:40.432Z",    "delay"=>5.0,    "bus_line"=>"9",
"aimed_departure_time"=>"2018-02-27T14:30:17.000Z"}
  post location
    when the bus stop does not exist
      should respond with status code :not_found (404)
    when the bus stop exists
      should return status :ok
      should change `Location.count` by 1

Timetable
  get timetable (list of departures)
    when date not valid
      should respond with status code :bad_request (400)
    when time not valid
      should respond with status code :bad_request (400)
    when not empty
      should include "all"
      should include (a hash including {"delay" => 5})
  get timetable (list of departures) for a bus line
    when date not valid
      should respond with status code :bad_request (400)
    when time not valid
      should respond with status code :bad_request (400)
    when not empty
      should include "all"
      should include (a hash including {"delay" => 5})

Finished in 0.8978 seconds
(files took 2.04 seconds to load)
76 examples, 0 failures
```

7. SCHEMA

```
ActiveRecord::Schema.define(version: 20180416115858) do

  # These are extensions that must be enabled in order to support this
database
  enable_extension "plpgsql"

  create_table "bus_stops", force: :cascade do |t|
    t.string "atcocode"
    t.string "mode"
    t.string "name"
    t.string "stop_name"
    t.string "bearing"
    t.string "smscode"
    t.string "locality"
    t.string "indicator"
    t.float "longitude"
    t.float "latitude"
    t.integer "distance"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

  create_table "locations", force: :cascade do |t|
    t.float "latitude"
    t.float "longitude"
    t.datetime "time"
    t.float "current_speed"
    t.integer "note"
    t.bigint "bus_stop_id"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.float "delay"
    t.string "bus_line"
    t.datetime "aimed_departure_time"
    t.index ["bus_stop_id"], name: "index_locations_on_bus_stop_id"
  end

  add_foreign_key "locations", "bus_stops", on_delete: :cascade
end
```