OCR Computer Science

Programming Project

Ride Sharing App

Bradley Richard Cable

Table Of Contents

Analysis

Identifying A Problem

As you start reaching the age of 16/17 learning how to drive is close to the forefront of your mind. Some people are just starting to learn; some already have their full license. The latter may start giving out lifts to friends or family. Sometimes at a cost.

Especially on social media's such as Snapchat you see people posting that they are "giving lifts". There are benefits both to the person driving as they are getting very valuable experience behind the wheel and the passenger is getting to where they want to go for a considerably cheaper price than if they took alternative such as a Taxi or other forms of public transport. As well as this public transport in underdeveloped areas such as villages can have poor infrastructure compared to cities meaning they are unable to provide a comprehensive service. Therefore you are limited to where you can go and when. This poor infrastructure is due to cuts in funding. In England, alone funding for buses is "£400 million a year less each year … than to a decade ago"[4]. "The people who are worst affected [by these cuts] are those who are most reliant: those on lower incomes, the elderly, students"[4]. Furthermore, multibillion-dollar ride-hailing services such as Uber are only available in big cities such as London meaning those who need a service like that the most do not have access to such.

In the case of my application, it would never encounter the problems stated. Starting with a lack of infrastructure. This is

because infrastructure isn't needed, excluding phone and an internet connection. The service is provided by the users (drivers) not the app itself. Bus timetables may have gaps or random cancellations and delays[6] when you want to go, again if a user is online and willing to take you. No more "hour-long delays" which can be commonplace in local areas[6]. The user also wouldn't need to worry about how much they need to pay as that would be stated before the ride and paid for with one click at the end of it.

As already stated there are already companies such as Uber and Lyft that provide peer-to-peer ride-sharing, the downfalls to such a service are the larger they grow they have implications with the law. For example Uber had disputes with governments with regards to employment legislation and if their drivers are even classed as employees[1]. As well as that Transport for London took away Uber's taxi license due to a fall in demand of the iconic London black cabs[2]. In the case of my app if it was to grow to a very large user base as the purpose of this app is friends and favours not to make a substantial income which is the case with other self-employed drivers. You can see this is the case with the rise of the "gig-economy" in cities. Which is major because as said are they employees? With the current economic climate due to the Coronavirus (COVID-19) people that work in the gig-economy like drivers of Lyft and Uber will have problems due to their lack/ minimal support compared to full-time employees.

So to define my project idea simply it would be a ride-sharing app between friends and friends only which is clean and simple focusing on the idea of 'mates rates'.

Why solvable by a computer

Before explaining why it's solvable by a computer, firstly explain why it's a problem if you don't. If you're trying to find someone to give you a lift. To start you need friends to see if they are available and willing to take you. This could mean calling multiple people, then you run into other problems:

1. What if the friend is too far away?
2. Your friend has had a drink and therefore can't drive?
3. Will they want some money back (petrol money)?
4. What if you've already agreed to have a lift with friend A but it turns out friend B is closer and can pick you up sooner?

The problem in all those cases is mismatch in information between your friend and you. Before you pick up the phone you don't know "*XYZ*", and you probably wouldn't of called if you knew "*XYZ*" meant he couldn't do this lift for you.

So why is this problem solvable by a computer? One of the best things about computers is the ability to take in a vast amounts of data, manipulate it for the current user and display it in some kind of digestible formate.

In the case of this problem it can take in data from the user and users friends such as:

1. Where they are, using location services.
2. If they want to be "online" and available to drive.

3. They can set if they want to charge and if so how much.

4. You will be able to see all your friends that are online and which one can get to you fastest.

By collecting all this data and displaying it in a clean and concise format will solve all the problems above as there will be no mismatch in information between the friend and the person that wants a lift.

As stated above as well as collecting this data can solve the problem manipulating it will make the experience better for the user. As the app will know your current location when you are on the app, as well as your friends' location. Therefore it can work how long it would take for your friend to get to you, then to drive where you want to do.

Algorithmic Thinking

Decomposition

Overall my app is a way to get lifts from your friends and make it easier and simpler (inc. some kind of payment if necessary. This can be broken down it to its key areas

- ***"Getting Lifts"*** - this is the main feature and requires a multitude of steps. Firstly displaying all the user's friends (which are online) on a map to see where they are. Secondly is requesting a lift from a friend of their choice. If accepted they can type in where they want to be dropped off. It would also display how much it would cost depending on the friends' price per mile they have set in their settings. If the driver confirms they will be routed to the friend's location to then take them to where they need to go. After the ride is

complete the payment system will kick in to get reimbursement from the rider.

- ***Friends*** - these are users that you can add/remove from your profile. Adding them to your profile will mean you have easy access to them, as well as seeing if they are online. This feature will have sub-components of friend requests, accepting the said request and fetching your friends from the database.

- ***Payment System*** - this is the way you will reimburse your friends for fuel money. The amount for each ride would depend on the friends' rate and how far the journey is as obviously the longer the journey the more fuel. Before making this system I will have to choose a 3rd Party payment system. Examples of these include Stripe, PayPal and Square. These all have ups and downs. These advantages and disadvantages will be discussed in another section.

- **"easier and simpler"** - the overall goal of this app is to make it "easier and simpler" to have lifts with your friend's. All the objectives I've stated above will be the building blocks to this. As well as making the UI and UX design clean and simple to make it as easy as possible to go from opening the app to being in a car.

Pattern Recognition

- Using the data collected by the user and said users friend, the app can use basic logic to make the assumptions for example if one friend is 50 miles away from the user then it would be reasonable to assume that he would not be picking

them up and therefore putting that friend to the bottom of the suggested friends list.

- The app can also make assumptions based off who the user chooses, for example if the user always chooses the lowest price compared to just which friend it is. Then the app can automatically suggest lower price per mile friends first in the list.

Abstraction

- This app will collect a lot of data from its users when it's in use, for example: location data, current rides taking place, ride history, prices, payment data (such as card details). When and what and how to display this data is very important (mention about data protection, what section would this be in).
- This is especially important to abstract data due to the fact this app will mainly be used on the move therefore using mobile data. So to use as little data as possible when uploading and downloading (fetching) data is very important.

- Data I will abstract the most is location, this is because the app will be collecting their exact location, however this doesn't have to be shown to the user, instead I can place the user on a map so the user understands where they are, but not exact coordinates for example.

Algorithm Design Debugging

- I will cover this in the design section of this report

Research The Market?

The current ride share market contains two competitors Lyft and Uber. When it comes to Uber cost is broken down into three sections. The base rate for an UberX ride ‘X’. Then you have the variable parts which is ‘Y’ a minuet and ‘Z’ per mile. XYZ depends on where you are and if there is high demand.

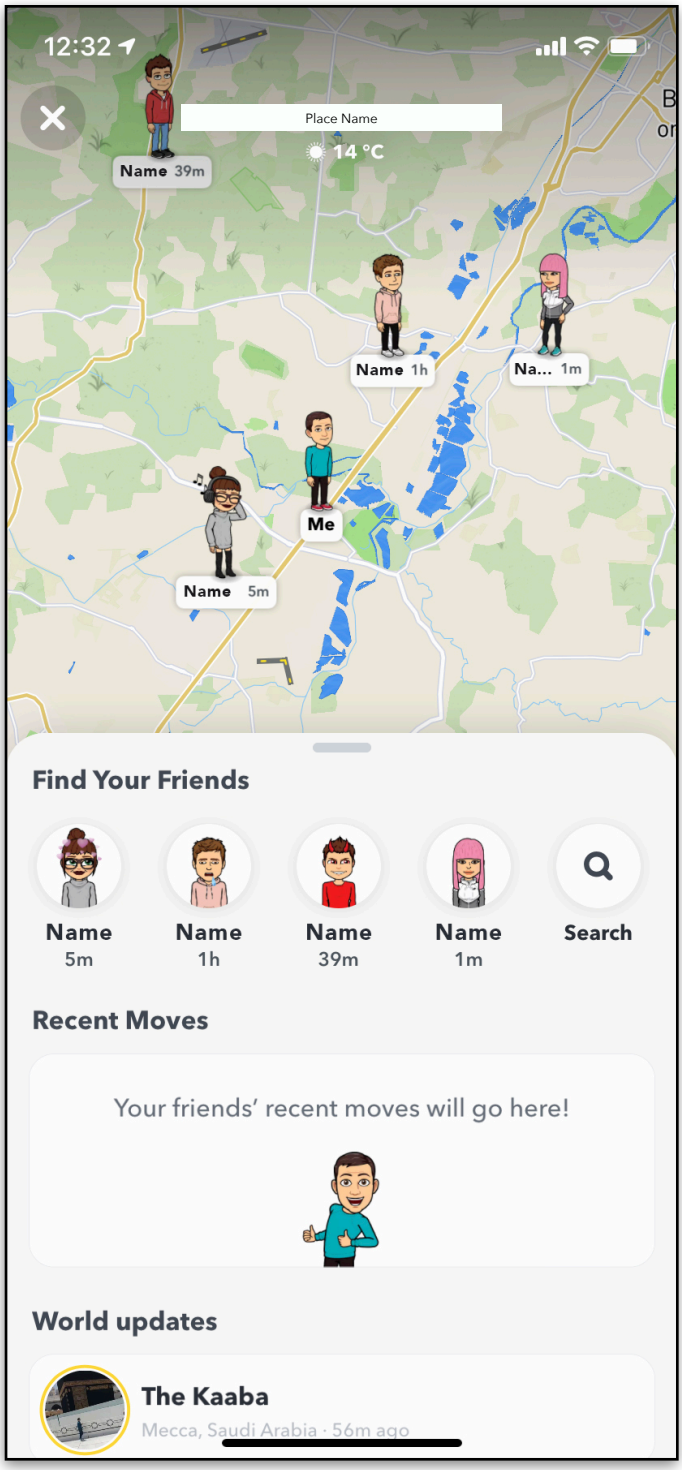
Even with its slightly complicated pricing structure, Uber comes out cheaper on average compared the normal taxi, with it being around 80%[8] cheaper than a normal taxi kilometre for kilometre. However, the UK is the most expensive to haul a (normal) cab[8].

However, there are other forms of transport apart from taxis. As mentioned before in previous sections busses are another viable option. Which from my experience alone cost around £1.80 one way. Which if you compare with your average ‘mates rates’ of 50p / £1.

Research The Market - Application Perspective?

After looking into many apps from ride sharing apps such as Uber and Lyft and other general apps such as Amazon and Snapchat.

As said my app will need some way of displaying where your friends (which are online) are. Snapchat's 'Snap Maps' is a very simple interface for displaying such data.

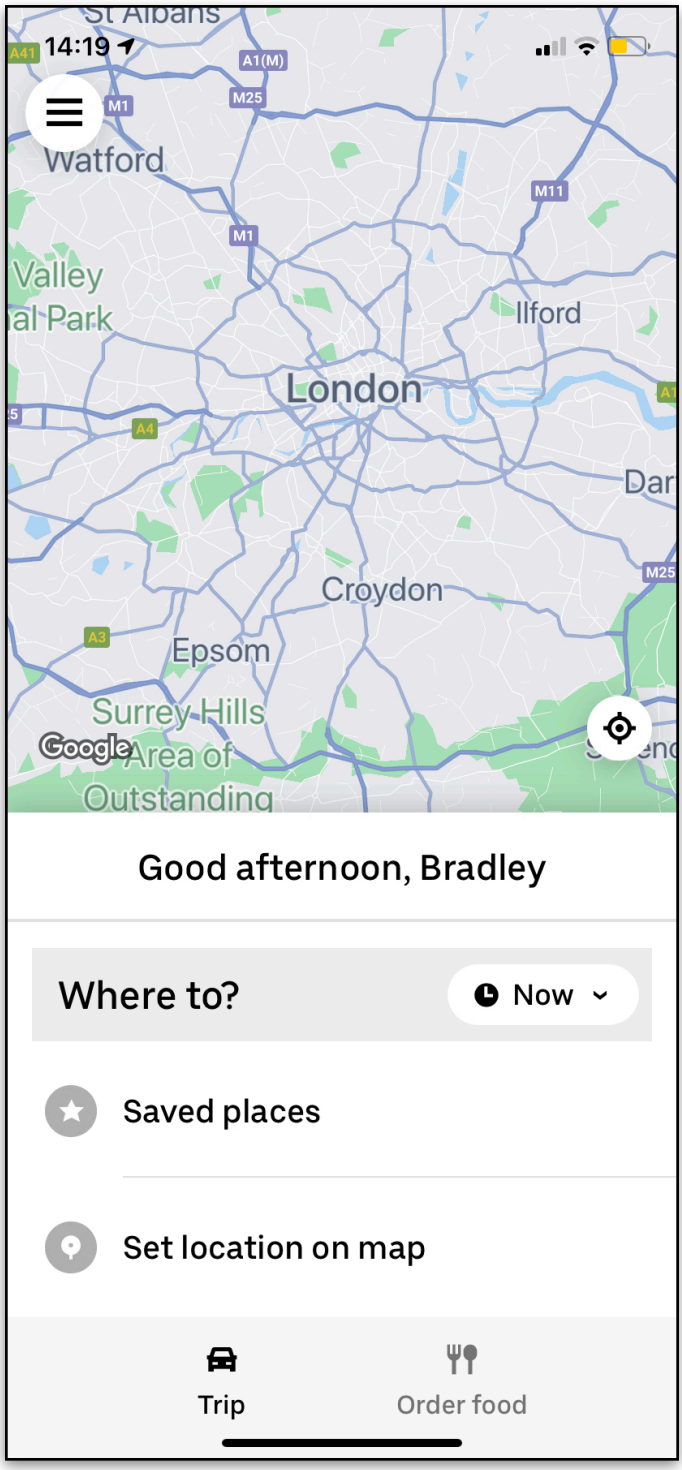


Snap Maps

Very easy interface that gets the information off very quickly. You can immediately see where your friends are compared to you. As well as when they were last online. The map is very clean and abstracts a lot of place names away apart depending on zoom level.

Quick access to your top friends which changes depending on how you talk to people.

This is obviously focusing on the friends aspect.



Uber App

Clearly the focus isn't on friends as Uber is more of a consumer and freelance connector platform.

Allowing users to connect with freelance drivers in their area (Uber drivers) so they can get to their destination.

From the image you can tell that compared to 'Snap Maps' there is less abstraction, showing the user key infrastructure (eg: M25), even when zoomed out to a significant level.

Bellow the map is suggested and saved places you might want to go too. Again comparing to the friend status on 'Snap Maps'.

Combining these two interface styles, detailed maps with curated places. As well as the ability to see if friends are online (available to drive in this case). But making sure the user doesn't get overwhelmed with too much data and a cluttered interface is important.

Existing Solutions To Problems

With this app the new system will be more to do with data is manipulated from both user input and 1st and 3rd part library.

Some examples of library's:

i) Stripe Payment

ii) Sign In With Apple

iii) MapKit & CoreLocation

iv) Mobile Backend (Database)

Maps & Calculating Distance

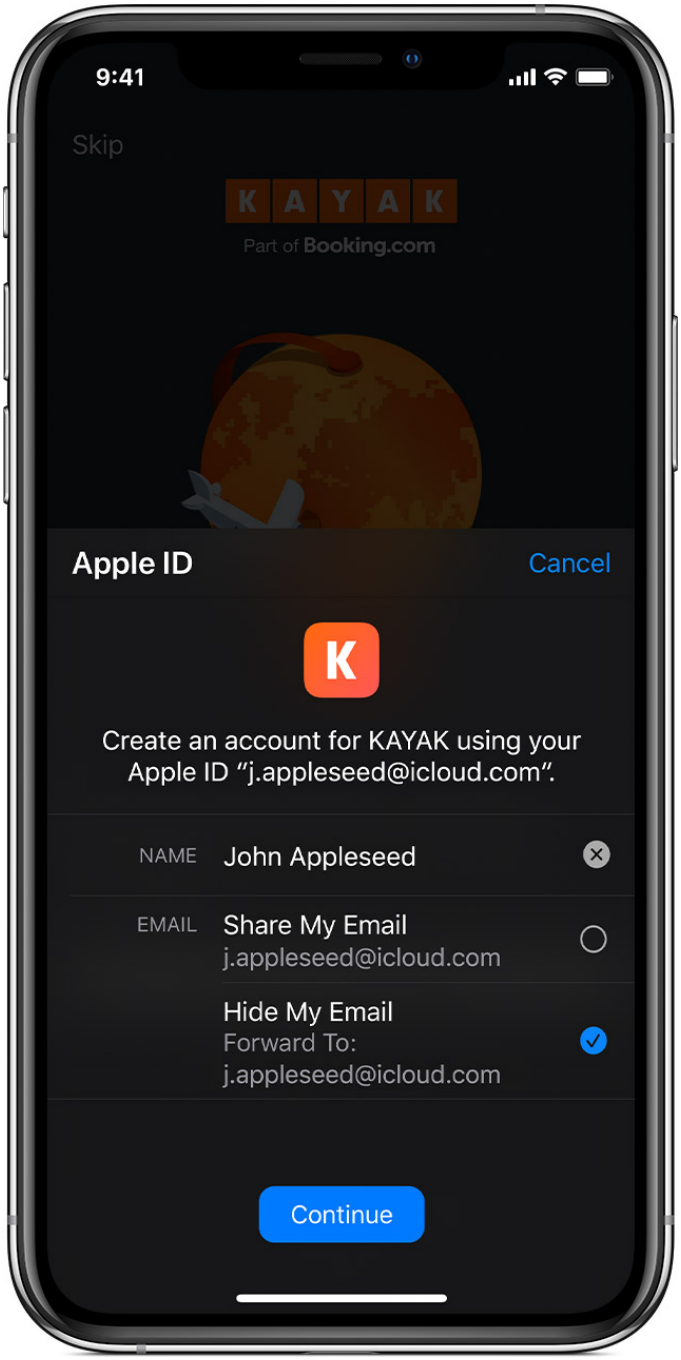
iOS (Swift) comes with 1st party library's that can be used. The reason you can import all these libraries is so you can use the full potential of the hardware as well as making sure the app is as small and streamlined as possible. As you have to import the libraries you, they are not pre-imported.

MapKit will allow me to[9]:

- Display a map to users which I can manipulate.
- Provide text completion to make it easy for users to search for a destination or point of interest.

Using MapKit, showing the users friends and allowing them to find and display the place they want to go (in relation to them) to should be relatively easy. MapKit will also allow me to retrieve driving directions between two points, this can then give the user an estimated ETA as well as the price for the trip (dependent on the friends "price per mile" rate).

However this map will need to be supplied data from both the database (the location of the users friends) and the location of the user itself. This is the purpose for “CoreLocation”.



Sign In With Apple

As this is programmed for iPhones only this means I can make use of a 1st Part O-Auth system, nearly released called “Sign In With Apple”.

A extremely privacy conscious and *simplistic system* allowing users to sign up and login with one click. Authenticated biometrically with ether FaceID or TouchID.

As it is fairly new documentation can be sparse but it means login and sign up can be dealt by Apple and the Database provider of choice (which will obviously have to support such O-Auth system).

No sign up and login form will have to be created on my end. All that will be needed is a clean landing page with said button, and can communicate with the database once login is successful.

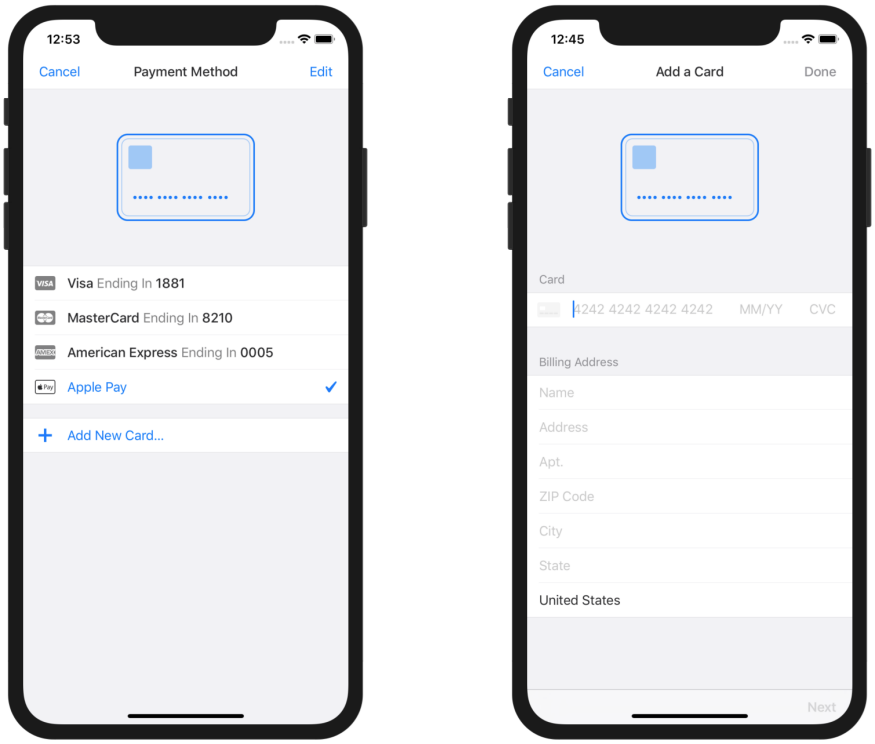
Sign up with Apple

Payment System (Stripe)

Obviously I personally can't make a payment system from the ground up. Therefore I will use a 3rd *payment gateway* system, the most popular of which is Stripe.

Stripe has both a prebuilt (with slight customisation) and build your own API and library[10]. Which one I use will be dependent on time constraints during development.

They also have a the ability to just add an Apple Pay[11] button. So if time is not sparse, payment can be mainly Apple Pay with a *fallback* of normal Stripe Payment (entering card details)[10] if the app detects there are no cards active in Apple Pay Wallet.



Mobile Backend Choice

When looking though my options two services stood out "Parse Server" and "Firebase (Google)". FireBase seems to be mainly focused on their "Realtime Database". Which is a NoSQL database with the ability to update realtime in-between platforms.

The fact that it updates realtime in-between platforms is not important as I am only developing it for iOS (Apple). Furthermore personally the use of NoSQL is not a preference I would choose due to its lack of relationships between data, and the classic table structure of databases.

Whereas Parses database structure is more traditional getting close to database standards such as MySQL with a more relationship focused structure.

Personally, separating data in the logical chucks and then relating that all to a central user. Is more sensible than the "parent and child" tree approach seen in the FireBase "Realtime Database".

Parse also has the upper hand mainly due the fact I've developed with the API before and have a solid foundation with the system. In addition this foundation is also solidified by its very active open source community on GitHub meaning intensive documentation not just by main developers but the community as well.

	Firebase	Parse Server
General Purpose	Fast real time updates (Real time BaaS)	Open source
Hosting	Google hosting. Free up to 100 simultaneous connections .	Self hosting and Parse hosting providers. No limits. Supports Local testing and developing
Custom Code	Custom code not supported	Custom code totally supported(Cloud Code)
Database	Supports model observer scheme. Now introduced Firebase Storage to upload and download files securely	Has huge relationship based databases
Push	Support Push notifications. Firebase Remote Config to customize apps	Support Push notifications for Android, iOS. Also it is possible to send Push Notifications campaigns.
Setup	Easy setup	Quick setup on Parse Easy step by step set up guide available for migrating from Parse to Parse Server
Storage	Stores data as JSON and data backup can be uploaded to Amazon S3 bucket or Google cloud storage	No restricted time limits and No file storage restrictions. Control over backup, restore, database indexes.
Provider	Developed by Google	Developed by Facebook
Ideal for	Suitable for time applications	Suitable for building general purpose applications

(Firebase vs. Parse Server | Back4App Blog, 2020) [12]

At current writing (27/06/20) I will be going ahead with the assumption I will be using Parse. This is more a preferential choice as I already know the system, and prefer its structure.

Who are the stakeholders?

The app is going to be focusing on the age range 15-18 but it is split into two sectors. The driver and the rider.

The driver has just passed their driving test and wants some more hours behind the wheel as well as maybe some extra cash on the side. Cash on the side could be very important to some drivers due to the very high insurance costs for new drivers on any car. The average price of car insurance in the UK can range from £485 a year to £692[5]. But that is the average across the board. For people in their 20s, it can be around £1,035[5], that is over double of the lowest average! Very high costs due to insurance and car prices do not mix well with going out often and minimum wage zero-hour contracts. A few pounds here and there by doing lifts for your mates may help smooth it over.

Riders have the same problems of drivers when it comes to lower incomes, as they will have the same minimum wage zero-hour contract as drivers. This is just due to their age. As a result of this, they will want transport as cheap as possible. Which is possible with this idea as the whole point is it is based around 'mates rates'. As well as all that, there are other factors why a user might be a rider and not a driver, this includes but is not limited too: been drinking, they have not passed their test, taking their car may not be appropriate (eg: airport).

This age range (15-18) is very different compared to other peer-to-peer ride sharers as they have age limits to ride and to be a driver of 18[3] the main reasons for this is so Uber can be on the right side of the law.

Another stakeholder could be university students, due to the fact it is too expensive to keep your car on or around campus. Consequently, their will be very few people with cars, however, people still need to get around. Therefore people that have their cars can be drivers which can facilitate their income as well as helping out friends and classmates.

As well as being useful for people of younger ages and friends, it can also helpful for co-workers if their business introduces a car share scheme. As it would allow them to track whom they're taking and for how many miles. Which could be helpful for two reasons: allowing them to claim mileage on fuel usage with proof of how far they went, as well as ask for some kind of reimbursement from their co-workers.

But as said the main focus is not making money off the app, like such other apps (self-employed). It's meant for friends that need a favour doing, which I think personally there is a big market for.

System and user requirements

The system requirements of this app will be iPhone user running the latest major release of the iOS operating system (iOS 14). Luckily Apple has very high adoption rates of updates compared to competitors. At the time of writing (07/03/2020), the current device version rate according to Source 4 (see biography) is 92.5% is on iOS 13. So In real terms, it isn't a requirement.

If you're a driver you also need a valid driving license of your country of residence (maybe check that in-app somehow?). Riders do not have any other special requirements.

Obviously the app will only be available for iPhone as it will be coded in Swift, Apple's programming language. Therefore an iPhone is a requirement. Again luckily to Apple's image and popularity, the majority of the people in the age range I stated in the stakeholders' section (15-18) have iPhones.

Survey On Ride Sharing

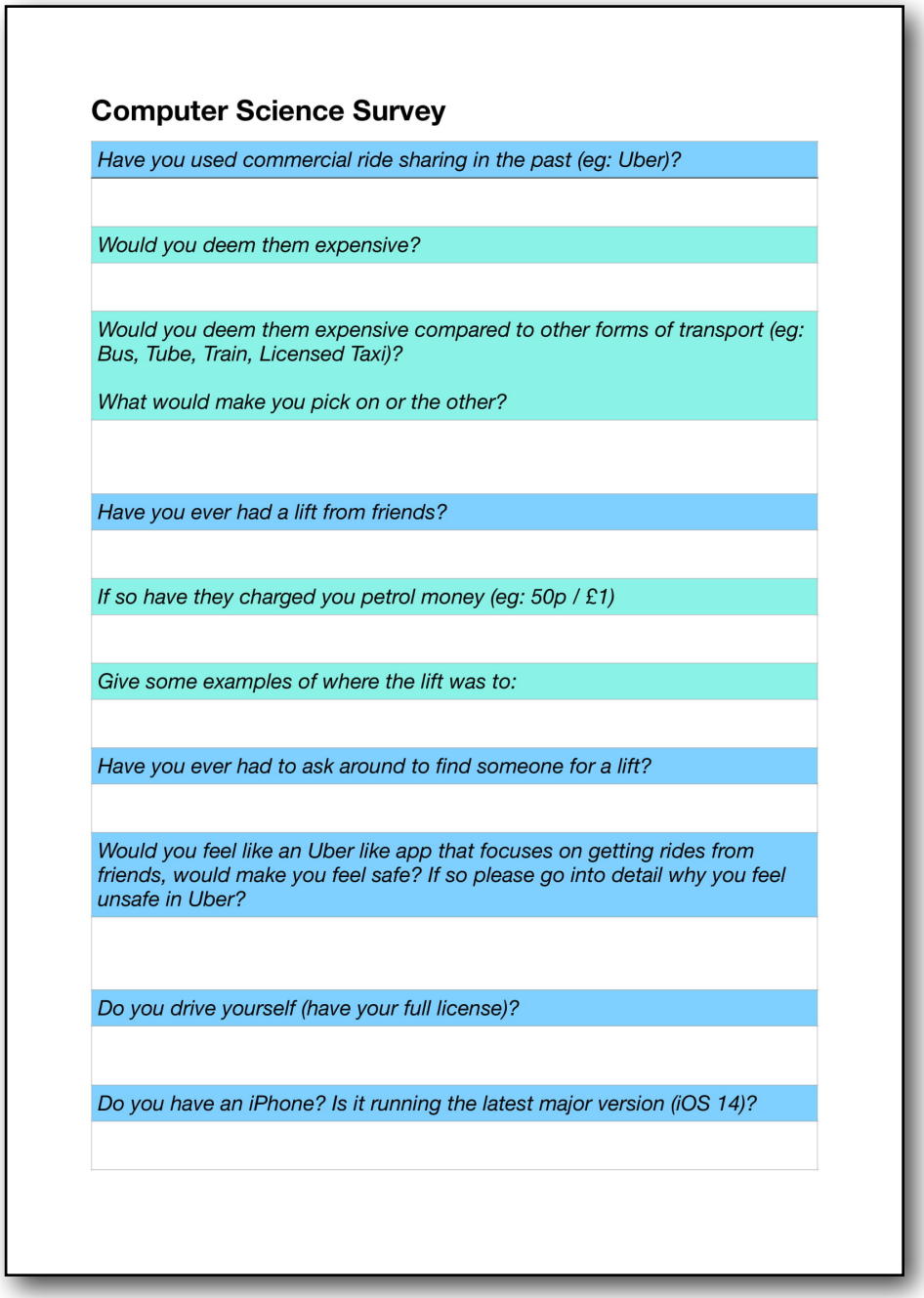
I personally think there is a big market available for a friend based solution but to confirm my thoughts, I will create a survey which will give to a vast majority peers and family which will ask important questions such as:

- Have you used commercial ride sharing in the past (eg: Uber)?
    - Would you deem them expensive?
    - Would you deem them expensive compared to other forms of transport (eg: Bus, Tube, Train, Licensed Taxi)?
- Have you ever had a lift from friends?
    - If so have they charged you petrol money (eg: 50p / £1)
    - Give some examples of where the lift was to:
- Have you ever had to ask around to find someone for a lift?
- Do you drive yourself (have your full license)?

I've decided to do this survey in a paper based format, the main reason for this is because of the limitation of some online survey platforms such as “survey monkey”, not allowing to do logic unless you pay. The paper format will mean it I will be able to get more legitimate responses as I’m going to be the one physically asking the questions and writing the responses. This conversational format will means people will be more open and I can enquire for more detail on what they say if required.

The Document

The survey itself contains questions not only about past history of people using 'Uber' and its limitations but general questions about 'getting lifts' in general this allows me to know where my app stands out from the crowd and maybe what limitations it has.



Computer Science Survey

Have you used commercial ride sharing in the past (eg: Uber)?

Would you deem them expensive?

Would you deem them expensive compared to other forms of transport (eg: Bus, Tube, Train, Licensed Taxi)?

What would make you pick on or the other?

Have you ever had a lift from friends?

If so have they charged you petrol money (eg: 50p / £1)

Give some examples of where the lift was to:

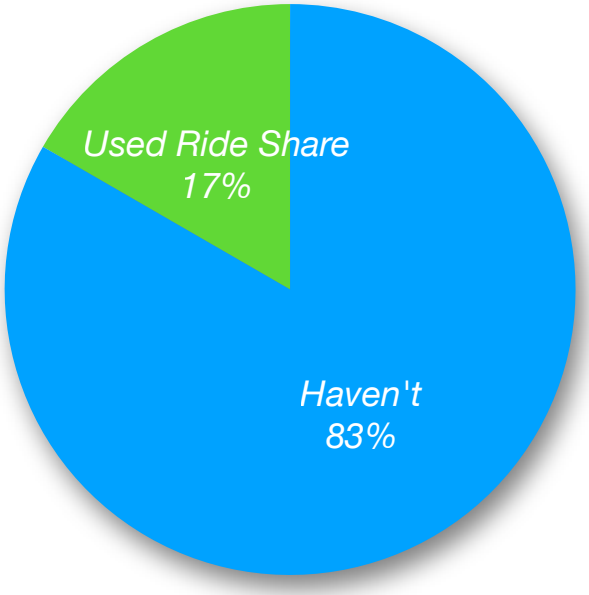
Have you ever had to ask around to find someone for a lift?

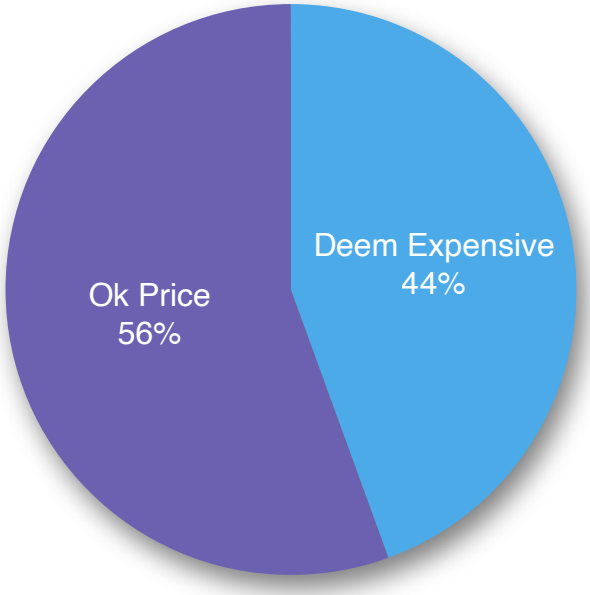
Would you feel like an Uber like app that focuses on getting rides from friends, would make you feel safe? If so please go into detail why you feel unsafe in Uber?

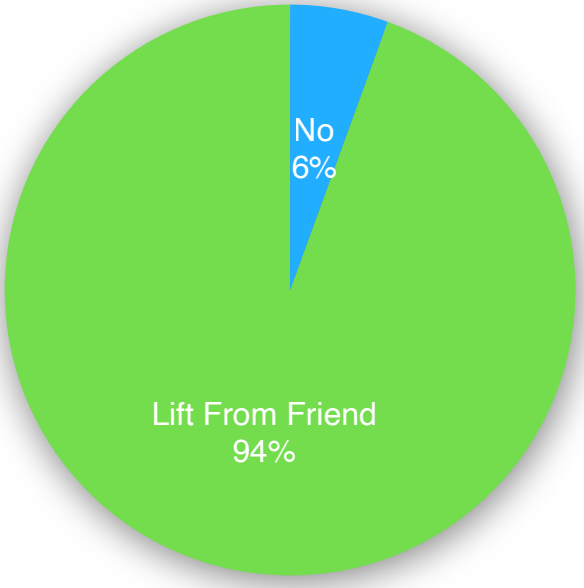
Do you drive yourself (have your full license)?

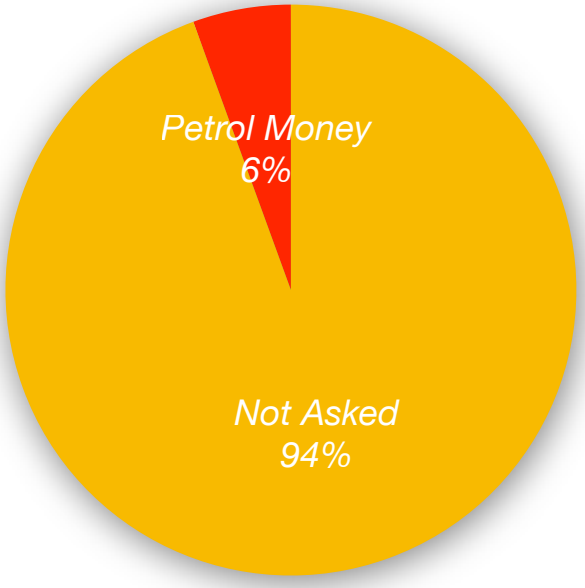
Do you have an iPhone? Is it running the latest major version (iOS 14)?

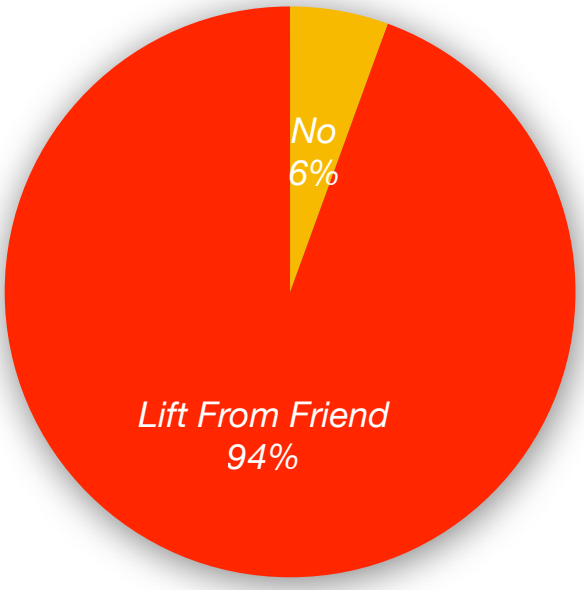
Visual Representation Of Data

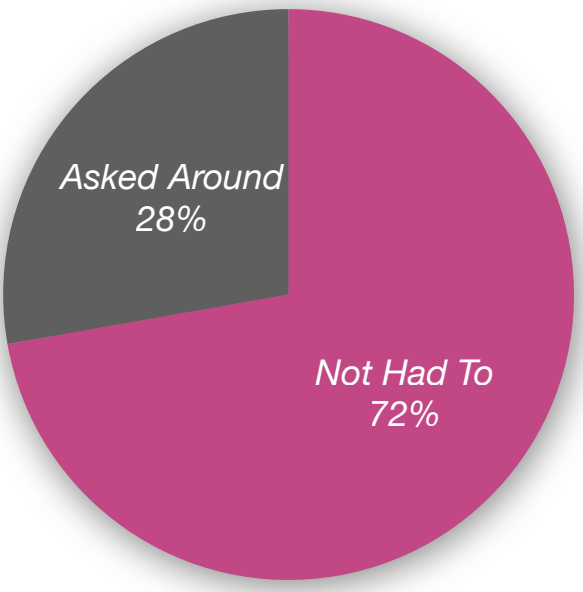


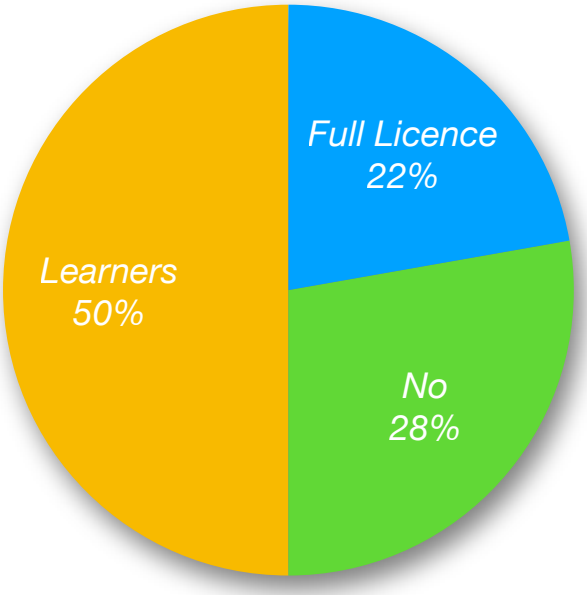


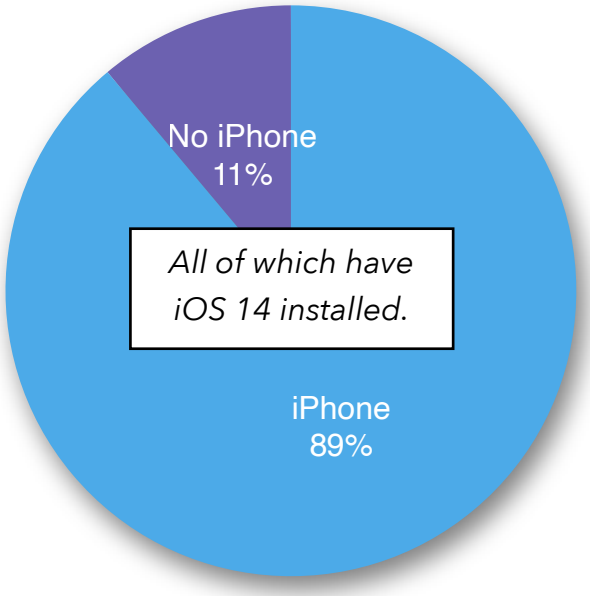












Analysis Of Data

As the data above shows, a large amount of people have had lifts from friends before, some having full licenses and a large majority being 'learners'. In the future they will also be able to drive people around. Especially as people turn 18 and go out for drinks, people will want to find sober riders ***THEY CAN TRUST***.

Trustworthiness was a reoccurring theme throughout, especially with women, as they felt said they would feel unsafe in an Uber, if they were not in a group. Feeling as because they didn't know the driver it would be a safeguarding issue.

Another reoccurring theme is the need for connivance as most said the reason they would pick an 'ride sharing' over other forms of transport are factors such as:

- Direct to the destination
- Apple Pay
- Faster than public transport
- The Uber will come to you (after summoning the cab via app)

As shown above the concept of 'ride sharing': the connivance, simplicity and the speed, are all reasons why someone would choose 'ride sharing'. However the downside, such as safety, massively outweigh these gains.

As my app would remove these externalities, while keeping the benefits. Especially features such as Apple Pay. Will hopefully mean users will adopt what I see as the future of 'ride sharing' for some cases.

However some responses did raise issues with the idea. If friends are not available then obviously you won't be able to get a ride. With Uber as the driver pool is so large especially in cities that you can always get a driver. On the other hand I guess Uber has the same problem as undeveloped small rural areas will not have enough coverage.

Success Criteria

Login System ~

- 3rd party login (O-Auth).
- Username and password (if O-Auth not possible).

Drivers ~

- Allow the user to be “available for rides” (online/offline).
- Track the users location when they are “available”.
- If driver engaged in ride automatically update status.
- Ability to decline and accept rides from riders.
- Use MapKit to show driver how to reach the riders location then to the destination for the rider.

Riders ~

- Fetching drivers that are friends with rider and are online.
- Ordering drivers by distance / price per mile
- Allow riders to select a driver and to request a ride to a location (riders destination)

Payment System ~

- Drivers can set their price per mile rate (updating database).
- Storing card details.
- Processing payment after ride is completed.

- Add fee percentage on top of price.
- “Pay with one click”

Limitations:

To make an app, especially a social one work effectively, you need a certain amount of users otherwise it is pointless. If there are no riders, drivers will see no need to have the app. If there are too many riders but not enough drivers the app becomes awkward to use due to too much waiting for a driver to become free. It’s a demand and supply problem. One of the things the app could provide but won’t due to time limitations and to keep it simple is to see other public transport options. So by the user using my app in a way it limits their options unless they ‘shop around’.

Paying through the app will be part of the process for completing the ride, however, what is stopping the user to just not pay at the end, or even another payment method (cash) especially if there is little cell signal at the drop of point causing the payment not to go through. Furthermore what if the rider doesn’t have the balance at the beginning of the ride. Guaranteed payment could be a problem.

Design

Decompose The Problem

To decompose the problem, I need to break down the entire application into a series of problems. These chunks are easier to comprehend, and to tackle in terms of programming.

These chunks are:
- Login System
- Relationships between users (Friends)
- Data Collection (User settings) & Tracking
- Requesting (accepting/declining) rides
- Driver screen and requesting payment

Human Interface Guidelines

When developing for iOS you have to follow strict design guidelines (Human Interface Guidelines), which yes can be irritating, but this standardisation across apps makes it easy for a user to download an app and instantly understand what everything does. Furthermore all apps follow the same design philosophy as the product they are on.

[User interaction tree.jpg - on full page]

User Interaction Between Chunks (v1)

The diagram on the page above sets out the screens the app will need and what's the user can see/manipulate in simple terms. I've used a top down hierarch structure to show that everything the user does originates from the main page. This gives the user a starting point, making easy to understand as the user can just back track and reach the main page again. Therefore the main page needs to be clear, and refined.

View Designs

Login / Register Page

This is what is shown if a user is not logged in or if this is the first time and they are registering for an account.

As said before as this app will use "Sign in with Apple" the user experience will be easy if you have an iPhone you have to have an Apple account. So in a way you are already registered.

When button is clicked many functions are executed in the background (see flow diagram on login).

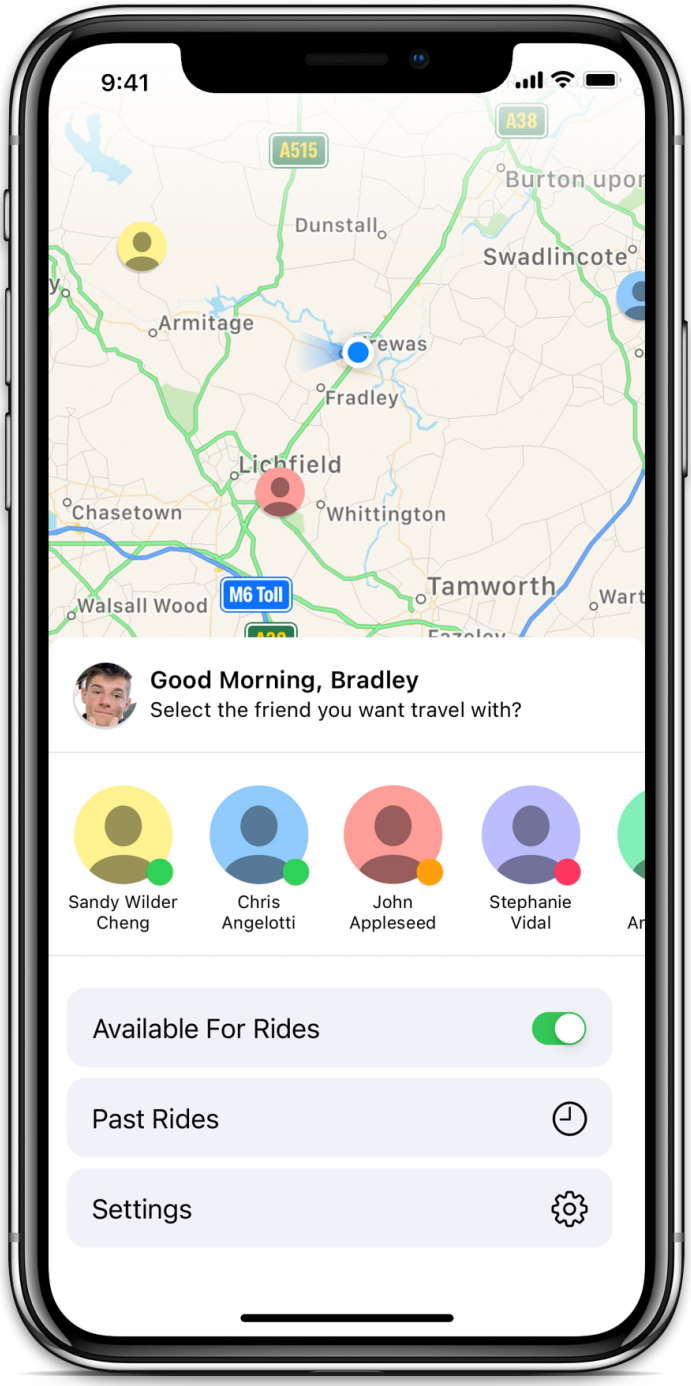
A form of biometric is asked for (FaceID/TouchID) then your logged in.

Main Page (V1)

This design is heavily inspired by the share sheet. A simple card like design, no clutter.

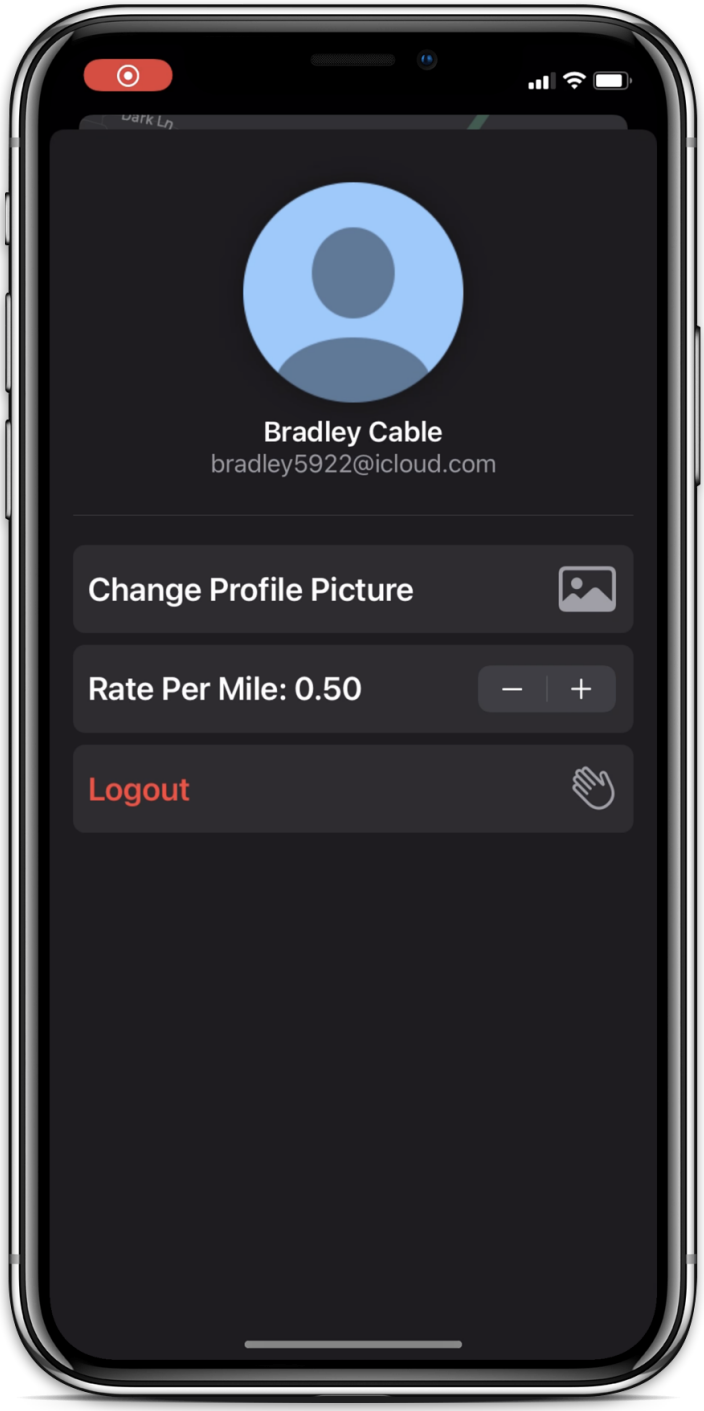
Showing your friends in a scrollview, as well as the map above, making it easy for you to see where they are in relation to you as well as if they are online. You can then click on them (either on the map or in the scroll view) to request a ride from them.

The buttons below and what they do could change. However they are large, simplistic and not many options. User is not overwhelmed. The icons next to them is the cherry on top.



Clicking on any of the interactive elements on this page (buttons, people, etc), will display another view modally. This means on top of the root view. This will keep the card like design throughout.

Indicators on the friends will make it easy for the user to interpret data quickly. Green, for available for rides. Orange means available but currently fulfilling a ride and red obviously means unavailable..



User Profile (v1)

This is obviously where the user can update profile details and settings, as the app fleshes out I will move things in and out of settings.

Profile picture front and centre. Mainly to make it look more interesting than just a wall of buttons. As well as viewing your profile picture when you change it.

Same button design as on the main page, keeping consistency across the entire app.

On the right hand side either an icon or a control of some sorts.

Allowing the user to customise their account gives it a more friendly vibe not just to other users but allows them to stand out, for example setting a lower “RPM” then most friends.

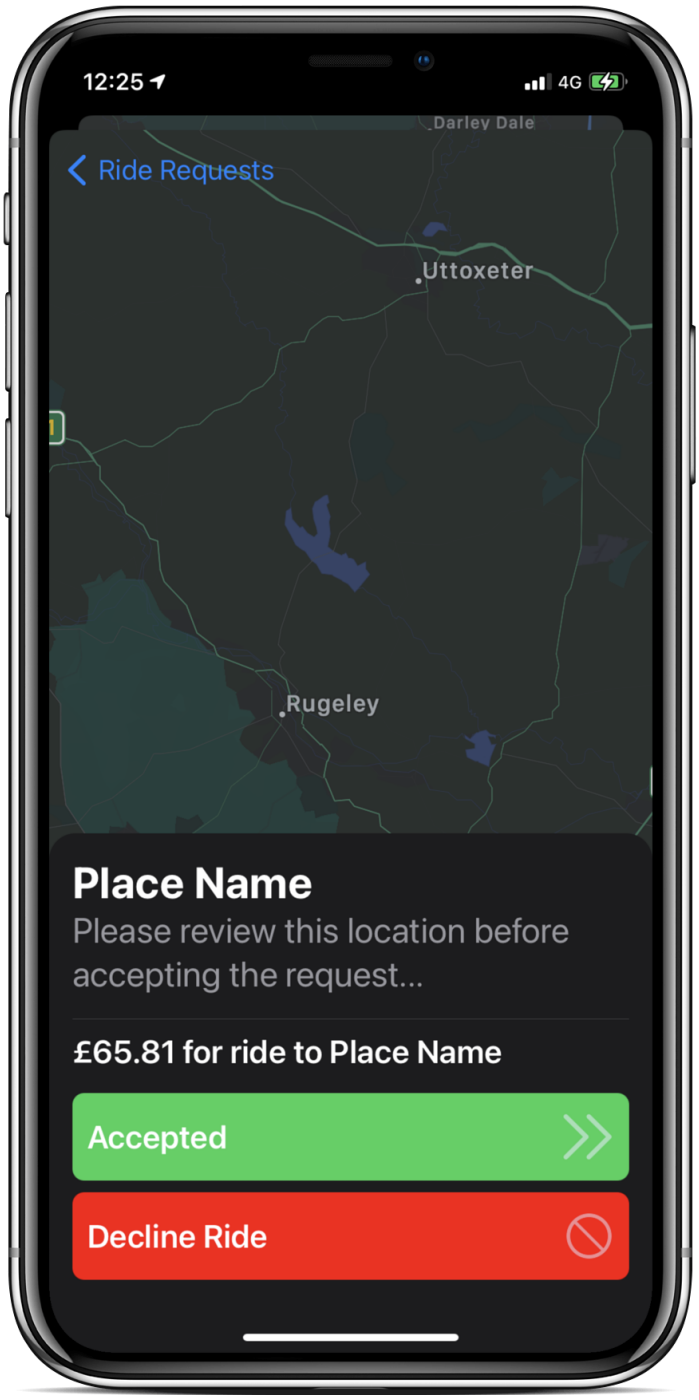
Side note: Mock-ups will change between light and dark designs. Not because this is how this page will look. This is more showing off SwiftUI’s built in “Dark Mode” compatibility.

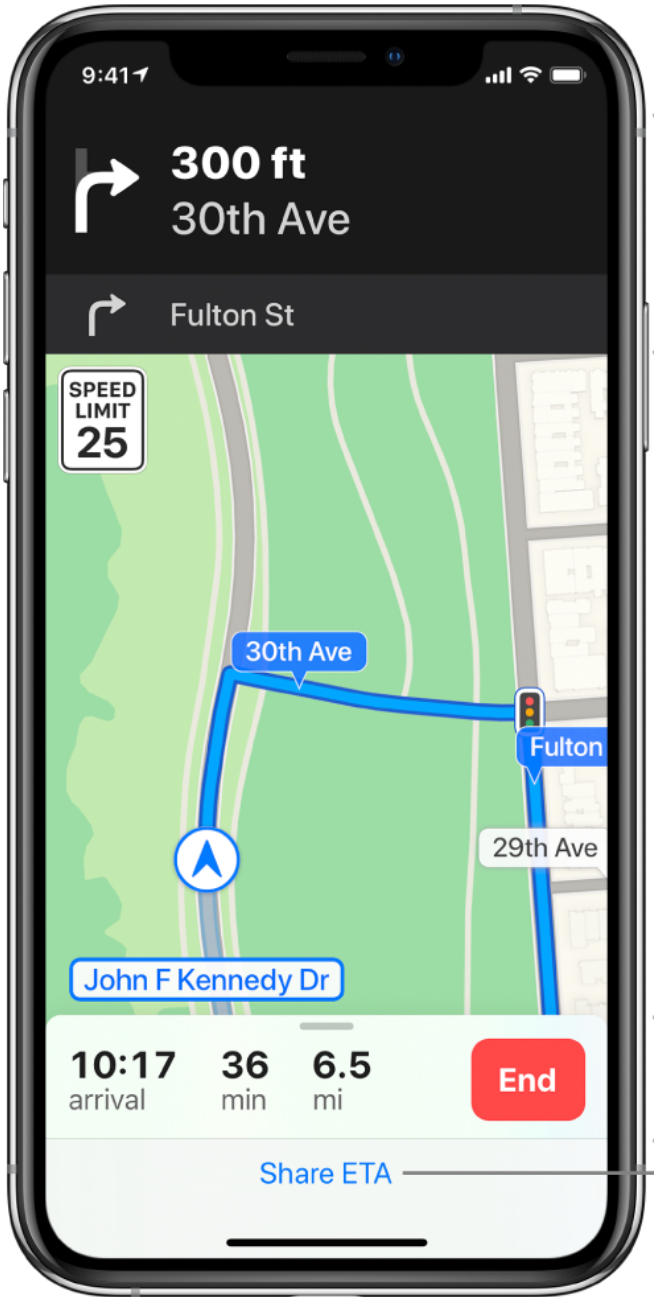
Location Detail (v1)

This view will be used when a user clicks on a ride request to them or requests a ride from another user. Its purpose is showing the location and price. Obviously button text, etc, will change depending on if you're a driver or rider.

The map at the top will show a zoomed in map of the location in question as well as a pin (📍) not shown here, of the exact location where the user needs to go.

The clean and simple design with bright and large coloured buttons, makes the process intuitive for the user.



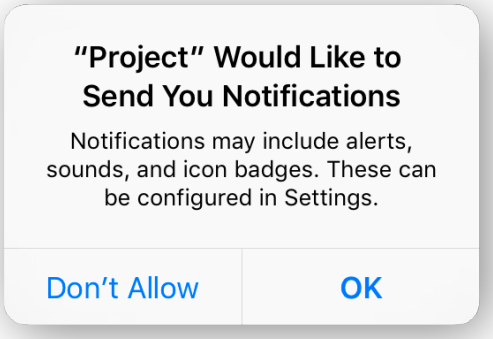


Map View (v1)

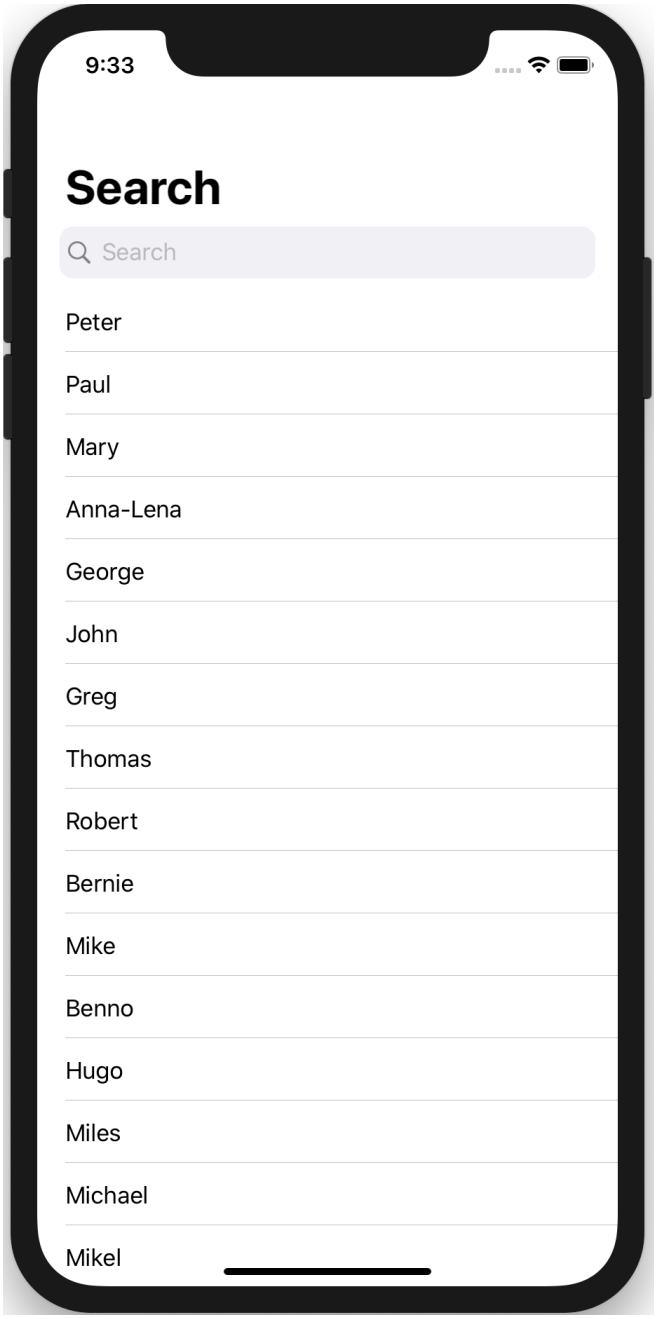
At this current moment, I have no custom design for this screen as I will either use a first or third party library, as building a navigation system like shown would not only be pointless as a there are libraries that do the job, put pointless as that would be a programming project in it self. From tracking the users location, drawing the route on the map, working out ETA, etc.

This is a screenshot of the inbuilt navigation view in Apple Maps. Which matches with the design of the app overall as I'm trying to follow the Apple design scheme.

When a user reaches the location of the user they are picking up, a alert box will appear with the title "*Waiting For Conformation*", this is when the rider will confirm they are in the car with the driver. This makes sure that the ride actually takes place. Stopping rides being scammed out of money. It also stop the navigation giving a break before the ride continues. The alert box will only disappear when the user confirms.



Friends Search (v1)



The image shown on the left is just a basic search screen found on *'Stackoverflow'*. Nothing special about this page just a box to type in the user name and usernames shown below.

If I was to adapt this is any way I would show the users profile picture to the left of the name to make it easier to identify people, especially with people with common names/ usernames.

The list will update when something is typed to allow it to live update. This makes it easy for the user in two ways. They do not have to click a search button when they are finished and if they are unsure on the full username they can partial type, then see what results are returned. Especially if I add the profile picture next to it, it will make it even easier.

When a friend is clicked a detail view will be shown with the users profile picture and then a big green request button, same will apply to friend requests. The users name in a cell that when clicked will show a detail with profile picture. In this case 2 buttons accept and decline.

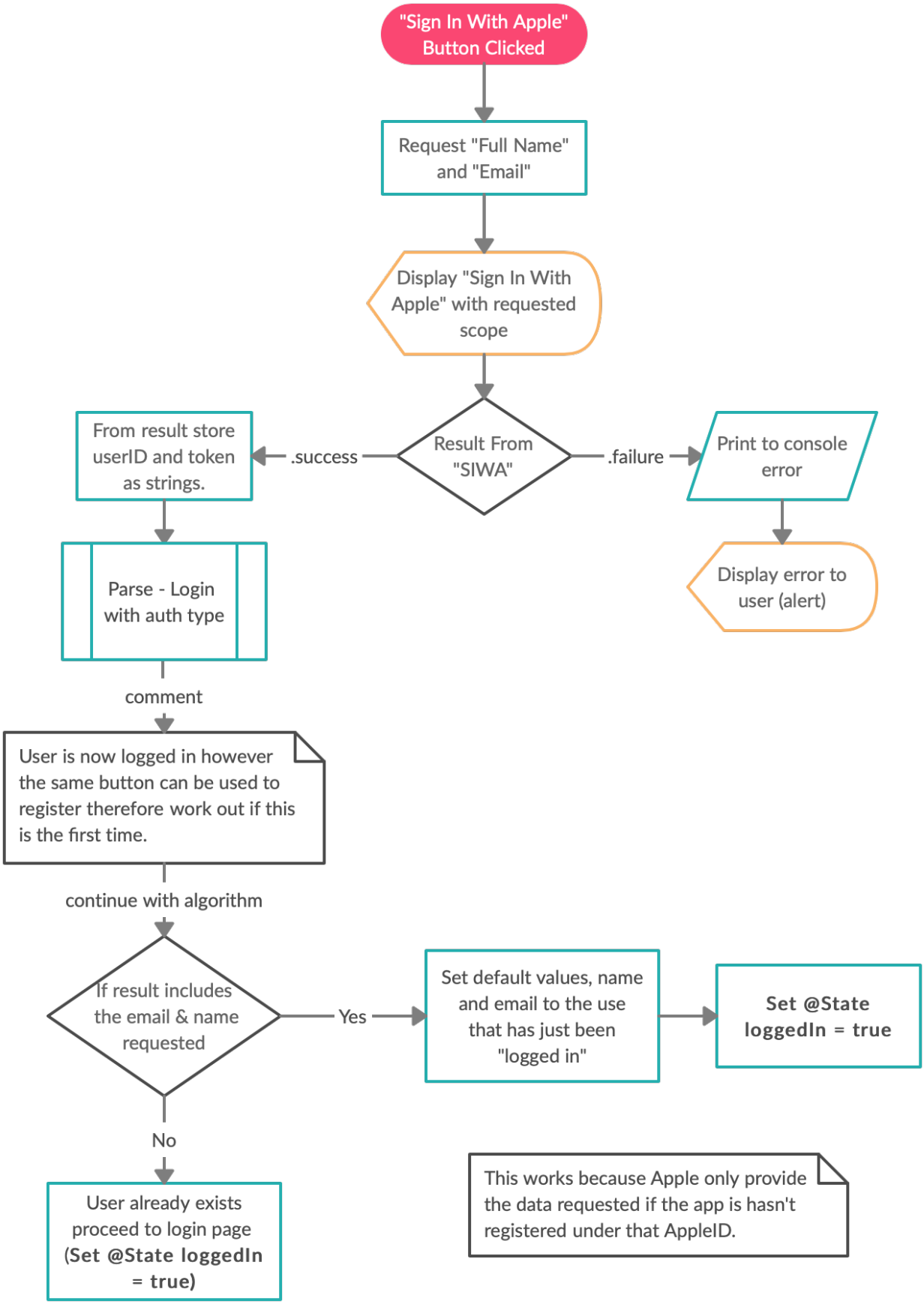
Establishing & Justifying Variables / Data Structors

Name	Data Type	Data Stored	Justifying
MainPage.swift			
user	currentUser (class I defined)	multiple (eg: string, int, bool, struct, etc)	This stores all the data related to the current user, stored using a class, 'currentUser', which conforms to the protocol observable object. This allows view updates on any changes to this var.
friend	userData	multiple (eg: string, int, bool, struct, etc)	This stores an array of structs, these can contain all relevant data about a said user. 'userData'
userDataController.swift			
userData	Struct	multiple (eg: string, int, bool, struct, etc)	Allows me to make sure data has a 'set standard' of sorts allowing me to know that all userData will be stored in this format. Any time I fetch data from the database I store I in that variable that conforms to this.
DatabaseFields	Enum	Providing values to a cases.	When this enum is used I can give it a case and it converts that to a string. I'm using it so it allows me to autofill (see below in development).
currentUser	This is not strictly a data structure as its a class which conforms to ObservableObject	multiple (eg: string, int, bool, struct, etc)	By conforming to the protocol observable object. This allows view updates on any changes to the @Published vars inside of it. Therefore any changes to a variable which uses this class will update all views that use that variable.
rideRequest	Struct	multiple (eg: string, int, bool, struct, etc)	A way of organising data related to a ride request, containing: the driver, rider, location, price, etc.
Requests.swift			
rideRequests	Array	An array of rideRequest structs	This array can then be used on a foreach in SwiftUI. To show the user what rides they can accept. Then depending on if they accept or not use that data to take out certain actions.
waitForUser	Bool	True / False	If the user accepts the request to drive, this variable stores if the Nav is waiting for the user to confirm that they are in the car. As this is the "root" of the variable, the children below are bindings. This allows the nav view to be blurred if this Bool is true.
Requests.swift			

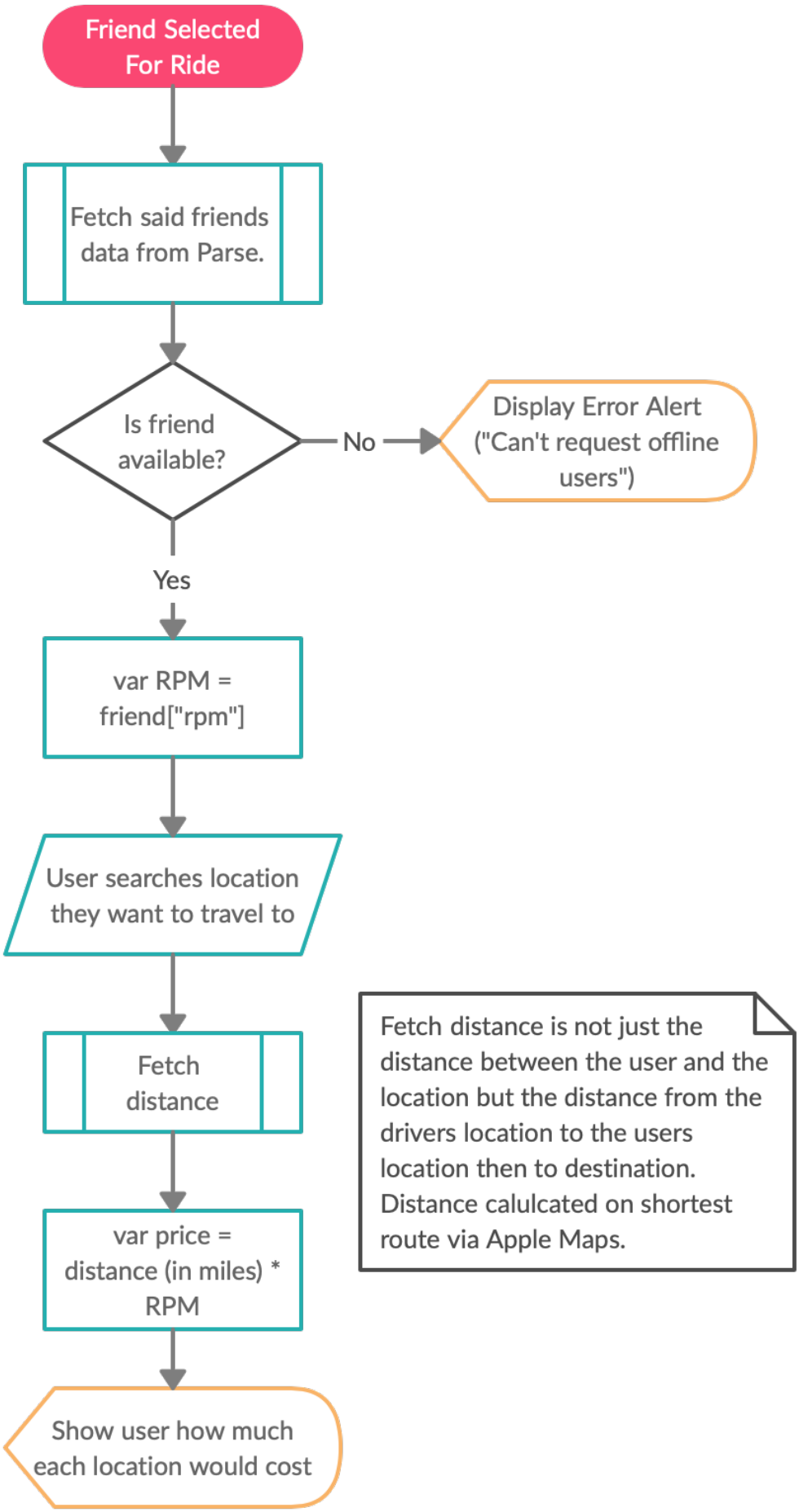
Name	Data Type	Data Stored	Justifying
friendRequest	Struct	An ID for the request, the requestID in the Parse database and the user data relevant to it.	As this struct conforms to the protocol identifiable, every object must have a unique ID which I can then use later in a ‘For Each’ loop to create a ‘List’. The requestID is the ‘objectId’ for that field in the Parse database which means I can update the accept field when the user clicks a certain button. The userData() stored can then be used again when creating the list to show all the relevant data.

Algorithm - Flowcharts

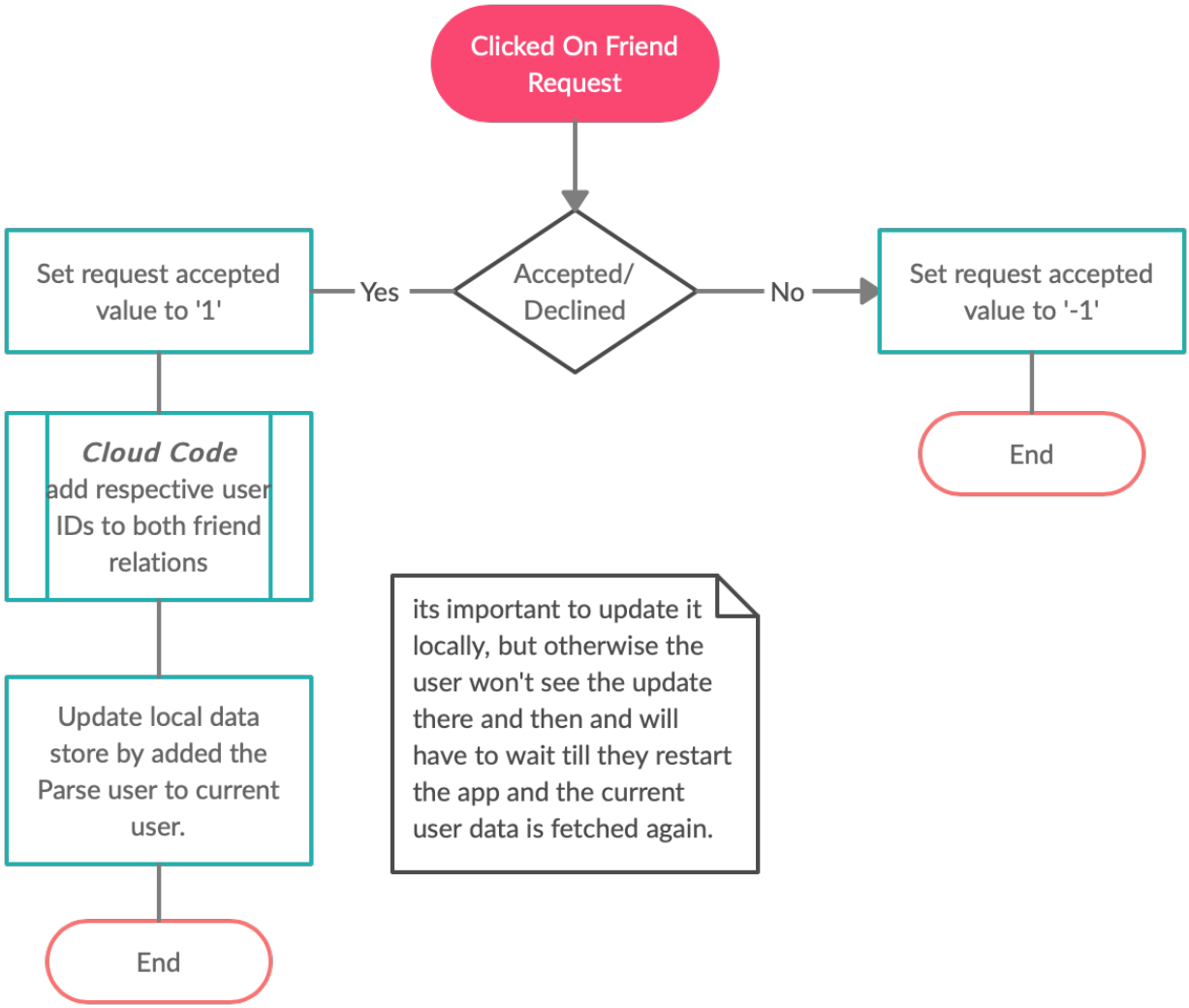
Login System



Requesting Rides



Friends List Page:



Development

General Overview of SwiftUI

I will use this as an opportunity to explain parts of SwiftUI (explained with help from reference 13) and how this helps me in the future.

Bit of history - UIKit and SwiftUI

SwiftUI in terms of a programming toolkit is *VERY* new being released only in June 2019. Therefore as of date of writing. It has only recently became a year old. Before SwiftUI the toolkit that was used was UIKit. UIKit is know as a "imperative UI", this is when you the developer has to constantly update the UI when something changes. This is very messy when larger views can have many functions updating different bits of data (the "state" of the UI).

The example my reference (13) gives is perfect so to paraphrase:

"If you have a boolean property that affects the UI, that boolean has two possible states in the first place, "true" and "false". Make that two booleans and you have four states. Each would require some change to the UI. Now add strings, ints, times and dates. The complexity is endless".

Why SwiftUI is better and useful

SwiftUI is declarative meaning we can tell iOS all the possible states of a view. Now this may sound complicated. But simply we can set rules on how the UI looks and then when the state changes in the app these rules are followed to get the desired state. Comparing that with UIKit where you would tell SwiftUI how to get

the state and when the state itself changes. Instead this is all handled by SwiftUI.

User Data Controller

This leads me onto the data and states themselves that will be changed, obviously the data is coming from a mobile backend. In my case this backend is the very popular open source Parse Server.

The app then has to store this data coming from Parse in the form of a user of some sorts.

Parse Communicator

To make it easy for me to fetch data/upload data to the mobile backend I made a class called "ParseCommunicator".

This condenses the code, and simplifies errors as if something is not updating / returning correctly then I only need to refer to this class.

An example of how I'm condensing reoccurring code is the function "syncWithParse()" as it says on the tin it main job is to sync data to Parse.

[/images/Screenshots of Code/userDataController/Parse Comunicator/syncWithParse.png]

The value is allowed to be any data type, as there are many data types used in the database. I could have added extra protection to make sure that only a certain data type could update a certain field but that would be over-engineering on my part.

However, to make too easy for me as I will be calling this function many times thought development the "keyToUpdate" parameter is

a enum which allows two things. It allows the IDE to autocomplete the field name so I don't have to type it out every time, as well as preventing hard to find errors due to misspellings.

[/images/Screenshots of Code/userDataController/Parse Comunicator/.png]

This is the start of a homemade framework, that will be updated and changed if needs be. Therefore I will date this section if any more key sections are added.

Standardising The User Data Structure

To make sure that user data is effectively stored as well as type-safe. I have used a struct. This means when a user is accessed data you get back will be what is expected. Unlike if you just stored user properties in an Array, for example where its very possible for data to be added in a random order if data coming in asynchronously etc.

[/images/Screenshots of Code/userDataController/Parse Comunicator/userDataStruct.png]

As well as this it makes it highly reusable. For example to make a friends list. I just have a an array of users. Every user is stored in this struct format.

However as development went on using the user struct for current user data this became problematic, which will become clear in a minuet.

Current User Data - Observable Object

The current user can't just be a struct because its changing all the time. For example: profile picture, rate per mile, availability, if driving or not, etc. All these will effect the interface ether in a significant or minor way. Therefore I need to tell SwiftUI if this changes in any way the UI needs to be reloaded. Which is turn means the UI is always consistent with the data it is showing. Simply they are linked together.

It looks exactly the same as a struct in everything but name, the difference being, it conforms to the Observable Object protocol (which gives it the ability to tell SwiftUI to update).

[/images/Screenshots of Code/userDataController/Parse Comunicator/currentUserOO.png]

When this class is used to declare a variable at the top of a view. That means any changes to that variable will mean the UI is refreshed. As long as the main view you pass the variable down to any subviews changes of data in those subviews will also propagate up to the parent views.

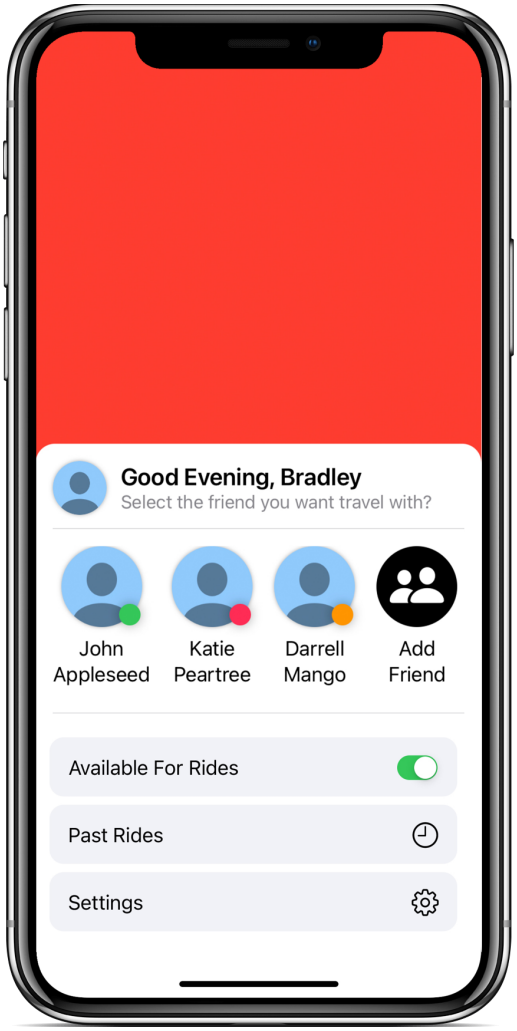
[/images/Screenshots of Code/userDataController/Parse Comunicator/OO.png]

Main Screen Development

The main screen doesn't really have any important technical bits when it comes to development because as said above in my design its main purpose is a "landing page" for the app where you can then bring up different parts of the app modally. However everything said above (User Data Control) is a prerequisite of sorts to understand why certain things were done later on in this development process.

The Main Card - Placeholder Data Only

Firstly I made the card design with some placeholder data. Using simple SwiftUI features such a VStacks, ZStacks, ScrollViews, Image, RectangularViews, etc. All these obviously having modifiers of sorts (for example with "Image" rounded corners).



Starting with only placeholder data, images, text, etc. It allowed me to get the general feel of what the app would do. At this point none of the buttons worked, purely aesthetic.

The red square above was just a placeholder for where the map would go.

As I designed it using a ZStack, when I finished the map view. That subview could easily be added behind.

The card obviously using a VStack, as it is a vertical layout.

The Use Of Subviews

As mentioned many times before subview (views inside views) are used, and are used a lot as this is recommended by Apple themselves. Due to SwiftUI's nature extracting one large view into many subviews combined doesn't have any effect on the loading speed of the view. So it is recommended to allow maintainability as well as just general clean code and good practice.

[/images/Screenshots of Code/userDataController/Parse Comunicator/currentUserOO.png]

This is from main view (a bit later on in development as MapView is finished). But obviously the card above is not coded into that view because thats messy. Also in the future if the card needs to be changed anyway it's easy to find what you need to change.

Therefore the Card view is extracted into a subview which then contains said view for card. As mentioned above the user var is passed into the view to allow it to access the current user data as well as update all views if anything changes.

UIViewRepresentable & MKMapView

SwiftUI is a new language only being out for just over a year at the time of writing/development. Therefore the extensive feature set of iOS hasn't been fully implemented into the framework unlike UIKit.

That means even some key features are not natively available. One of those that I use been "MKMapView". However SwiftUI has a solution to this problem: UIViewRepresentable.

[/images/Screenshots of Code/mainPage/map-UIViewRepresentable.png]

This allows you to use UIKit directly inside SwiftUI therefore giving you the best of both worlds. The mature, highly documented UIKit and the new, highly optimised, declarative based system of SwiftUI.

The Asynchronous Nature Of Views & Data

Beauty of the app isn't everything. Once the UI is in place in some form, I need to populate placeholders such as profile images of friends. This data obviously from Parse (the mobile backend).

However fetching data from a database can start to causes problems. These are the ones I faced.

My first attempt to fetch data was to use ".onAppear()" this is one of the "modifiers" available in SwiftUI. Allowing you to run bits of code after that view it is "modifying" is shown to the user - it appears.

I thought at the time that would be the best way to do it as it is normal for an app to show placeholders on first launch while data is being fetched.

This is obviously an example from the popular social media app Facebook. As the posts are loading in they show "blank posts".

[/images/placeholder example.png]

This picture itself was from an article called "How to make you app look like it's loading faster!". Which says it all really. Which is why I

thought “onAppear()” would of worked perfectly especially on modern day technologies such as 4G.

So what I had was an empty user struct (user struct talked about above). Which when “onAppear()” was executed it used the “syncWithParse()” function in ParseCommunicator to get the data from the database which I can set the values of the empty struct to all the data fetched. However when I tried this, I saw that it did exactly what I wanted (in the background) but I realised nothing would actually change because the view is already loaded. I tried to find a way to force reload the view (which looking back would of been bad practice even if I could do that). So I had to find another way. This is when I discovered the usefulness of @ObservableObject and @State.

Ive already discussed this above in the sense of what they do, but in context they would mean that when I changed data it would reload the view and all would be good. However there were still problems, this is exactly how I fixed this problem.

I created a LoadingScreen() view which only contains the globe logo with a spinning animation (basically a loading activity view). Which as soon as it appears it fetches the data from Parse Server.

When the app is launched there is just a HoldingPage() view which checks if a variable called “isLoadingData” is true, which it is by default.

If it is true it returns LoadingScreen() otherwise it returns HoldingPage() (which is the actual page shown above). I’m able to just return views inside this HoldingPage() struct. These structs

conform to the View protocol. Which returns a view anyway. Therefore I can just force return a view.

The LoadingScreen() after it has fetched the data it sets showLoadingScreen to false. This is again is just an observable object. So when this is changed then the HoldingPage() is just reloaded as showLoadingPage is declared in HoldingPage() and then passed into LoadingScreen() as a parameter. Therefore when it's changed in this child view, the parent view (HoldingPage) is reloaded. This effective means that the checks above on "isLoadingData" returns HoldingPage() instead.

[/images/Screenshots of Code/mainPage/loadingScreen.png]

//MainPage renamed to HoldingPage

As internet is so fast these days, WiFi or mobile data. You can't even see the loading screen in usage because the data is received and MainScreen() refreshed so quickly.

As a result the actual experience of all this for the user is impossible to even notice especially as its not actually fetching the larger file types such as images at that point. Only datatypes such as strings, ints, bools and floats are fetched.

Lazy Loading & UX

As said above the solution I came up with doesn't have any loading speed issues as its not fetching anything large like images. This is only because of "Lazy Loading" but before I did try and just load the images at the same time as everything else but that did cause VERY SLOW loading times.

So in ParseComunicator I tried to make my own function that would return and Image when loaded. However in this case the app just crashed as the App was expecting an image where I called enough and because it took so long it crashed.

Here is the failed code that used Dispatch Queue (Background thread):

[/images/Screenshots of Code/mainPage/loadingScreen.png]

So I decided concept known as “Lazy loading”, which is normally used for optimisation, only loading/initialising data when necessary. Instead of doing it when “necessary”. All profile images are loaded from URL using KingFisher on the background thread. In the mean time a placeholder is used. When loading said image is finished the placeholder is replaced with the actual image. This makes the application appear as it is loading faster. When in reality it's being done when the user is already using the app.

[/images/Screenshots of Code/mainPage/profilePic.png]

Overview of advantages:

- Loading of MainScreen() will be quicker.
- Internet usage will be spread across the use of an application, not just one single download at boot of the application.
- Lazy loading is best practice in most apps anyway.

When any user is fetched the image itself is not fetched only the PFFileObject, this object contains the URL of where the image is stored. This can be accessed by the KingFisher library to load the

image. As KingFisher couldn't use the native Parse way of loading images.

```
let profilePictureURL = (pUser["profilePicture"] as! PFFileObject).url!
```

How UI and UX can change during development

As I was developing the main screen view I realised that every possible feature I thought of was followed by me adding a button to the bottom of the list of buttons that I already had. Slowly overtime the card was just getting bigger and bigger.

My response to this was to move 'Past Rides' from the main page into 'Settings', as well as move the availability switch into the user profile. Firstly 'Past Rides' is not vital data that the user needs to see there and then.

Secondly the availability switch had two issues:

- The 'Past Rides' button was not something the user would need to consistently change on opening the app. More realistic use would be at the start and end of the night. Therefore I moved it into 'Settings'.
- It also created UX inconsistency between buttons. As all the other buttons opened another page in sheet form. Therefore that would be the only button doing something different, this would be illogical.

[/images/mainscreenV2.png]

Continuing on UX. Giving a hierarchy to your views allows a user to focus, as everything you would immediately need when you open the app is at your fingertips (literally), no 'pointless' buttons.

Side note: from this point forward 'Settings' will be referred to as 'User Profile', mainly because 'Settings' sounds scary to the user.

FriendScroll() - Changes During Development (21/10/20)

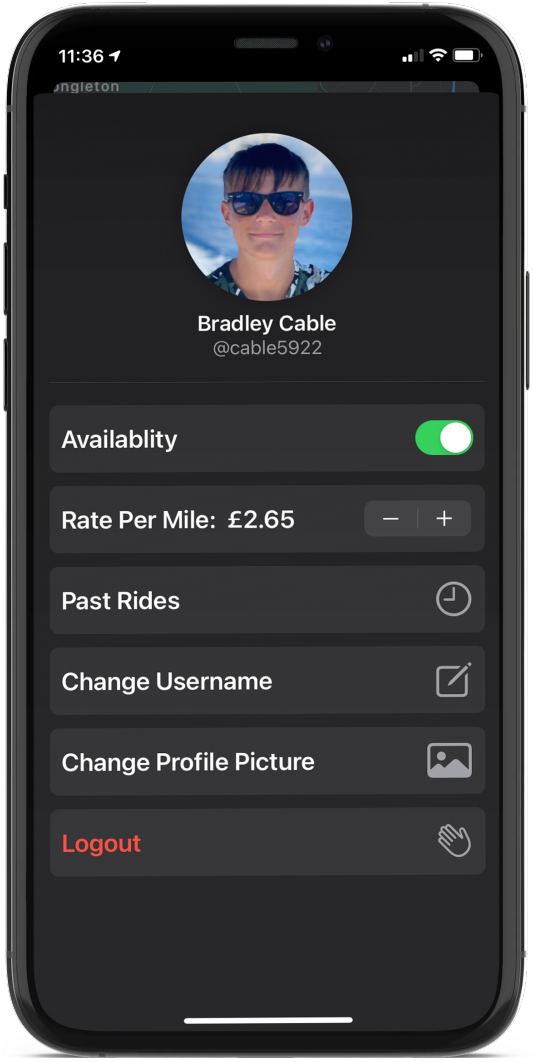
To give a better experience for the user I decided to use the library "Parse Live Query" which allows your application to "Subscribe" to a query. Which in simple terms allows the app to get notified whenever certain objects update in the database.

In this specific use case I am updating the bubbles next to each users profile picture which states if they are online/driving/offline. As before it only fetched this data on launch therefore get out of sync with the database very quickly.

[paseLiveQuery friennds main page]

This code above goes though all the users friends and add all their objectId's into an array, including the current users objectId. It then query's and subscribes to all users in that array. Whenever any of them are updated it finds which user (if no user found the whole friends list is fetched from scratch) that was and changes all relevant values. In this case the ones related to user status (DispatchQueue.main just makes sure that code runs on the main thread as anything updating the UI has to be).

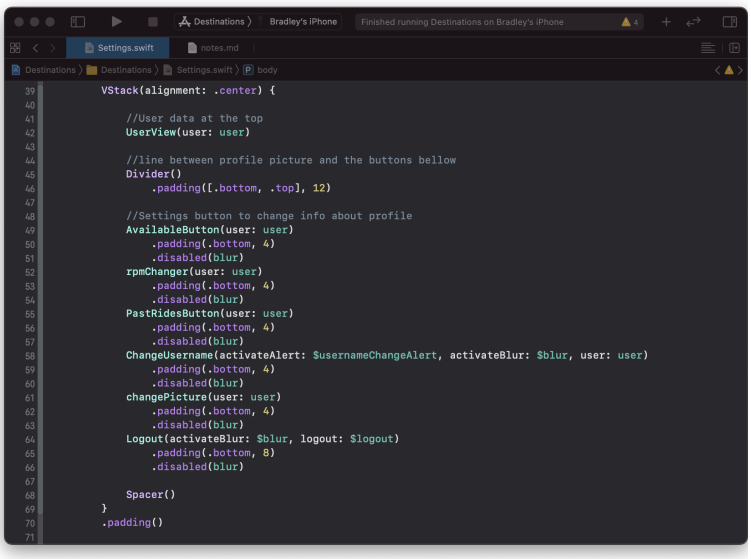
User Profile Page



Starting with the easiest part of the hierarchy from the home page. The 'User Profile' page. Not much prior design (on the actual code side) was done for the user profile page mainly because there is nothing to it apart from buttons which update a value in a database.

It has already been discussed in great detail how I will update and fetch data from the database. Therefore it doesn't need to be repeated. parseCommunicator().syncWithParse() did all the heavy lifting. This is the exact reason why I created this class to simplify things down the line.

This main page again are just subviews embedded in subviews. Each button is its own view, which makes maintainability really easy, as if you have a problem with one button. You will not be finding it in spaghetti code. You find that views struct. All code related to that is in there. As you can see the page show is just a vertical stack of views.



```
VStack(alignment: .center) {

    //User data at the top
    UserView(user: user)

    //line between profile picture and the buttons bellow
    Divider()
        .padding([.bottom, .top], 12)

    //Settings button to change info about profile
    AvailableButton(user: user)
        .padding(.bottom, 4)
        .disabled(blur)
    rpmChanger(user: user)
        .padding(.bottom, 4)
        .disabled(blur)
    PastRidesButton(user: user)
        .padding(.bottom, 4)
        .disabled(blur)
    ChangeUsername(activateAlert: $usernameChangeAlert, activateBlur: $blur, user: user)
        .padding(.bottom, 4)
        .disabled(blur)
    changePicture(user: user)
        .padding(.bottom, 4)
        .disabled(blur)
    Logout(activateBlur: $blur, logout: $logout)
        .padding(.bottom, 8)
        .disabled(blur)

    Spacer()
}
.padding()
```

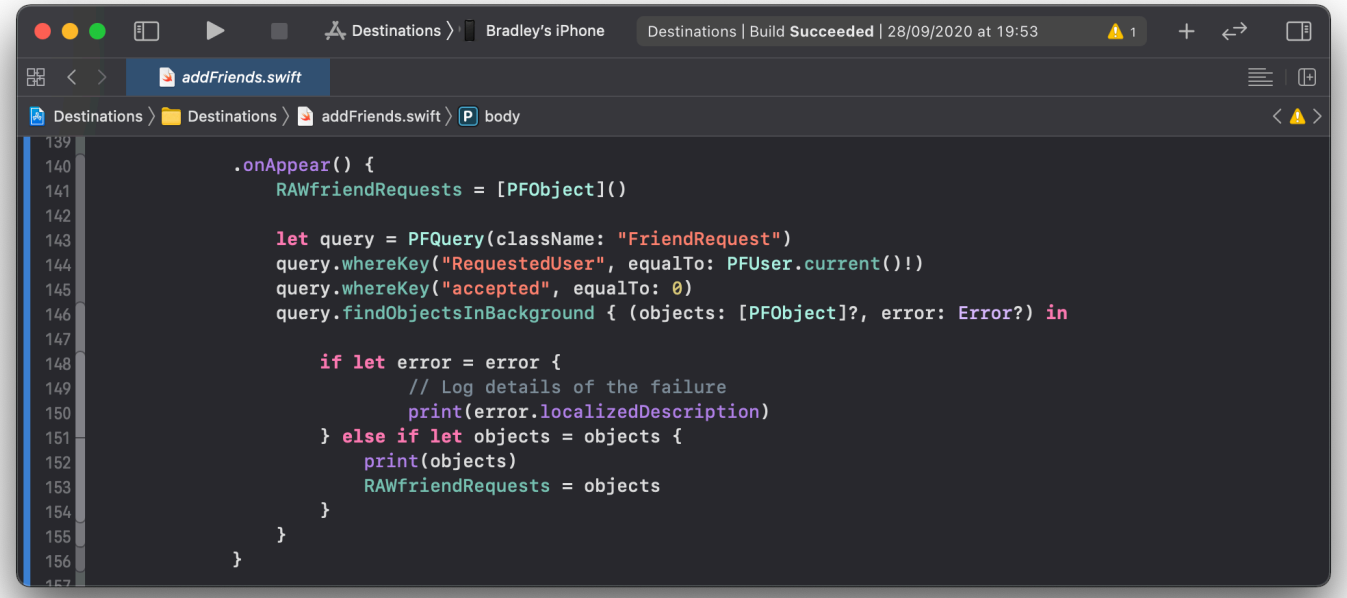
This is a section of code from the user availability button view , which on change of the '@State' variable (which stores the value of the switch), it updates the Parse database using 'pareCommunicator()'. As you can tell this makes the code a lot cleaner and if there are any problem with it, you only have to refer to the one function.

Then the 'user.available' below is just it updating the local datastore. That same kind of code is used throughout this, just updating different fields.

Friends Page

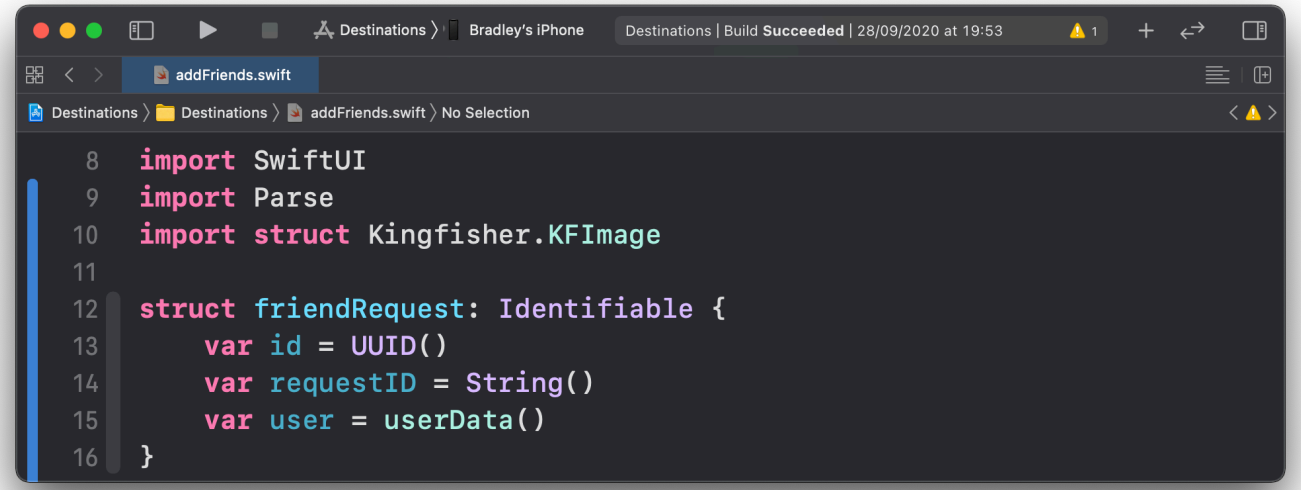
Friend Requests

Firstly the most obvious thing needed was to show what friend requests the user has. So that was an easy check, just fetching some data from the database. However because it was fetching from another table (not users) 'parseCommunicator()' could not be used at this time.



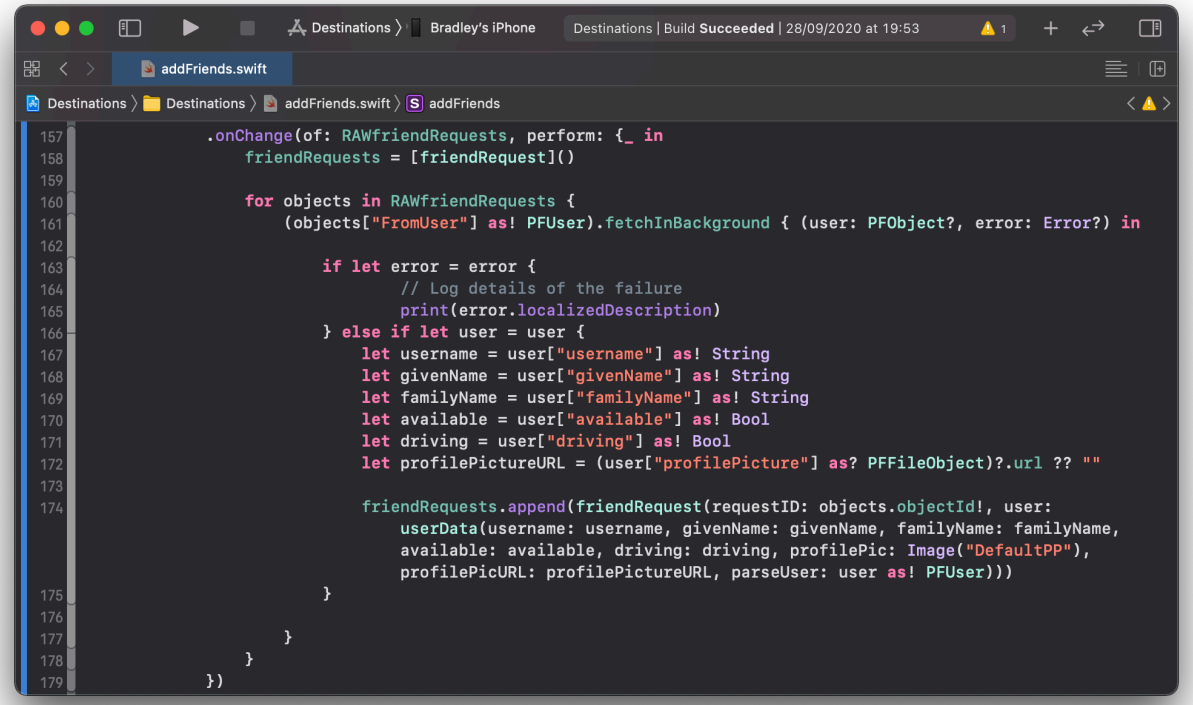
As this shows, fetches data from any data in “FriendRequest” where the request has been undecided (0) and that the user requested is the current one. All fetched requests are then added to an array.

However this array (“RAWfriendRequests”) cannot be used yet in its current form, because the field that contains the data of the user its from is only a ‘Pointer’ which means it doesn’t contain any actual user data just said objectID of user. Therefore to make this data useful I actually have to fetch this as well.



I decided I needed to create a struct to make it easy for me to organise friend request data this containing:

- The object ID of the friend request in the database.
- The actual user data of the user its from.
- An UUID for said struct object so it conforms to Identifiable which allows me then to use it in a SwiftUI list easily using ForEach.



I can now go though all the "RAWfriendRequests" fetch all the data I need related to the user, conform that to userData, and but that and everything else into that struct to be used.

Now all that needs to be done is when the accept button is clicked it appends the friend to the users friends locally. Updates the request so its accepted (1), therefore won't be fetched again.

Friend Requests - Changes During Development (21/10/20)

As stated above I am now using "Parse Live Query" meaning that whenever the user or the users friends objects are updated in the database, that function gets notified as it is "subscribed" to it.

Thus makes updating "friends list" locally (like above) is now redundant as it will be added anyway because the app is notified to a change to the current user object (added friend relation).

However this causes another problem of the user not being subscribed to this new friend. So any changes to that new friend

will not be live updated till the app is relaunched and a new subscribed user list is created (see above to understand further).

Friend Requests - Continued

Lastly some Cloud Code to update both users friends list. For obvious security reasons another user cannot update data of another user. Therefore it passes the objectID the friend to the Parse Server which can modify both of them using the 'Master Key'.

Cloud code here

Searching For Friends

Before the program searches for a user from the username that they have entered, it first retrieves all the logged in users friends. It can then form a 'Parse Query'. This 'Parse Query' checks:

- The username contains the text entered by the user.
- Then username is not equal to the current users username.
- It then uses all the users friends and loops though checking that no users that they are already friends with appear in that search.

The formed 'Parse Query' can then be used to fetch the users. Before looping though the fetched data a variable called 'conformedData', which is an array of 'userData', is created. It then for loops though the returned data from the database and feeding it into the struct ('userData'). As said before, because 'userData' is 'Identifiable', I am able to use this in a 'ForEach' to create a list to show to the user.

Evaluation

Lack Of Payment

As said in the design and analysis section, at the end of the ride I was going to make the user pay for obviously the ride they have done. I have decided during development, to not program payment for two reasons. Firstly because this app is only a computer science project and would be very time consuming, this would take time out of writing this report. Secondly as it would be mainly using Stripe's library. It would just be me following documentation.

Relationship Between Users

Friend User Friends

Unfriend User

Bibliography

1. Business Insider. 2019. *Uber Scored A Major Victory When The US Government Ruled Drivers Aren't Employees, But Not Everyone Is Happy*. [online] Available at: <https://www.businessinsider.com/uber-drivers-disappointed-ruling-not-employees-2019-5> [Accessed 20 March 2020].

2. BBC News. 2019. *Uber Loses Licence To Operate In London*. [online] Available at: <https://www.bbc.com/news/business-50544283> [Accessed 20 March 2020].

3. Uber. 2020. *Legal | Uber*. [online] Available at: <https://www.uber.com/legal/en/document/?name=general-community-guidelines&country=great-britain&lang=en-gb> [Accessed 23 March 2020].

4. Ft.com. 2019. *Bus Funding In England Slashed By 40% In 10 Years*. [online] Available at: <https://www.ft.com/content/ad5743bc-f673-11e9-9ef3-eca8fc8f2d65> [Accessed 7 April 2020].

5. Money Advice Service. 2019. *What Is The Average Cost Of Car Insurance?*. [online] Your Money Advice. Available at: <https://www.moneyadviceservice.org.uk/blog/what-is-the-average-cost-of-car-insurance> [Accessed 7 April 2020].

6. Yorkshireeveningpost.co.uk. 2019. *Hour-Long Delays And Bus Cancellations In Another Night Of Leeds Traffic Chaos*. [online] Available at: <https://www.yorkshireeveningpost.co.uk/news/traffic-and-travel/hour-long-delays-and-bus-cancellations-another-night-leeds-traffic-chaos-1338894> [Accessed 7 April 2020].

7. Smith, D., n.d. iOS *Version Stats*. [online] David-smith.org. Available at: <https://david-smith.org/iosversionstats/> [Accessed 7 April 2020].

8. Johnson, G., 2019. *Taxi Vs Uber Comparison: The Cheapest And Most Expensive Cities Revealed*. [online] Finder UK. Available at: <https://www.finder.com/uk/uber-vs-taxi> [Accessed 7 April 2020].

9. Developer.apple.com. 2020. *Apple Developer Documentation*. [online] Available at: <https://developer.apple.com/documentation/mapkit> [Accessed 27 June 2020].

10. Stripe.com. 2020. *Using iOS Basic Integration | Stripe Payments*. [online] Available at: <https://stripe.com/docs/mobile/ios/basic> [Accessed 27 June 2020].

11. Stripe.com. 2020. *Apple Pay | Stripe Payments*. [online] Available at: <https://stripe.com/docs/apple-pay#create-payment-request> [Accessed 27 June 2020].

12. Back4App Blog. 2020. *Firebase Vs. Parse Server | Back4app Blog*. [online] Available at: <https://blog.back4app.com/firebase-parse/> [Accessed 27 June 2020].

13. (@twostraws), P., 2019. *What Is Swiftui? - A Free Swiftui By Example Tutorial*. [online] Hackingwithswift.com. Available at: <https://www.hackingwithswift.com/quick-start/swiftui/what-is-swiftui> [Accessed 27 July 2020].