CO3016 Computing Project

Dissertation

GreenCar@UoL Carpooling Application

Travis Kirton

BSc Computing

April 2018

Department of Informatics, University of Leicester

Contents

[**Abstract** 6](#_Toc512793723)

[**Introduction** 6](#_Toc512793724)

[Project Background 6](#_Toc512793725)

[Project Focus 6](#_Toc512793726)

[Objectives 6](#_Toc512793727)

[Degree Relevance 7](#_Toc512793728)

[**Literature Review** 7](#_Toc512793729)

[**Theoretical Background** 7](#_Toc512793730)

[HTML5 and CSS3 7](#_Toc512793731)

[TypeScript (Angular2) 8](#_Toc512793732)

[Cordova 9](#_Toc512793733)

[Firebase 9](#_Toc512793734)

[PostGreSQL 10](#_Toc512793735)

[Git & GitHub 10](#_Toc512793736)

[**Requirements & Design** 11](#_Toc512793737)

[Requirements Specification 11](#_Toc512793738)

[Functional Requirements 11](#_Toc512793739)

[Non-Functional Requirements 12](#_Toc512793740)

[Design of User Interface 12](#_Toc512793741)

[System Design 14](#_Toc512793743)

[Application Architecture 14](#_Toc512793744)

[Use Case Diagram 14](#_Toc512793745)

[Activity Flow Diagram 15](#_Toc512793746)

[Component-Based Diagram 15](#_Toc512793747)

[Data Layer Services (Firebase) 17](#_Toc512793748)

[Data Layer Services (NodeJS) 18](#_Toc512793749)

[**Map Routing** 18](#_Toc512793750)

[Dijkstra’s 18](#_Toc512793751)

[A\* (A Star) 21](#_Toc512793752)

[Comparison 23](#_Toc512793753)

[**Implementation & Testing** 25](#_Toc512793754)

[Database Modelling (PostGres & Firebase) 25](#_Toc512793755)

[Rest API Setup 25](#_Toc512793756)

[Users 26](#_Toc512793757)

[Journey Creation 27](#_Toc512793758)

[Matching 29](#_Toc512793759)

[Messaging 32](#_Toc512793760)

[Testing 34](#_Toc512793761)

[REST API 34](#_Toc512793762)

[Ionic2 Application 34](#_Toc512793763)

[**Results & Discussion** 34](#_Toc512793764)

[**Conclusion** 34](#_Toc512793765)

[**Bibliography** 34](#_Toc512793766)

Declaration

**DECLARATION**

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people’s work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people’s work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Travis Kirton

Signed:

Date: 1/04/2018



# **Abstract**

The aims of this project was to build a system to explore the possibilities of a Hybrid Web Application (HWA) based carpooling application which would be aimed initially towards university students. Creation of the system involved building a software system that would allow the user to sign up and create journeys, or be intelligently matched to other journeys based on distance and user-set preferences. The system implements a custom routing engine, using map data and routing algorithms to best route riders to their destination, giving the application increased complexity and a more feature rich experience.

As the application was created as a Hybrid Web Application, it allowed exploration and evaluation of current web-based alternatives, as well as mobile-based alternatives, giving this project a wider scope of users and a bigger scale at which to operate within the future industry of carpool and ride-sharing. Target audiences would be university students, but using technologies such as Firebase, would allow a scalable and affordable solution for large amounts of users & data providing a Backend as a Service (BAAS) which could handle potential growth.

The implementation included services not offered by current alternatives, such as recurring journeys using a repeating journey system, as well as social aspects between passengers such as journey discussions and private messaging. The outcomes of the design, development and implementation concur that it is feasible to develop a HWA for use in such an industry, providing a key area of thought for future applications of a similar vein.

# **Introduction**

## Project Background

“All 13,000 taxis in New York City could be replaced by a fleet of 3,000 ride-sharing cars” [1]. A statement like this was proposed by MIT’s Computer Science & AI Laboratory. This would reduce emissions, passenger wait times & have a better impact on road congestion, all of which are constant issues today.

## Project Focus

For my project, I have researched and developed a carpool application directed at university students, but not limited to them. The projects core idea was giving more control to the user about who they want to ride with. Applications like Uber & Lyft only allow you to set destinations and then you’ll have to deal with whomever the algorithm seats you with, which could purely be based on the areas you’re coming from and going to. A more social and people centric approach would be to allocate riders with like-minded drivers as well as other suitable passengers. This could be completed by smart matching based on preferences, or by giving the user a choice of their own. The application I researched, designed and developed is all about creating a carpool application where the user is in control of the ride.

Current carpool applications simply exist to get riders from point A to B, but completely ignore the possibility of matching people based on personal choice. Daily commuters who work 9 to 5 jobs using the same route each day would absolutely benefit from an application that could set reoccurring bookings with passengers of whom they actually get along with.

## Objectives

The project included multiple core objectives that needed to be met throughout design, development & implementation. Firstly how the user would be interacting with the application and what would the best platform be? Asking this question led to the novel idea of using a Hybrid Web Application allowing access via a mobile deployed application, as well as a hosted web-application. A second core objective was to look at routing. As a carpool application is dependent on journeys, a way to view journeys & their route was required. Gathering map data, analysing it, performing algorithms on it and finally displaying results was a task that required the right tools and techniques to complete.

Design was a key component, as an interactive application requires clear usability and consideration into interaction. Using colour theory & sound Human Computer Interaction Principles, I could create an experience that the user would enjoy and gain satisfaction out of using it.

Evaluation of the applications was completed via usability testing by friends and family. This allowed a real response from individuals who weren’t involved in the design & development, users who might wish to use such an application in the future. This proved highly advantageous and results were positive with minimal complaints.

## Degree Relevance

I chose this project proposal as I found the idea of a carpool application interesting, especially as an avid user of current services such as Uber. As I put more & more time into the application, I realised that it wasn’t just an application to store and display data, but had core fundamentals that needed to be met. This includes algorithm development, database deployment, working with frameworks & libraries all while having to think critically about how to accomplish such a task. Throughout my degree I was taught all of these, and putting them to use with new technologies allowed me to understand quickly what was going on.

Throughout the project I have grown as a developer and gained more skills that I can now use in the workplace. The entire project is now one of the reasons I have secured a graduate position within the UKs leading car manufacturer, as it helped in my application and interview to explain routing, map data and the ideas behind it.

# **Literature Review**

Carpooling is the sharing of car journeys so that more than one person travels in a car, and prevents the need for others to have to drive to a location themselves.

Carpooling as a shared service has typically taken traditional forms of promotion through friends, family and work colleagues. Carpooling itself was on the decline between the 1970s and 2000s [0]. Carpooling decline was high in the 1970s as fewer cars were on the road, meaning more people were inclined to share, but with the boom in car availability, people getting married later which meant more singled on the roads each with their own cars. Cars per household climbed from 1.55 in 1970 to 2.08 in 2013 [0.1]. Decline in and organic carpooling becoming a thing of the past meant new mediums needed to take place.

Ivan Taslimson conceptualised what would become rid-sharing and on-demand online transport when he created a web-based app for college carpooling at University of Washington [0.2]. The App ran on PC and MAC, this work is considered to be the forerunner of carpooling and ride sharing systems, a precursor to technology developed such as Uber, Lyft and the likes.

## User Research and Considerations

The GreenCar application implemented was designed initially to target University Students, though isn’t limited to them. The application could be used by anybody needing to share a journey. Age range considered when developing the application were typically between 18 and 30, however the age range of the application doesn’t have an upper bound associated with it.

Designing for adults takes multiple factors into consideration, such as layout, design style, navigation through the application and even font style and size.

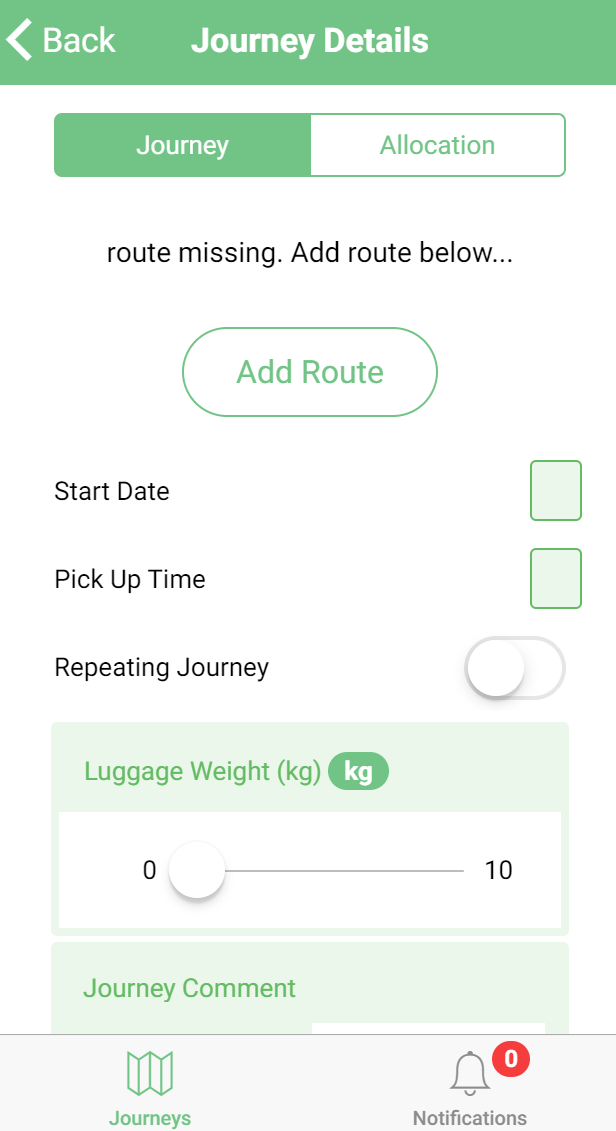
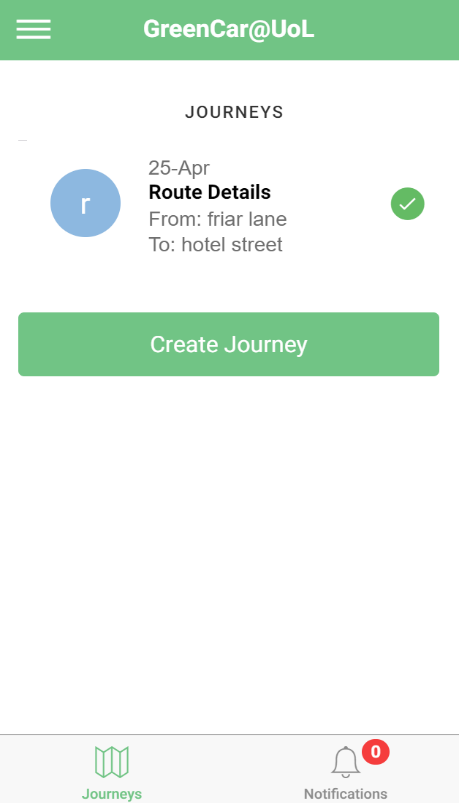
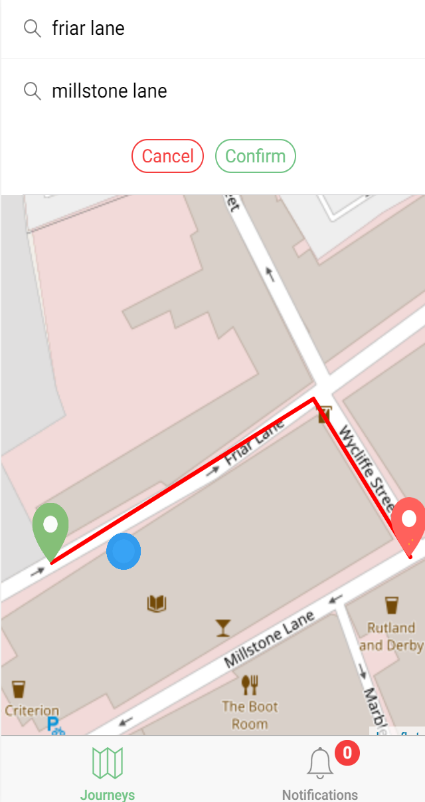
Font Style was chosen by looking at studies on legibility of fonts, a study found that Arial and Courier were considered the most legible fonts and larger font sizes (up to 14) didn’t impact the legibility for the reader. [0.3] The study also found that Arial was read quicker than other fonts, which allowed me to choose this font where possible, but to also realise that the application changes font depending on the device it’s running on, but Apple Fonts are similar to Arial and have careful design considerations.

UI design was taken by looking at a technique called “Designing for All” [0.4], which was brought about during HCI research towards the end of the nineties. Design for All is rooted in three core principles:

1. User centred design placing the user at the centre of interaction
2. Accessibility and assistive technologies for disabled
3. Universal Design for physical products and the built environment

Point two hasn’t been considered during the design of my application, but the choice of colours were primarily solid green on white, which I feel isn’t hard to view and testing on target users hasn’t shown any aversion to this.

User centred design around the user relies on the user understanding what is going on and what their task is at all times. At each page/view of my application I have specified what each view is with easy understanding of the tasks ahead. Labelled headings, clear buttons and highlighted notifications are all examples of this.



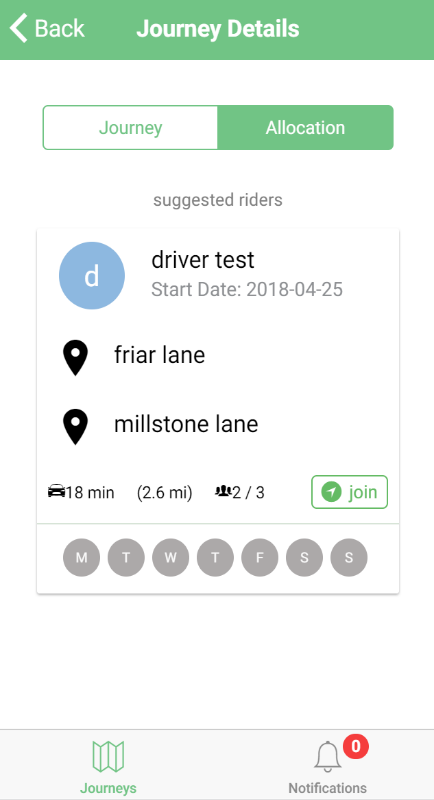
## Review of Current Applications

### Web Based Carpool Applications

Carpool application were as Ivan Taslimson conceptualised, Web-based. Europe has a high popularity of Web Based carpool applications for long-distance journeys. According to published users, German based ‘carpooling.com’ and French-based ‘BlaBlaCar’ had more than 6 million and 10 million respectively [0.5], however carpooling.com was folded into BlaBlaCar in 2015.



BlaBaCar has a fantastic design, with easy to use navigation. You can up as a driver and offer rides, or search a journey to find drivers. The website has a rating system, as well as a ‘per passenger’ price, which is agreed by the driver.

My application uses these basic features, as they are fundamental for a carpool application. Choosing a journey from start to end, then being presented a list of possible drivers with seats available and the pickup time & date

You can see from my design choices that they reflect the fundamentals of information required. BlaBlaCar, however doesn’t offer repeating journeys, only one time journeys. This is great for long-distance trips that would typically cost the driver more in petrol than a daily commute, but not everybody is always taking long-distance trips. GreenCar offering functionality for repeating journeys in the week is a unique aspect not seen in BlaBlaCar.

### Mobile Based Carpool Applications

Mobile based carpool application are extremely popular. Top mobile based applications are Uber & Lyft. UberPool yet again only offers you the option of a single, one-time journey which acts the same as a standard Uber taxi-service yet allows you to join others who are travelling similar directions.

The design of Uber however is very friendly, has a great UI and the map feature works particularly well. Using a Map to choose your start and end destinations was a key focus during development of my application. I wanted users to be able to drop pins on a map and drag them around, much like in Ubers application.

Another consideration that I made during my application that sets it apart from alternatives is the ability to sign up as a driver or rider within the same application. Many mobile based carpool application such as Uber or even Waze require a second application to be downloaded which acts as the driver equivalent. This driver equivalent in Uber does provide enhanced functionality of which I couldn’t match, but if you own a car and want to be both a driver and rider, switching between apps instead of just logging in as your driver account could become a hassle that isn’t needed.

I believe that the merits of both web-based and mobile-based carpool applications are apparent. Web-based is great for a higher level of information when booking journeys, giving it a more community feel, yet mobile-based allows the portability and easier use of functions such as geolocation & map usage. Combining these into a Hybrid Web Application might be the way forward, and GreenCar@UoL could be a shining example of how carpool applications are used in the future, combining all platforms.

# **Theoretical Background**

The main technologies that I used throughout the application include front-end tools such as HTML5 & CSS3 as well as use of the TypeScript language throughout for handling views & controlling data. The application was based on the Ionic2 framework and the Angular2 framework as the core. The styling of the application used SASS instead of CSS which gets compiled into CSS during the build-process regardless. Cordova is used at the core of Ionic, which provides the build-process for the application onto mobile devices while working natively. Firebase was used for storage of data, authenticating users during login & registration and acting as a real-time database. PostGreSQL was the database solution used to import OpenStreetMap data and perform normalisation so it could be used for routing purposes. Git and GitHub were picked as the distributed version control tools, allowing me to manage changes & keep track of history, while giving a good visual view of the project’s progress.

## TypeScript (Angular2)

Angular, is a “TypeScript-based open-source front-end web application platform” [1]. It was led by a team at Google and is also a complete rewrite of AngularJS. Angular has become highly popular as a framework in recent years, as more developers look into fast-prototyping frameworks that allow modularity and control over components of an application. React, developed by Facebook is another similar framework that was built on top of JavaScript, utilising a “declarative of components” [2], much like Angular’s use of “hierarchy of components” [1].

JavaScript & ES6/ES5 doesn’t include a module system, instead using <script> tags to load files with functionality. Script tags require the developer to be very aware of their order and possible errors with some being loaded before others, if they share dependencies between one another. A large application would do well to include encapsulation data, ensuring that information isn’t overwritten.

1. var app = angular.module
2. ('app', [newsFeedService])

AngularJS introduced a module system, which allowed parameters to be passed into the initial application call. These parameters would be the dependencies (or array of dependencies). AngularJS also provided a $scope object, which would bind between a view and controller within an MVC application, however Angular2/5 has completely changed this, with the following core differences implemented.

Angular doesn’t have a concept of “scope” or controllers, instead using a hierarchy of components as its primary architectural characteristic. $scope and controllers were replaced with components, which is a directive with a template. See Figure 1 [1] for a view of the Architecture Angular is using.

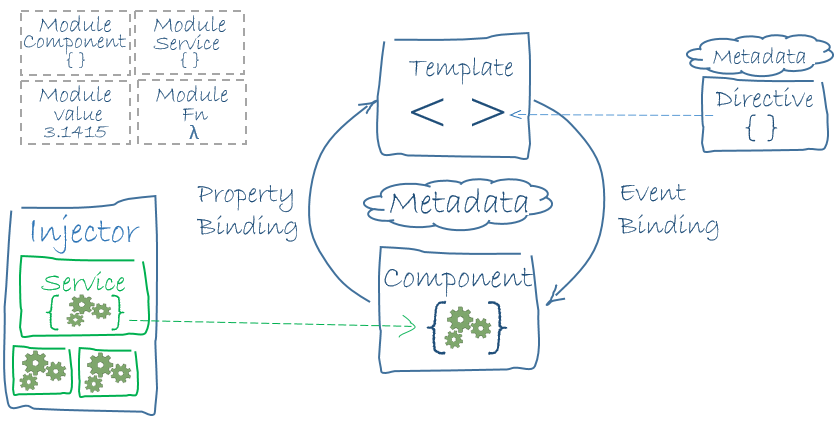


Figure 1 – Angular2 Component Architecture

## 

## Cordova

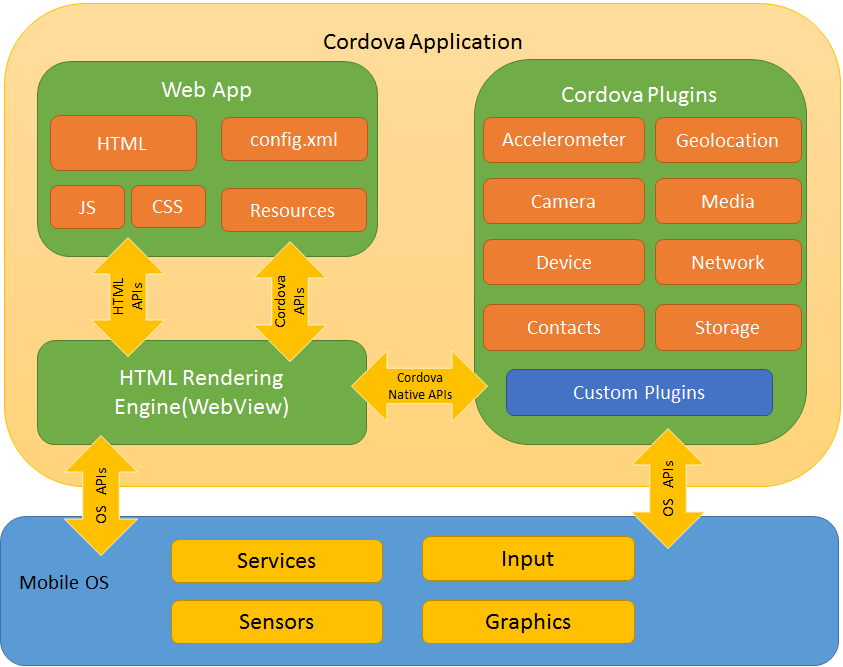
Cordova is a “mobile application development framework” [3]. Developed in 2011, later rebranded to PhoneGap and released as an open source software named Apacha Cordova. Apache Cordova allows mobile application to be built using Web languages such as CSS3, HTML5 and JavaScript (now TypeScript) instead of having to rely on platform specific languages such as Java for Android, or Swift/Small-Talk for IOS. The result of this is a Hybrid Application, which means it’s never a truly native mobile application.

Figure 2- PhoneGap Architecture

Ionic2 is built on top of Cordova and Angular2. Ionic & later versions provides an easy framework and platform to build mobile applications using easy to use web technologies. Ionic also includes a large library of UI components that can be used to help with applications running on mobile, for example navigation bars, notification systems and mobile responsive layouts.

## Ionic

Ionic is a complete open-source SDK for hybrid mobile app development [17]. Ionic was originally released and built on-top of AngularJS and Apache Cordova, but more recent releases such as Ionic 2 and Ionic 3 (commonly known as Ionic) are built on Angular. Ionic provides tools for developing hybrid applications that use Web technologies such as HML5 & CSS3, along with SASS. Ionic allows apps to be developed with these technologies, then built and deployed onto the relevant native app-stores.

Ionic provides custom components for interacting within native devices, such as repeating collections, allowing the user to scroll through thousands of items without the user experiencing any performance hits.

## Firebase

Firebase is a real-time database service that allows you to synchronise data between multiple devices at the same time It provides a core module which allows you to expose the database API and call authentication methods where needed. This vanilla firebase module is useful for authentication, and using HTTP PUT & GET requests, you can push and fetch data with ease, though calls are asynchronous and needed to be handled as such.

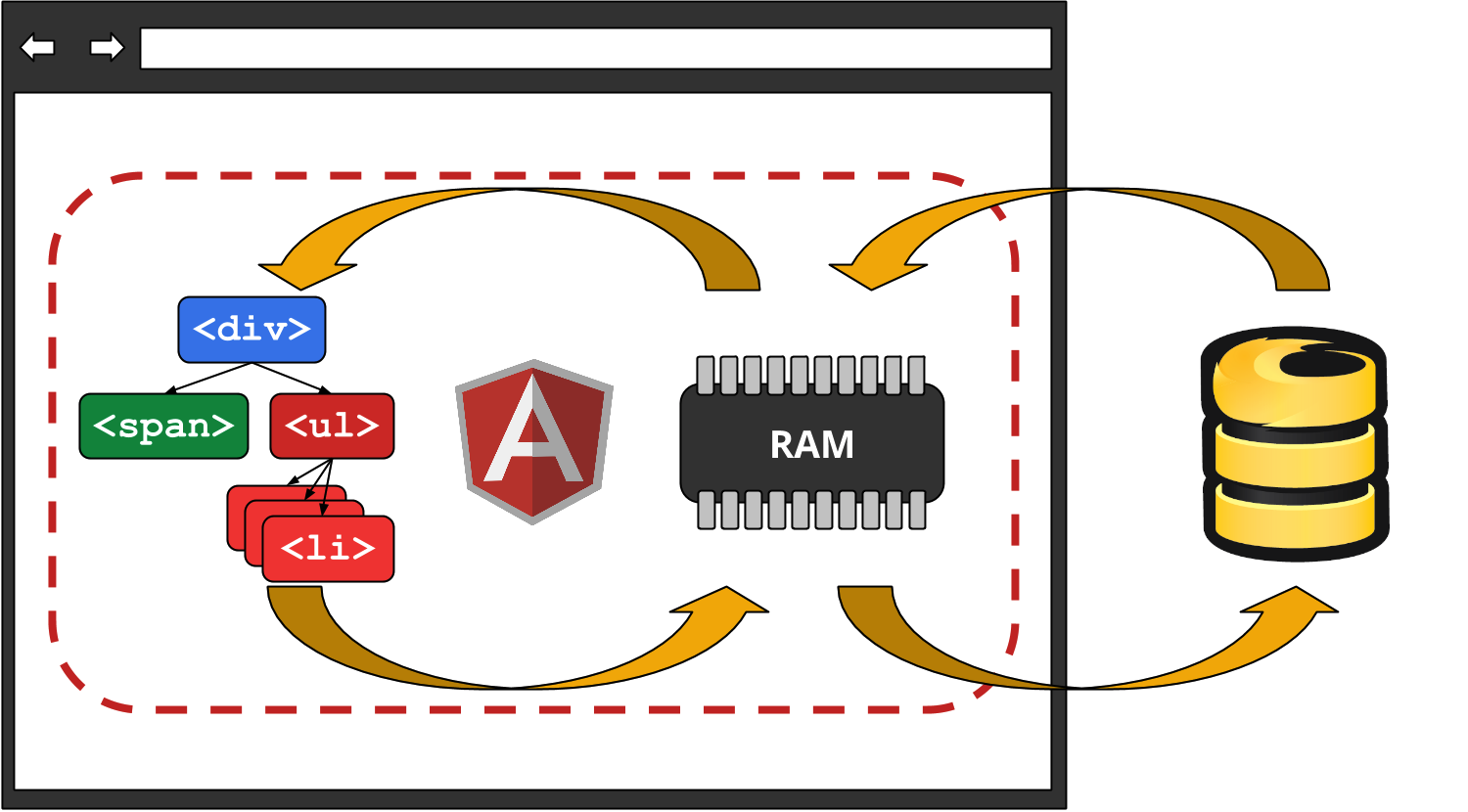
There is also a module which can be installed via Node Package Manager called AngularFire2. This wraps the vanilla firebase methods and provides easier use of Observable calls to Firebase. An Observer patten allows you to bind Objects, Arrays and any number of variables to a reference in the database, and when changes within the database happen, the observable will react to the changes & update your local data accordingly. Having real-time updated data without having to refresh the application is extremely useful and improves the entire user-experience. This is also called Three-way data binding Firebase is also a NoSQL database, storing data in a JSON format, which allows developers to store JavaScript/TypeScript Objects extremely easily.

Figure 3 - Three-way data binding [4]

## PostGreSQL

PostGreSQL is an “object-relational database management system” [5]. This database storage solution was picked by myself because it allowed integration with tools that deal with map data. OSM2PGROUTING was my map tool of choice, which allows importing ‘.osm’ files into a PostGreSQL database with ease. This extracts nodes and edges from the OpenStreetMap data, while also splitting it up into multiple tables that can be queries easily.

PostGreSQL is also supported by NodeJS, which I used to create a RESTful API to query the map data. NodeJS was quick & easy to setup with this database choice, allowing multiple requests & implementation of Request Pools to stop concurrency issues if requesting from multiple devices (which is what the application would be doing).

## Git & GitHub

Git is an open source distributed version control system that is used to primarily keep track of changes during an applications development. It was originally developed to manage Linux Kernal development, but then became a mainstream version control system.

GitHub is an open source hosting application for Git repositories. It includes many features to support the development of applications, especially across teams.

# **Requirements & Design**

The initial application proposal was to have a web-based carpool application with an additional mobile application if time permitted, however due to choosing Ionic2 as a framework, this has allowed the design and development of an application which works as both. I was not restricted to a particular platform, which allowed more time to focus on design and functionality over what each platform could handle.

The application implements many of the core requirements outlined below. The general idea is that a user can register and login to the application as a rider or driver, afterwards creating a journey and specifying certain information. If a driver has created a journey and then a rider creates a journey with similar journey points, they will be matched together and placed into a shared journey. Once grouped, they can chat in a shared discussion, or message members individually. A user can create multiple journeys, join multiple rides & set their preferences as desired. Data is in real-time, thus the application does require an internet connection. This was considered as a negative, but with a wide amount of Wi-Fi hotspots, mobile data & connectivity in areas that users would be commuting to/from it wasn’t taken into account, but discussions will follow.

## Requirements Specification

The following are functional and Non-functional requirements that were taken into consideration during development

## Functional Requirements

*Users*

* A user can be a driver, rider or admin
* When a user account is created, they should specify core journey preferences, and optional journey preferences (preferences change depending on driver/rider)
* An administrator may ban users if they are deemed not to be trusted
* Each user can create one or more journeys, each indicating origin/destination and modality (driver/rider), as well as times of journey
* A user may cancel an arranged journey at any point, as long as the cancellation is 48 hours prior to scheduled journey
* Users can discuss the journey amongst each-other
* Users can message each journey member individually

*System*

* A journey is automatically associated with a preference route (from origin to destination), and is computed by most efficient route. Showing duration, cost & a direction breakdown
* Once a journey is created, the system will suggest a driver allocation based on core preferences and journey being taken
* The system must support automatic conversion between miles & kilometres based on user preferences
* The System could allow notifications to users, for journey allocation

## Non-Functional Requirements

*Environment*

* The user interface will be a Progressive Web Application, allowing use on browsers and natively on mobile devices with support for:
  + IOS 8+
  + Windows 10 Universal App
  + Android 4.4+
  + Google Chrome (recommended)
* Application views will include managing user accounts, journey creation/requests and messaging
* Technology to be used includes Angular2 as the main language, using Ionic2 as the native mobile wrapper
* The internal communication will between components will be handled by Firebase, allowing the application to have a Serverless architecture using a BAAS
* Data collected from maps for routing & displaying to user will be taken from OpenStreetMap

## Design of User Interface

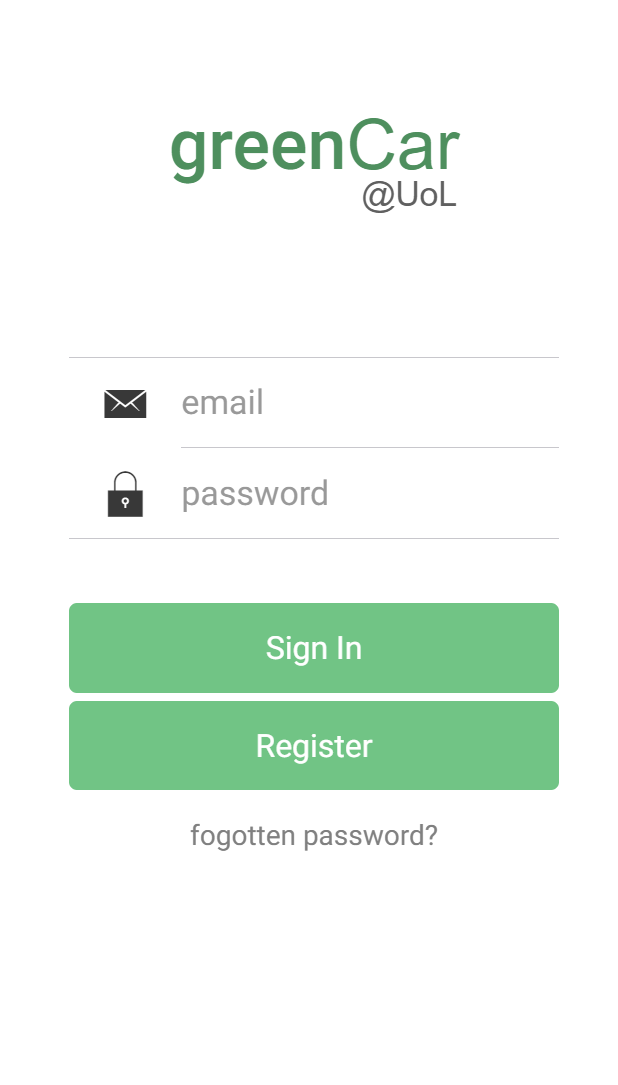
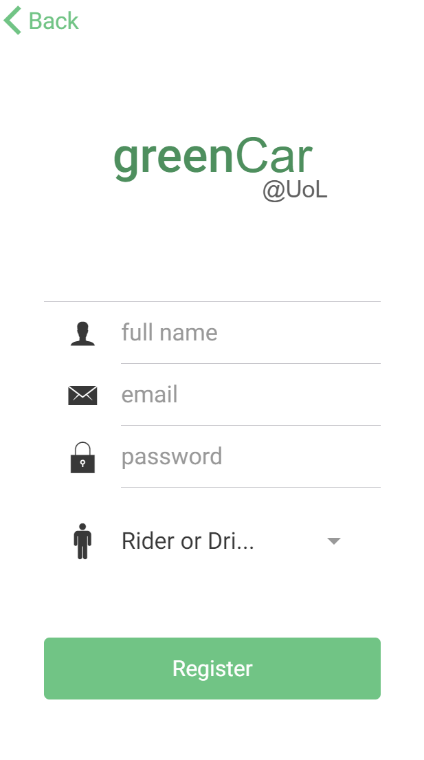
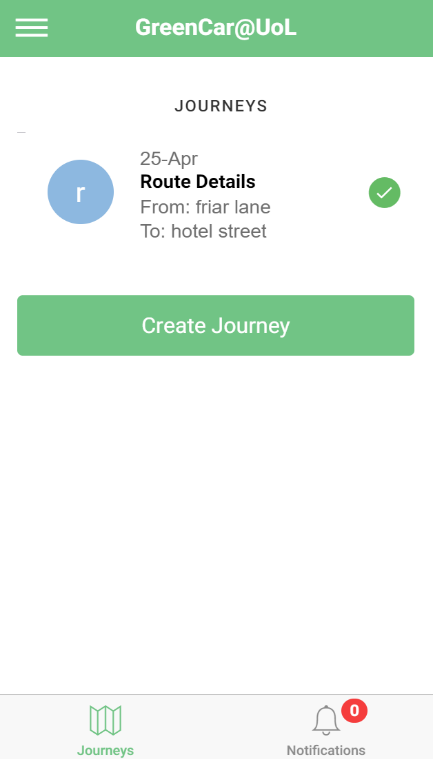
The design of the application is extremely important. Mobile based applications cannot be cluttered, as the screen real-estate is small. Managing information, flow between screens and how the user interacts with the application are what makes a user satisfied enough to keep using it.

I followed core principles of Human Computer Interaction, such Norman’s 7 Principles [references]. These include principles such as:

* Simplify the structure of tasks
* Make things right
* Get the mappings right

I wanted the design to be simple, yet usable with enough information for the user to be satisfied and not wonder where something is. There are only 2 tabs, one for journeys and one for notifications. Only two tabs were needed because there are additional pages accessible from either the slide-menu, or when creating journey and talking to users. All of which you can press a tab or back button from.

For the applications colour scheme I wanted to emphasise the awareness of green. Green works well against a white background, which worked especially well when needing to use other variations of colour on top of the white background. This allowed the green to be a natural colour theme throughout, but doesn’t bombard you, while other nuances such as journey status and user profile icons can be coloured differently.

 ­

## 

The above shows a selection of views a user would see within the application. Once logged in the user is presented with the journey page, giving access to the journey creation page, which includes functionality to add a route and once the journey is added they can navigate to the allocation page to see any matched routes. There are multiple design choices I felt were needed to enhance the user experience of the application, such as including a geo-location marker based on the users position while choosing a route, or seeing green highlighted days for a matched journey to show it would be repeating on those particular days of the week.

Ionic2 allowed me to have great design control, utilising multiple core components, such as toggle buttons and sliders which really make the application feel more intuitive on a mobile device. I design parts that would populate with more data, such as multiple journeys and comments by using Ionic list components, giving amazing control over the content inside them.

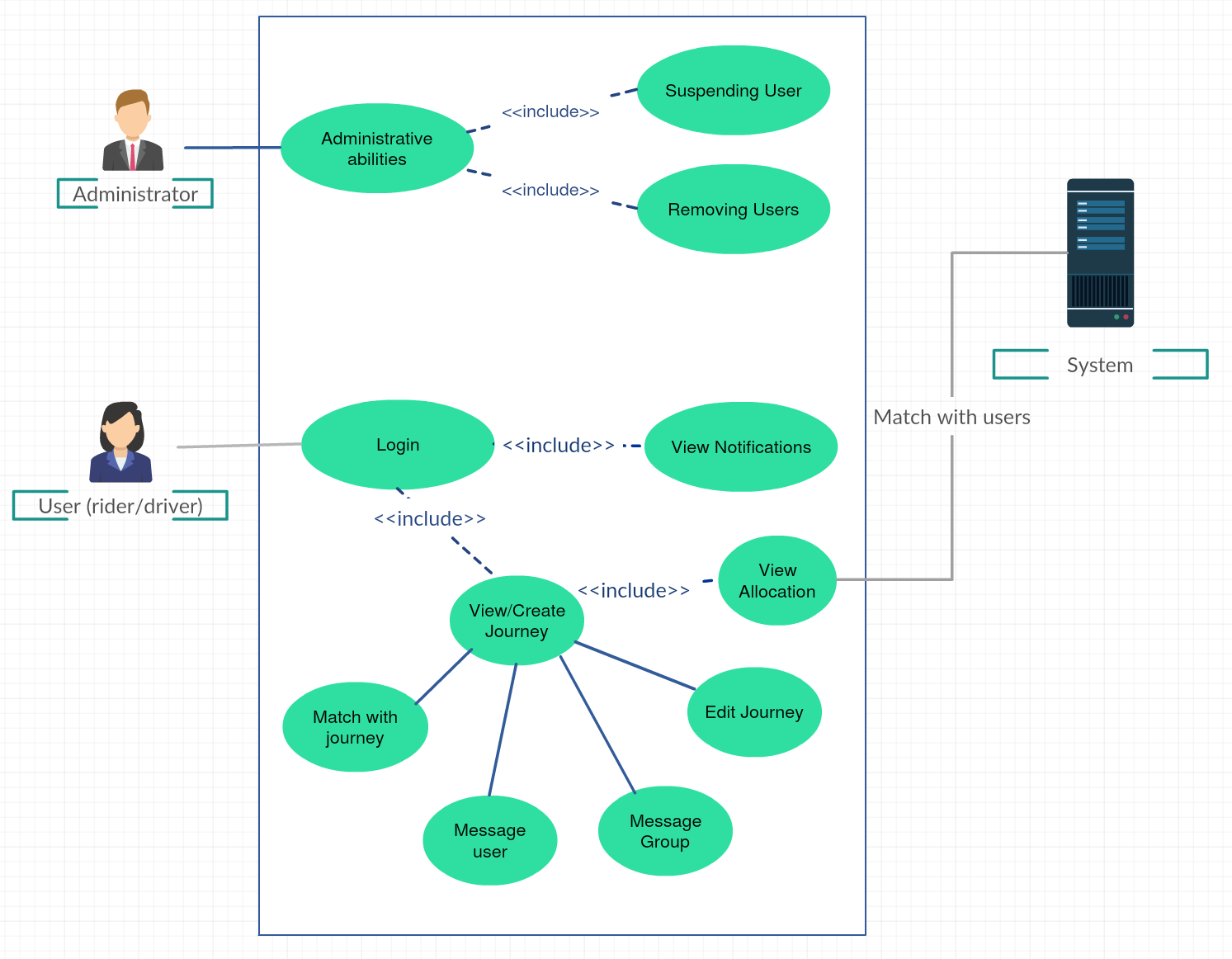
## System Design

### Application Architecture

The application architecture is based on Angular2 and its use of components. Each Angular application has a root component, which connects to a component hierarchy with a page DOM. Each component defines a class that contains the actual application data & logic, which is then associated with an HTML template that defines a view to be displayed in the target environment. This allows different components and their associated templates & data to essentially be ‘pushed’ or ‘popped’ from the application stack.

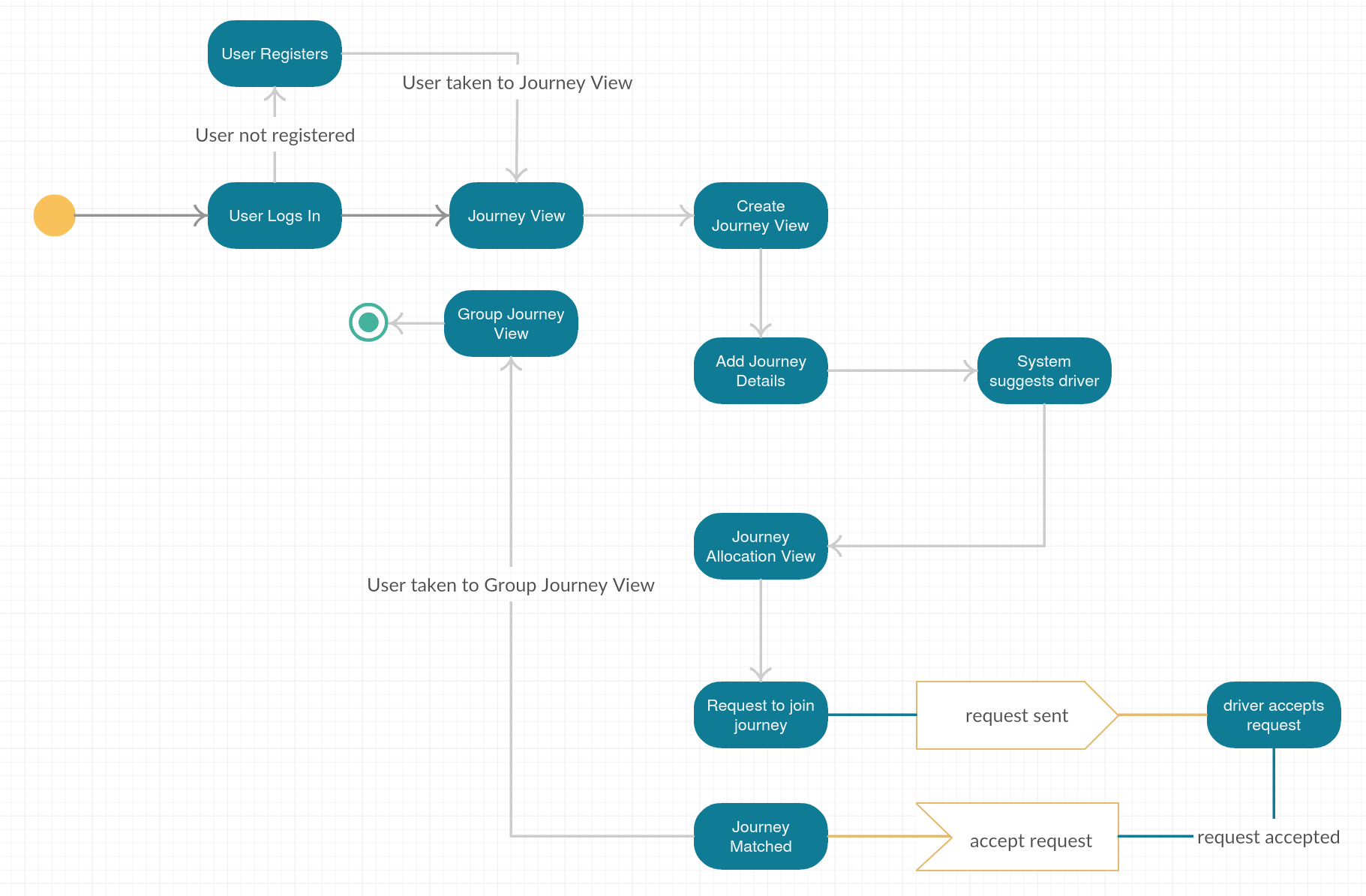
The architecture of Angular systems also take into account services and dependency injection, which is data & logic that isn’t associated with a specific view. Services can be shared across multiple components and require the *@Injectable* decorator. Using this decorator allow your services to be injected into other client components as a dependency. I chose to use services for multiple tasks such as fetching and pushing data to Firebase, or retrieving map data from PostGreSQL, which allows my other components to be lean and focus on handling logic that effects what the user sees in the view based on their actions.

### Use Case Diagram

A use case diagram was created to give a brief high-level view of the system and how a user could interact with it, providing a concrete view for some of the core functional requirements outlined above.

### Activity Flow Diagram

An Activity Diagram allowed me to capture some of the decisions and workflows that presented themselves while building on the use-case diagram. It also provided a good logical layout when implementing certain features of the application. I created a single activity diagram for creating a journey and joining a driver journey that would previously have been created.



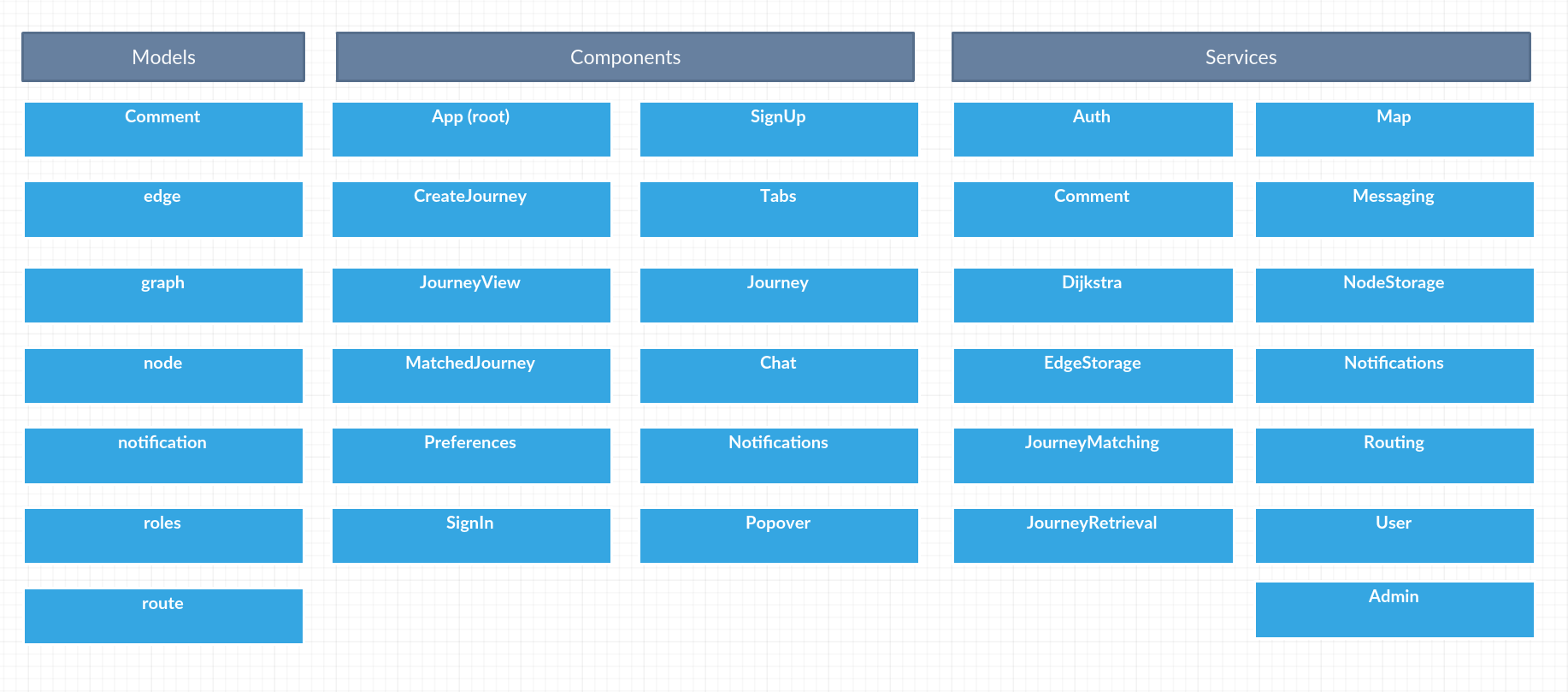
### Component-Based Diagram

Because of how Angular uses a component-based architecture, I created what I thought would be the required components, including page-components which include a view and data/logic file, as well as injectable service components used for logic processing and database operations.

I have designed the diagram to showcase how each component interacts with one-another, showing the hierarchy and injection of services into each. There are also model-based components that don’t require a template associated with them. Model based components are lean and used for storing custom objects used throughout the application, such as Routes, Comments and Notifications amongst others.

In total there are:

* Model: *7 Components*
* Page: *13 Components*
* Services: *12 Components*

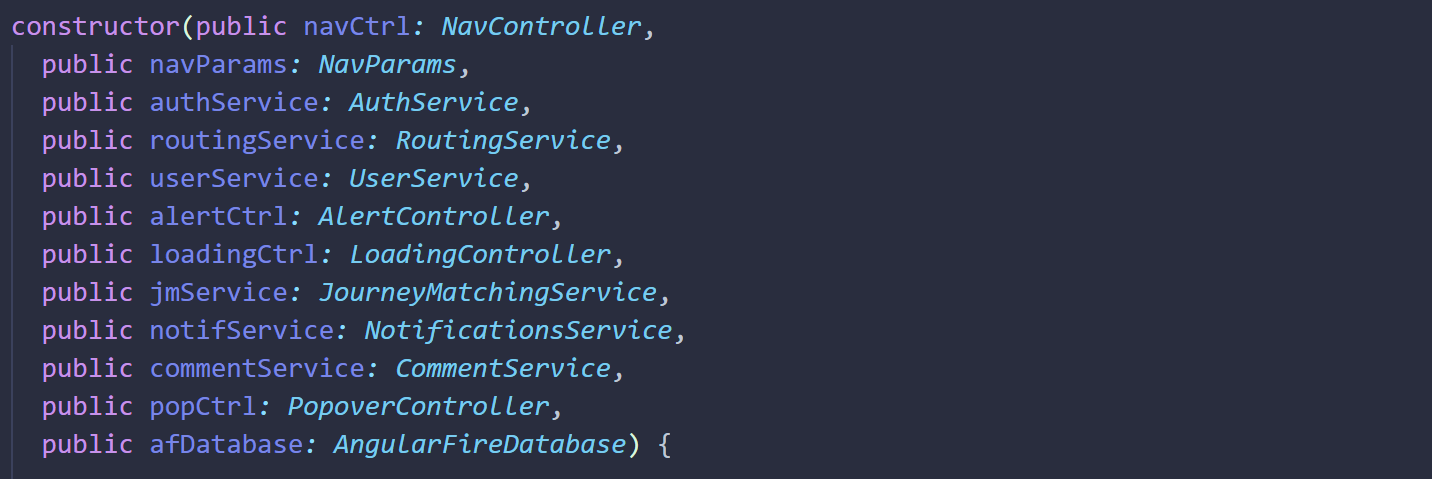


Each model contains minimal data and possible Getters and Setters. The components contain a TypeScript file which deals with the data & logic of the component, as well as a Template HTML file which is what the user would see as the view for the component. Each component also includes a SASS file to handle styling. Services are single Typescript files acting as @Injectable components, which can be imported and injected into other components using *import* statements along with specifying the Service inside the constructor.

To give an example of how components connect together, one of the larger components was JourneyView, which dealt with data during journey creation, showing suggested driver routes once allocated by the JourneyMatching Service, along with handling Model data. The import section can grow and grow if you require access to multiple other components.

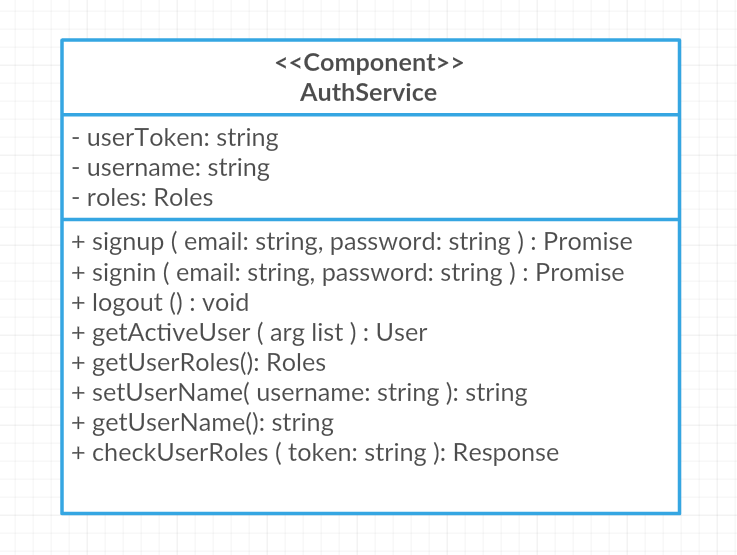


Most components are then instantiated within the constructor, so that you can then call the instance of them throughout the rest of the components.



### Data Layer Services (Firebase)

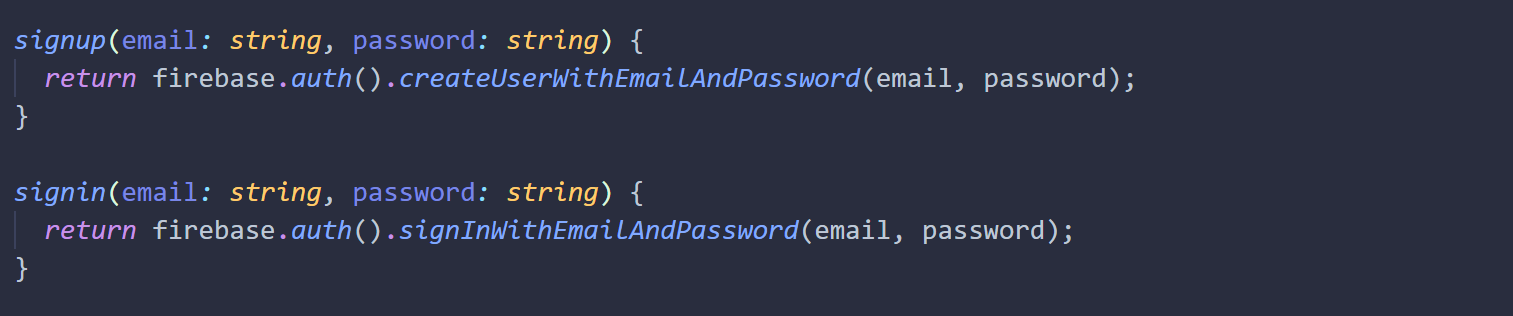
Firebase is used throughout the application as a way to perform Create, Read, Update and Delete (CRUD) actions. These include tasks such as registering new users, which requires pushing registration form to Firebase and checking for success before granting a user access. Services that talk to Firebase are required for pushing new Journeys, Sending request and responses between users and also either commenting on journeys or messaging between users.

The following are component diagrams to show services and their functions to work with Firebase.

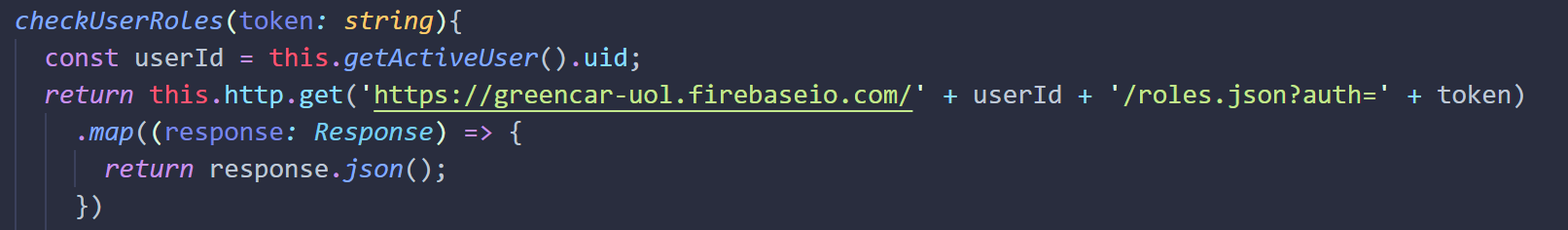
The Auth Service interacts with Firebase through *checkUserRoles (), signup () & signin*

They call Firebase via either importing Firebase as a component or calling its *auth ()* to create or authenticate users, or by using HTTP GET requests to return Firebase JSON data object responses.

Firebase auth () calls can be seen here



CheckUserRoles can be seen in Action here, using HTTP GET calls.

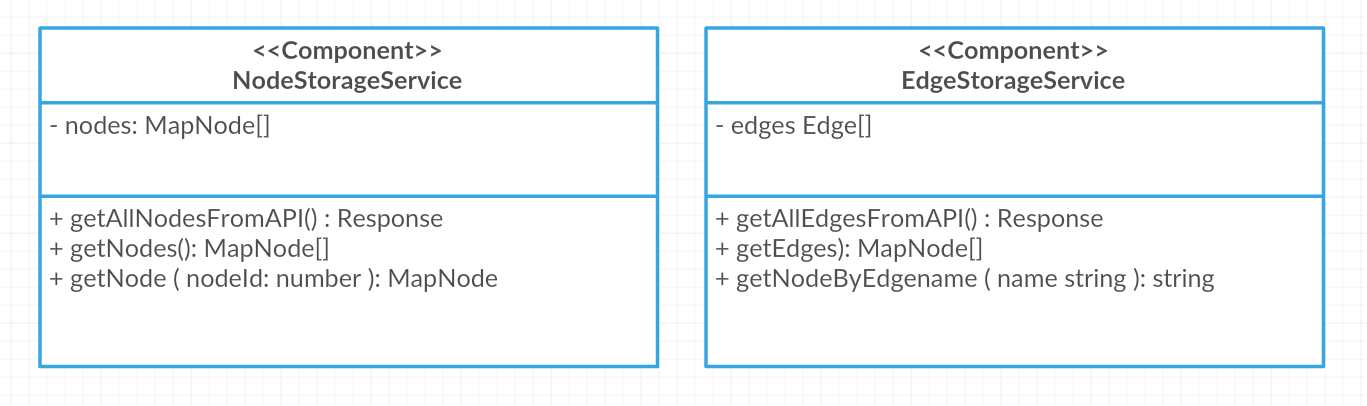


Components can also use AngularFire2, which is a package that gives easier use of Firebase via wrapping complex methods in easy to use and read wrapper methods, which can return Observables. AngularFire2 with Observable responses is great as it allows the application to ‘listen’ for changes and change data in real-time to match the real-time database.

### Data Layer Services (NodeJS)

The application uses a NodeJS RESTful API running which allows easy queries and JSON responses from the PostGreSQL database storing map data. There are two services which interact with this API, returning ALL nodes and edges, allowing quick calculations.

Initially the routing algorithm which requires these nodes & edges was calling the API on separate requests for each Node and Edge. Calling separately is time-consuming as the database contains approximately 28,000 of each, on top of which NodeJS uses Asynchronous calls so waiting for one call to finish before the next was defeating the idea of asynchronistic calls.



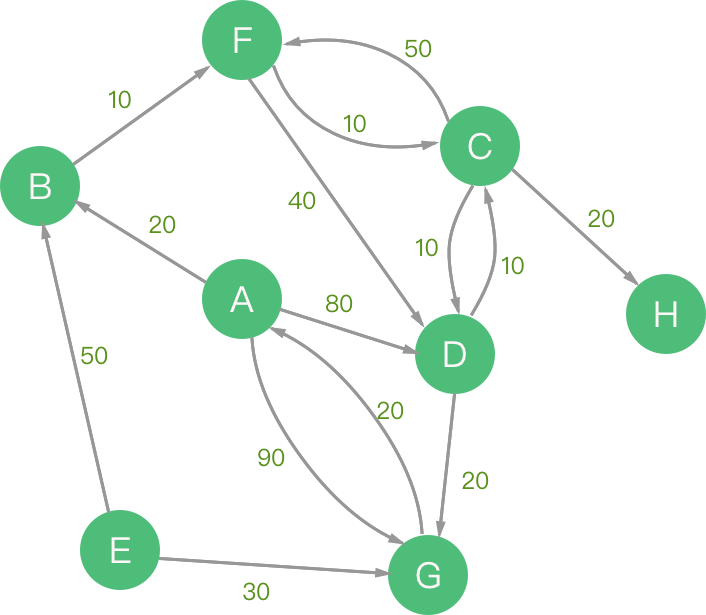
# **Map Routing**

The application has a handful of core functionality of which requires use of specific algorithms. Firstly, the application computes routes from a pair of latitude & longitude coordinates. Given these coordinates and a database populated with OpenStreetMap data imported in such a way that can be queried like a graph, the system would return the shortest path from the first pair of coordinates to the second. This acts as a source and destination situation, which given a resulting array of coordinates containing the shortest path can then be overlaid onto a visual map using tools such as LeafletJS.

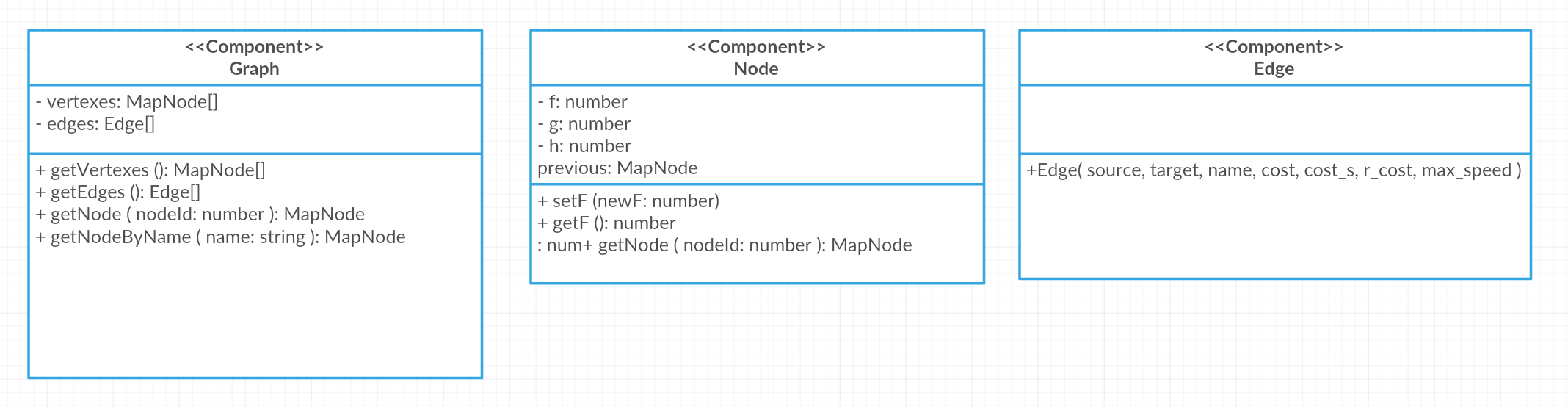
## Dijkstra’s

Dijkstra’s algorithm is “an algorithm for finding the shortest paths between nodes in a graph” [6]. This graph could represent a road network, network of routers or positions within a video game. The algorithm was conceived by computer scientist Edsger W. Dijkstra in 1956, being published three years later.

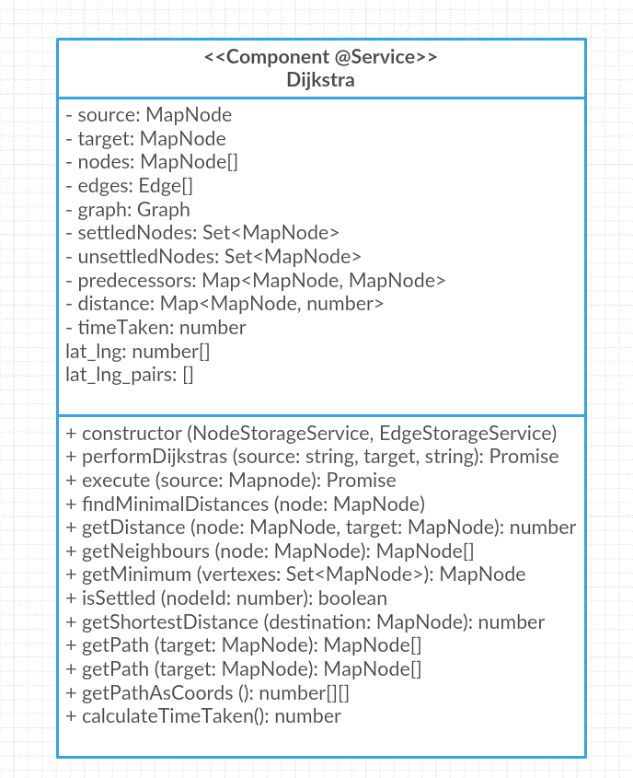
The original algorithm found the shortest path between two nodes [7]. The algorithm works as follows [8]:

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes
2. Mark all nodes unvisited. Set the initial node as current. Create a set of unvisited nodes called the unvisited set consisting of all the nodes
3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbour B has length 2, then the distance B (through A) will be 6 + 2 = 8.
4. When we are done considering all of the neighbours of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked visited or if the smallest tentative distance among the nodes in the unvisited set is infinity, then stop. The algorithm has finished.
6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new “current node” then go back to step 3.

I was able to work with three models, which handled the Graph, Edges and Nodes. Nodes and Edges were populated by using the EdgeStorageService and NodeStorageService which interfaced with a RESTful NodeJS API.



I chose to handle the implementation of Dijkstra’s using a Service, as it could be imported into components that would require the returned coordinates, such as a Map Service. The following was the component Implementation for Dijkstra’s. It would take 2 road names as string values, fetch their appropriate NodeIDs using the graph implementation and start performing the algorithm.

It is *execute ()* which is of most important, and calls the other methods given a source and target node. The algorithm will find the shortest path from the source to all other nodes within the graph however, the algorithm will stop once it has visited and settled the target node. This method was designed as follows, with some checks for time performance included.

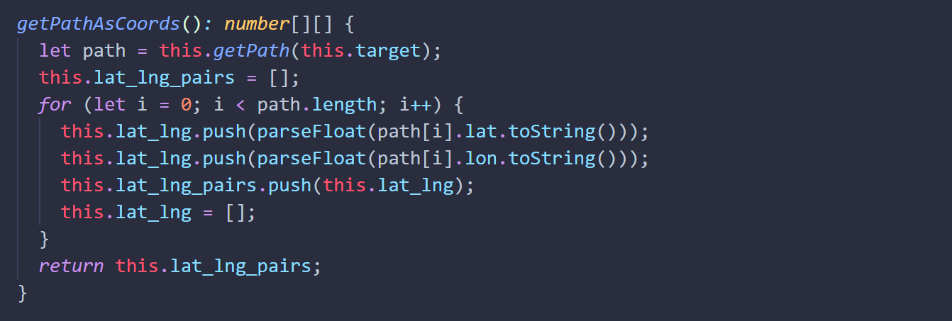


You can see how it runs while the unsettledNodes array isn’t empty (unless the target is found, at which point a *break* is called).

For each node that is visited and settled a predecessor is set which allows a path from the target node back to the source node is available. This is in-fact how you can traverse the predecessors in reverse to build-up a two-dimensional array of latitude & longitude pairs. The following code-snippet implements this feature.



The target node is passed in once Dijkstra’s has completed, then the algorithm gets the predecessor of the target node and pushing this to an array of MapNodes called *path*. The algorithm will run until there no more predecessors (the source will be the end). Once completed, the array of MapNodes is reverse to be in the correct order.

This only retrieves an array of MapNodes, but what we want is an array of coordinates. Luckily, each MapNode contains a latitude & longitude associated with it. We can then call our second function to loop through our returned *path* array and push each MapNodes coordinates to a second two-dimensional array.

## A\* (A Star)

A second and very powerful routing algorithm is called A\* (pronounced “A star”) and is widely used within pathfinding and graph traversal. It follows similarities of Dijkstra’s, being given a source and target node, then traversing a graph to find the shortest path between the two. It can be a lot more accurate in some cases, but is often outperformed by algorithms who use a pre-processed graph.

A\* is an informed search algorithm (or best-first-search), which means it will search amongst all possible nodes, but pick the one that it ‘thinks’ will lead most quickly to the solutions.

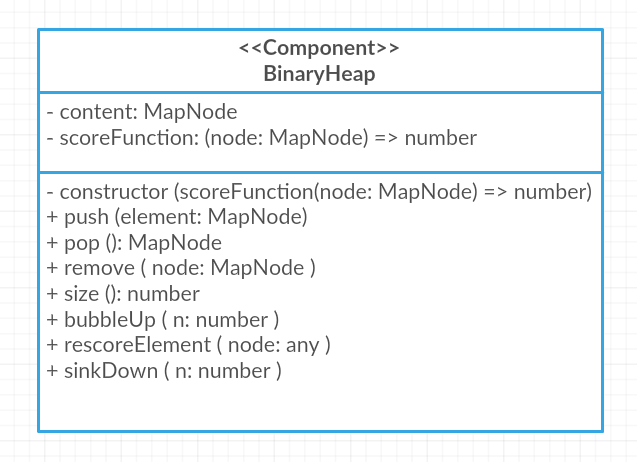
At each iteration of the main loop, A\* needs to determine which of the partial paths to expand into one or more longer paths. This is computed on an estimate of the cost (weighting) still to go to the goal node, which is calculated using the following equations

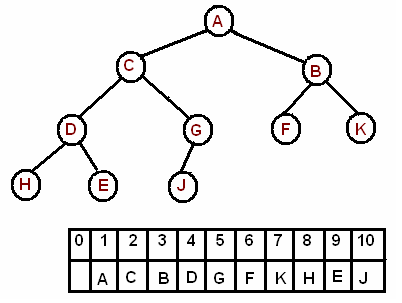
Where *n* is the last node on the path, *g(n)* is the cost of the path from the start node to *n* and *h(n)* is the heuristic, which is estimated as the cost of the cheapest path from *n* to the goal.

As the typical implementation of A\* uses a priority queue, I chose to implement a binary heap data-structure, which provide a more efficient way of adding values, over an ordinary Array.

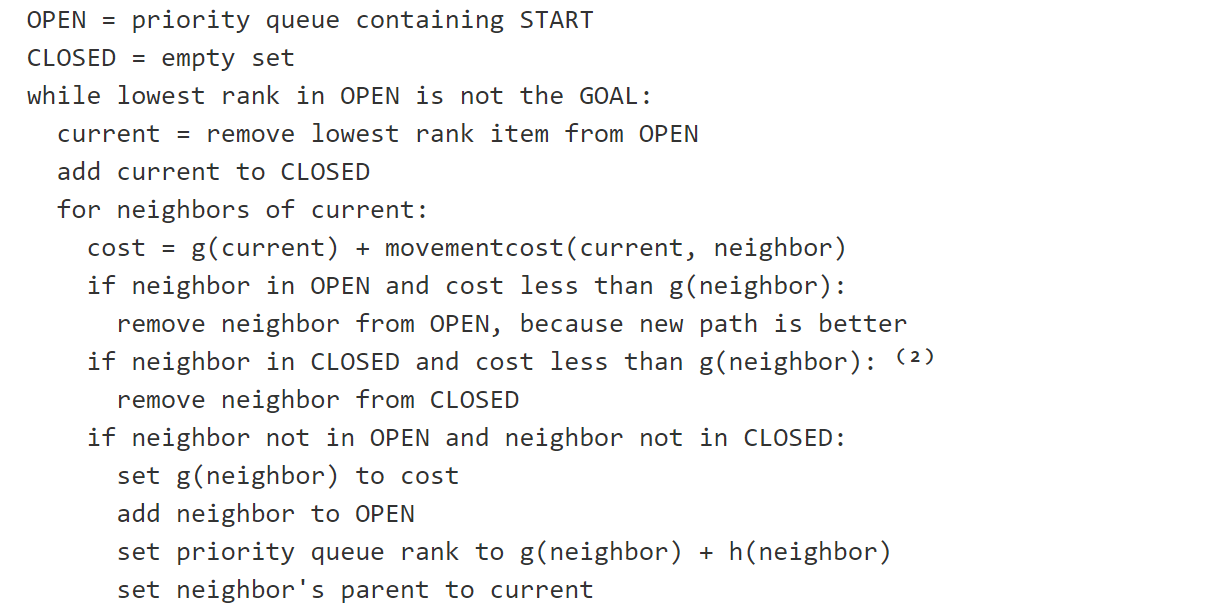
|  |  |  |
| --- | --- | --- |
| Algorithm | Average | Worst Case |
| Space | O(n) | O(n) |
| Search | O(n) | O(n) |
| Insert | O(1) | O(log n) |
| Delete | O(log n) | O(log n) |
| Peek | O(1) | O(1) |

I was able to effectively implement a Binary Heap data-structure in TypeScript, as TypeScript doesn’t include one. It wasn’t hard to follow an implementation in JavaScript & convert it into TypeScript using [10]

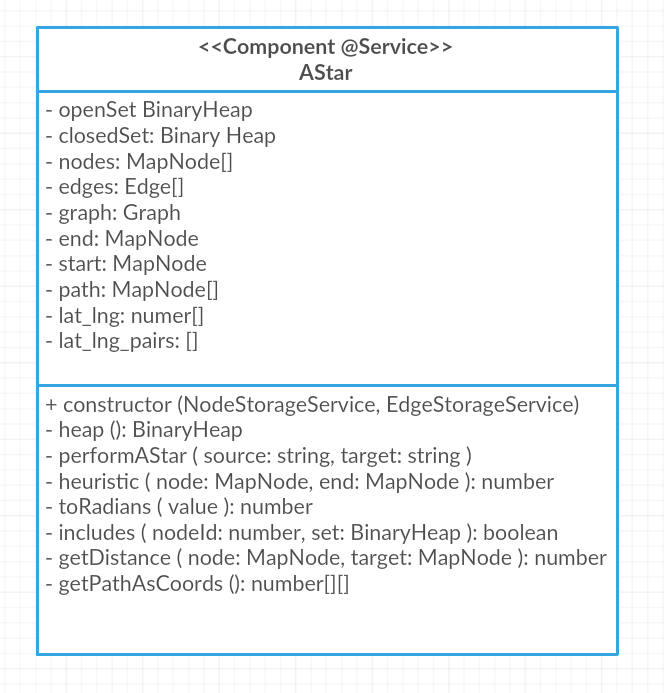
The following is the model component diagram for my Binary Heap Implementation.

The idea of a Binary Heap, is using a partially sorted array, much like a tree structure. The highest (or lowest) priority element is always stored at the root, hence the name “heap”. A heap with N nodes will always have *O(log N)* height. A Binary heap can be implemented using it’s levels in an array, the root being the second item in the array. With the following example we can see how this would be implemented. Left children = 2 \* k, Right children = 2\*k+1, Parent = k / 2 index. [11]

A\* can be explained with the following pseudo-code [9]



My implementation followed a similar step, and much like the Dijkstra’s implementation, I used a @Injectable Service Component which took a source and target string and again converted them into their respective NodeIDs using the Graph implementation.



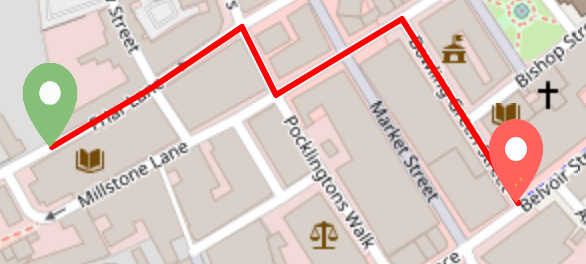
## Comparison

Both pathfinding algorithms are high performing in certain conditions, and can have increased optimisation via techniques such as pre-processing, faster data-structures and more professional implementations. From my own implementations of each algorithm and routing them both between the same roads within Leicester, I found the following that with the following tests Dijkstra performed well under short distances most of the time, however A\* did perform better on slightly longer journeys

Test 1

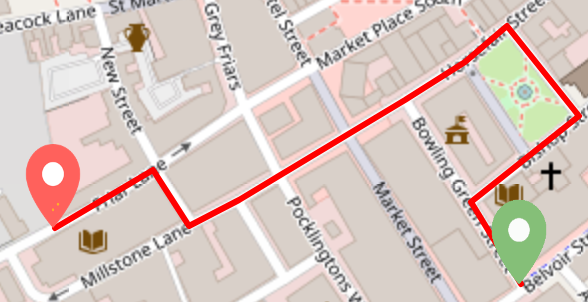
From: Friar Lane *to* Bowling Green Street (distance of 0.2 miles)

**Dijkstra** Completed in: 402.30000007431954ms



**Notes:** The path taken differs slightly, with A\* taking a seemingly longer path, which may reason to why the time taken to complete are 130ms slower

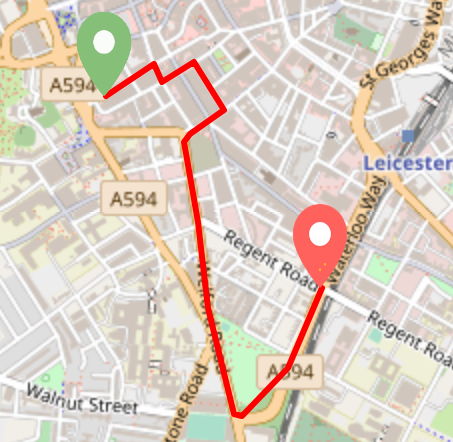
**A\*** Completed in: 532.0999999530613ms



Test 2

From: Friar Lane *to* Regent Road (distance of 1.3 miles)

**Dijkstra** Completed in: 3023.799999966286ms

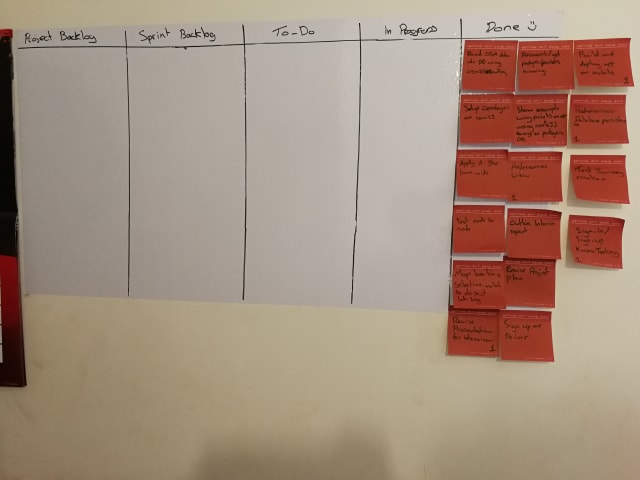


**Notes:** The difference in Paths is significant, with Dijkstra’s looking quicker by eye, yet A\* computed faster. The path A\* took may not be the most efficient however when actually driving it.

**A\*** Completed in: 940.9999999916181ms



# **Implementation & Testing**

All implementation was completed by breaking requirements into atomic tasks that could be accomplished outside of any other task. Having atomic tasks meant I could set-up a Kanban board to fulfil each task. A Kanban board is a physical representation of tasks, typically using card on a board sectioned into software life-cycle stages. I used it for each task, assigning it onto a Kanban implementation in my living residence. Using a physical Kanban board proved extremely helpful to visualise task completion and outline what else needs to be done. Though a Kanban board is better in teams, and studies have shown 2x improvement using them [13], I also found it to increase my productivity.

## Database Modelling (PostGres & Firebase)

The application used PostGreSQL & Firebase for data-storage. PostGreSQL was specifically used for storing OpenStreetMap data, which was imported using osm2pgrouting (12). Given an .osm file obtained from sites such <https://www.openstreetmap.org/export>

I could then create my PostGreSQL database via the command line:

1. sudo -u postgres psql (windows: psql -U postgres -h localhost –W)
2. CREATE DATABASE greencar\_uol;
3. \c greencar\_uol #connecting to DB
4. CREATE EXTENSION postgis;
5. CREATE EXTENSION pgrouting

Once the database is setup and osm2pgrouting is installed, I could then import the .osm file using the following command:

osm2pgrouting -f leicester.osm -c mapconfig\_for\_cars.xml -d greencar\_uol -U postgres -W password

This import created multiple tables within my created database, however it is **ways** & **ways\_vertices\_pgr** that contain the nodes and edges used by the application. **Ways** contains the edges and **Ways**\_**vertices\_pgr** contains the nodes.

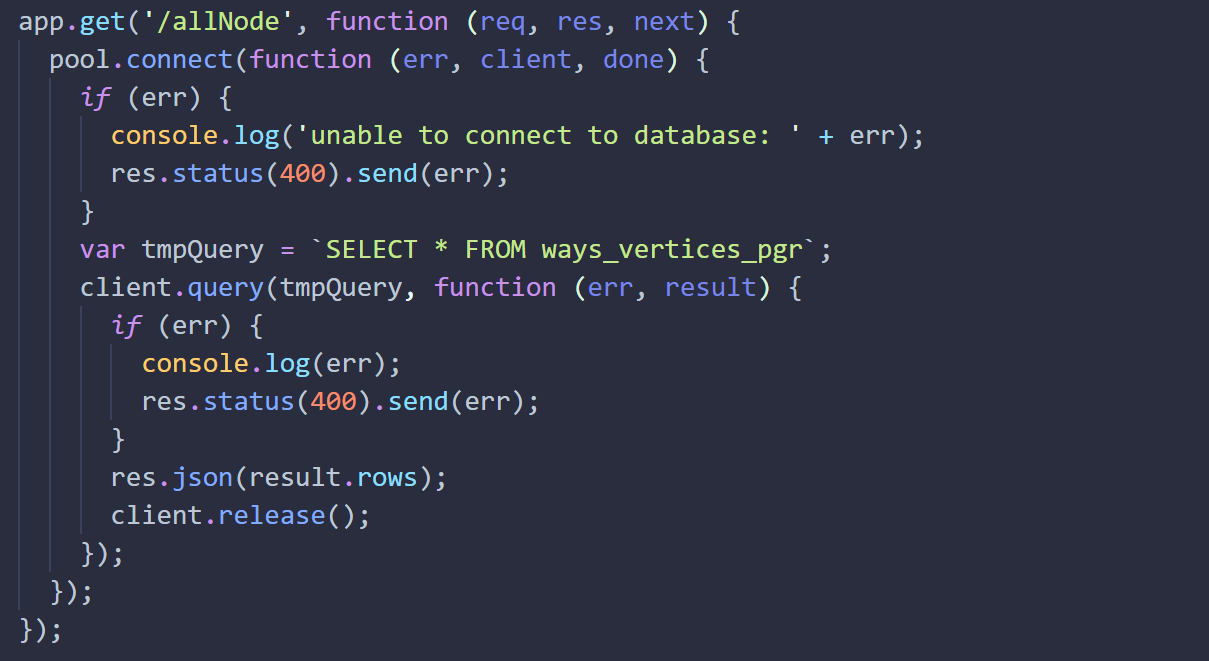
I took into account that the user could type in a mixture of lowercase and uppercase letters, so decided it best to update the generated WAYS table, setting all Road Names to lowercase. This would allow me to convert all road names inputted by the user to lowercase before calling the database, ensuring no case issues occur.

UPDATE WAYS SET name=lower(name)

## Rest API Setup

For the RESTful API I decided that NodeJS would be efficient enough to query my running DB. NodeJS runs using JavaScript and including some extra libraries allows it to be a lean and fast API. Express was used with NodeJS as it provides many HTTP utility methods and allows excellent middleware use. As I was using it to call PostGreSQL I required the use of extra methods to handle connection pools and clients. Setup was as easy as the following code.



Once this was setup, I could create API end-points which would query specific parts of my database. As discussed previously, I required storage of all Nodes and Edges, which were simple ‘*SELECT \**’ statements I could create within my API. The following is an example of how this would work.

This would be accessible using a URL such as: <http://localhost:400/allNode> and would return a JSON array of nodes.

The following were the final end-points

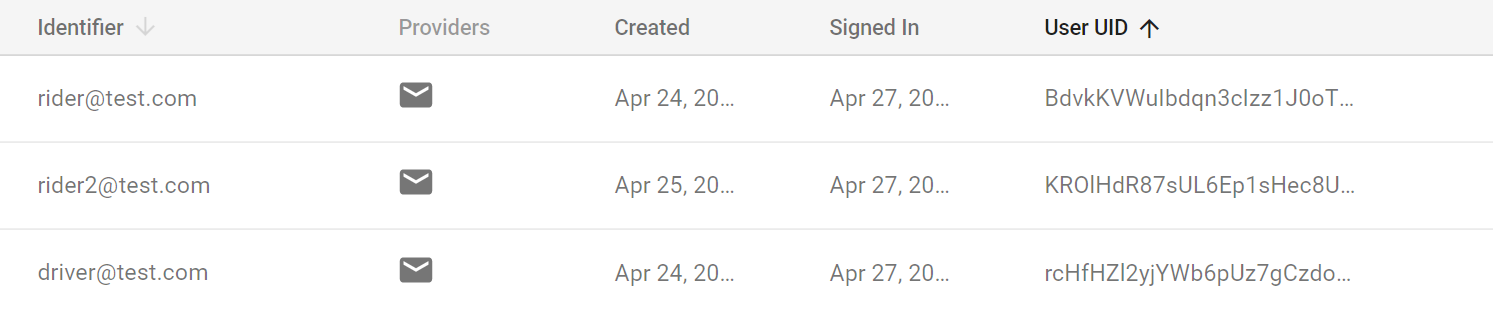
|  |  |
| --- | --- |
| Name | Arguments |
| /route?source= | Node Id |
| /neighbours?nodeID= | Node Id |
| /node?nodeID= | Node Id |
| /allNode | -- |
| /allEdge | -- |

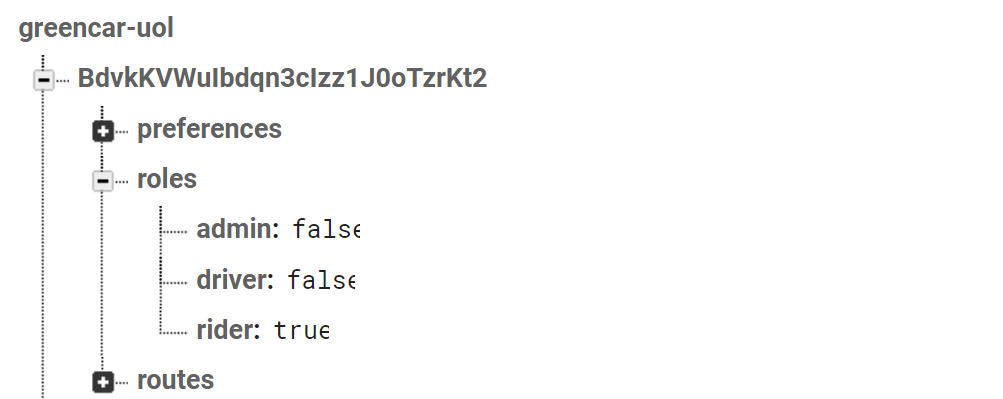
## Users

A user has the ability to Register and/or Login to the application. Registration and Signing in is handled by Firebase using an Authentication Service described above. Users also have additional information associated to them while using the application.

A user can set preferences after signing into the application, such as choosing kilometres or miles as their distance type, meeting radius (km/miles) as well as setting a wait time. These preferences are set within the preferences view and once saved are sent to Firebase using the UserService Component using the HTTP PUT utility available by importing the HTTP module into the application.

Roles are also set during the registration stage, choosing either *rider* or *driver*. Once chosen this is sent to Firebase during registration and stored under the user’s Unique Identification. The UID is obtained via Firebase using its Auth function, which gets the current user’s details. This is then used as the root structure when storing data about specific users. Below is an example of how a User’s data is stored into firebase, first showing the Firebase authentication view, highlighting the UID and then the Database view with stored data.



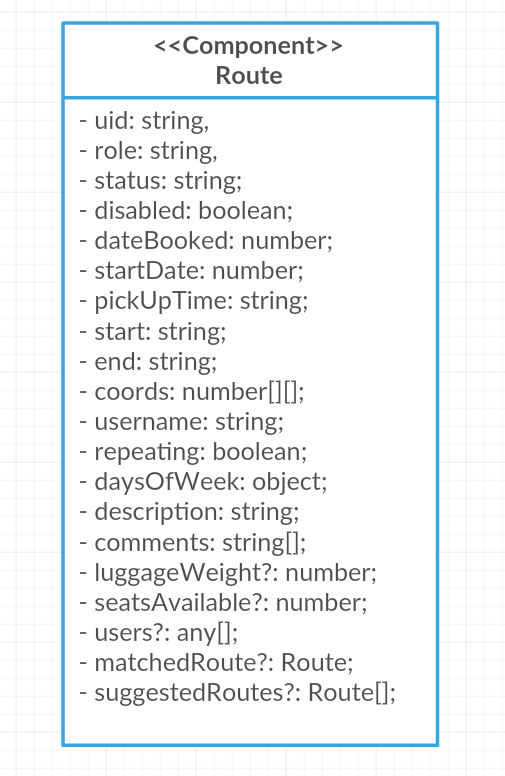


Admin User

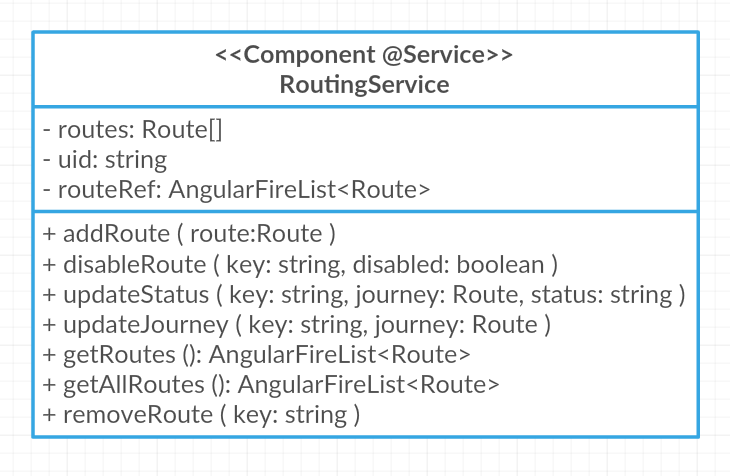
## Journey Creation

Journeys can be created by Riders & Drivers by selecting ‘Create Journey’ from the home screen of the application. A User can enter certain details about their journey, such as Start Time, Pick up Time, whether the journey is repeating or not and the days it would be repeating on and specific to a rider is luggage weight, while specific to a driver is available seats.

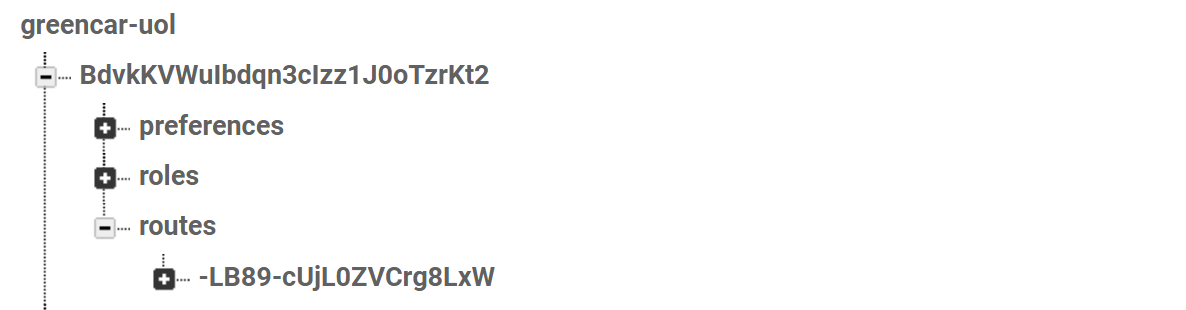
Once the Journey details are filled in, they are created by using the Route Model, which takes all the above details above. The Model for which can be seen as:



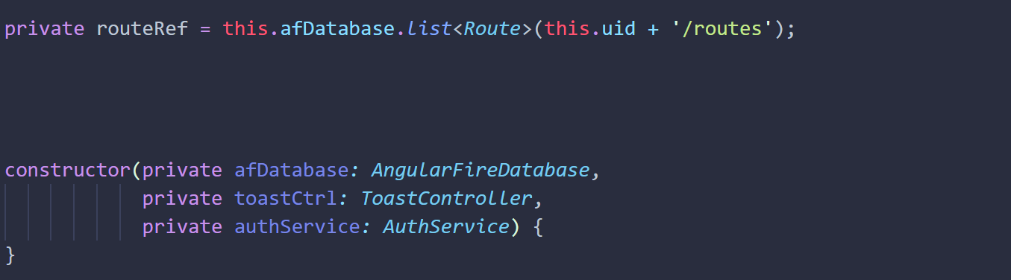
The attributes which ‘?’ are optional indicators available using JavaScript/TypeScript. For example, a driver won’t need to create a Route with luggage weight, while a rider would. Optional arguments are a great replacement for constructor overloading, which is common in languages such as Java where the need for multiple Objects with slight argument variations.

Each Journey created is added to an array of Journeys (or Routes) which is controlled by the RoutingService (this can also be thought of as a JourneyService). There are basic CRUD operations within the RoutingService such as adding a route, disabling a route, or updating its status. The RoutingService is as follows:

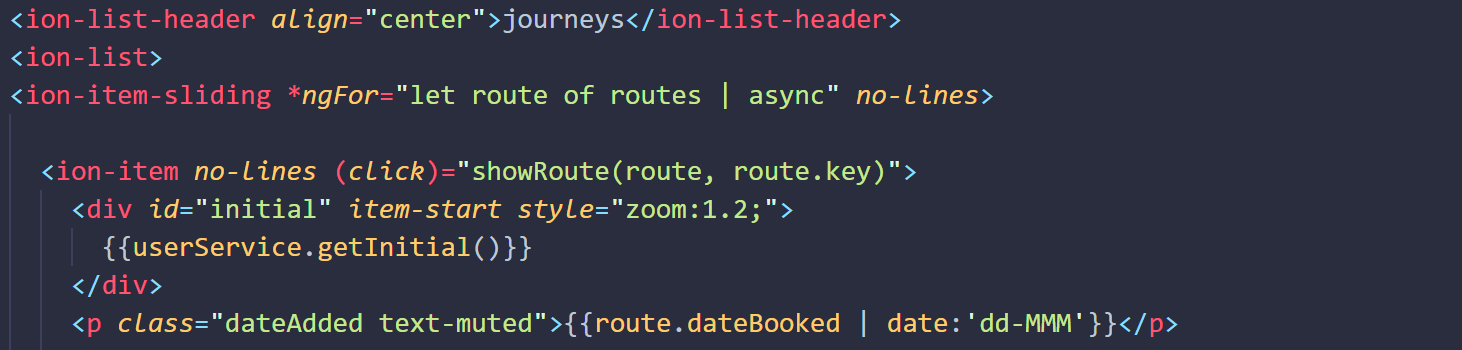
AngularFireList are observable objects returned from references to specific parts of the database. For example, once a route is pushed into Firebase using AngularFire, a key is stored as the parent of each individual route, which can later be accessed.



When displaying the route to the user via an AngularFire Observable call pictured below, you return all routes. Which can then be called by the Component to display to the user, and once displayed you also have access to the key for each returned route.



Having access to each key is extremely useful. The returned Observable list can be displayed to the user within the HTML Template by using Angular directives such as *\*ngFor*, which are similar to JSTL or ThymeLeaf operations.



This is a small example of using the Angular Directive to loop through a variable called ‘routes’ within the Component TypeScript (‘.ts’) file, which is typically of an Observable type. The intial ‘let route’ specifies that it is an i-th element of the routes array, thus you can have more template binding and by using calls such as {{ route.dateBooked }}, you call the individual attributes of each element

## Matching

The application will match riders to driver journeys if they match on certain criteria. Riders are matched to Drivers based on the following

* Whether or not the driver route is disabled or not (Disabled Driver Routes won’t be used in the matching process)
* If the Riders Journey isn’t repeating, then match it to Driver Journeys running on the Riders specified start date.
* Match Riders to Drivers if both of their pick-up times are within the window of the riders wait-time specified in their preferences. E.g. A driver journey pick up is 8:30, a rider chose theirs as 8:15 but allowed a 20-minute wait time, so these two would be matched.
* Riders are only matched to routes where the start & end points are within their given radius (km/miles).

The first steps of the matching process involve retrieving all journeys created by drivers. This means fetching all routes from users with a user role where ‘driver’ is equal to true.

This method is contained within the JourneyRetrievalService, and populate an array of Route Objects under the following conditions

* Routes must not be undefined or null
* Routes must not have the same UID as the current user (won’t suggest own routes)
* Routes must have been created by a driver

This service is called by the JourneyMatchingService, which requires an array of journeys to perform the next step of the routing process.

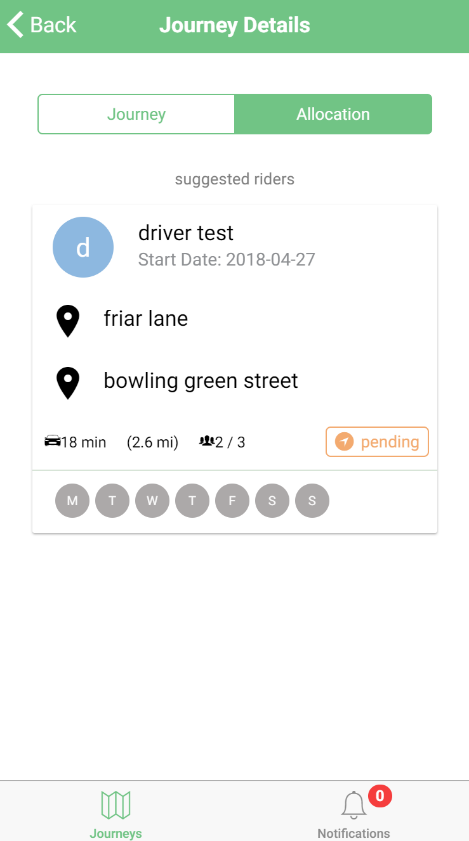
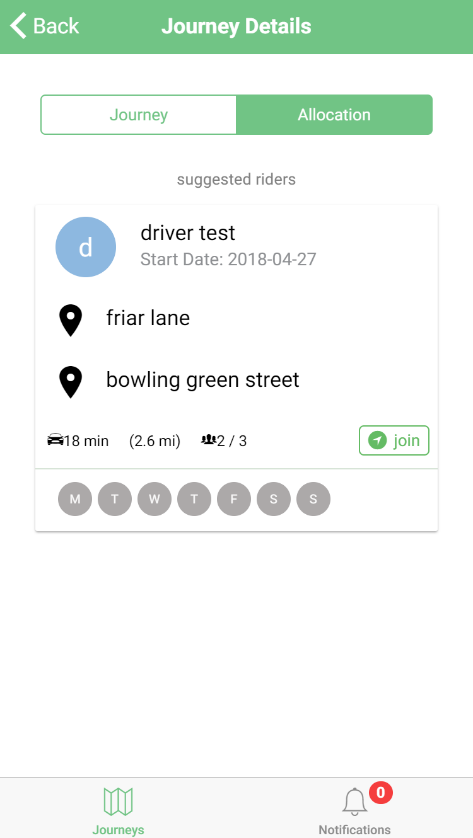
After the matching service has populated an array with appropriate journeys, it executes a method to find the closest journey cords from a given latitude and longitude pair. Given this coordinate pair, the method will loop through each journey within the journey array and calculate the distance between the current journeys in the loop against the given coordinate pair.

This distance calculation is computed by using the Haversine Equation, with some key criteria it needs to meet. The journeys it compares are put into an array of closest journeys if the calculated distance is less than or equal to the radius specified by the user preferences.

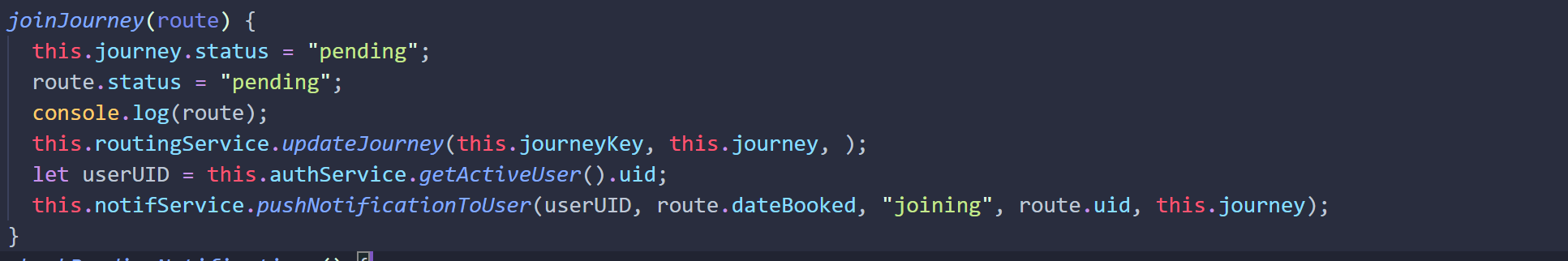
Once an array of journeys that match based on radius, I can then attempt to refine this list by checking it against criteria such as journey times, repeating journeys and whether it is disabled or not.

This works by passing in the newly created rider journey and checking it against the returned matches based on radius. Date Objects are created to break time and start date down, then matches are looped through with checks included before adding them to a final suggested matches array which is returned to the user.

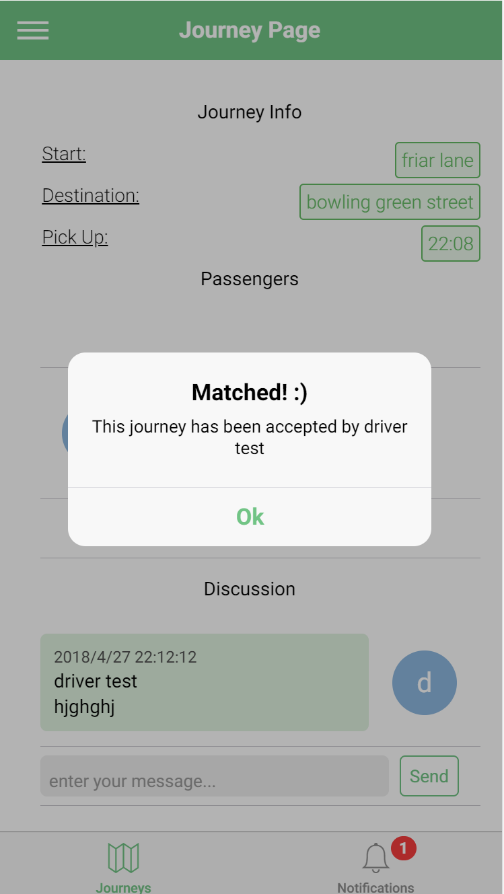
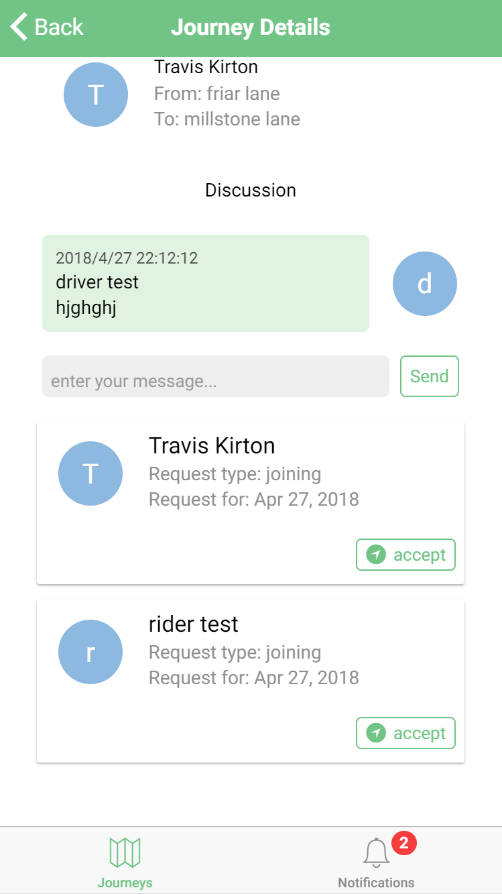
Once the suggested matches have been populated, they are viewable by the rider within the **Allocation** section of the Journey View page, pictured below. It Shows details of the route, such as the start and end locations, current amount of riders and repeating days highlighted or not. The rider can then click **join** which sends a request to the driver of the journey, while setting the rider’s journey to ‘pending’.



The driver recieves a notification and can then accept this journey request. The request system is handled by notifications, the first beind sent to the driver, which includes information such as the UID of the requester, the date booked of the matched journey, the UID of the driver, a string containing “joining” and all of the journey details that the rider created.

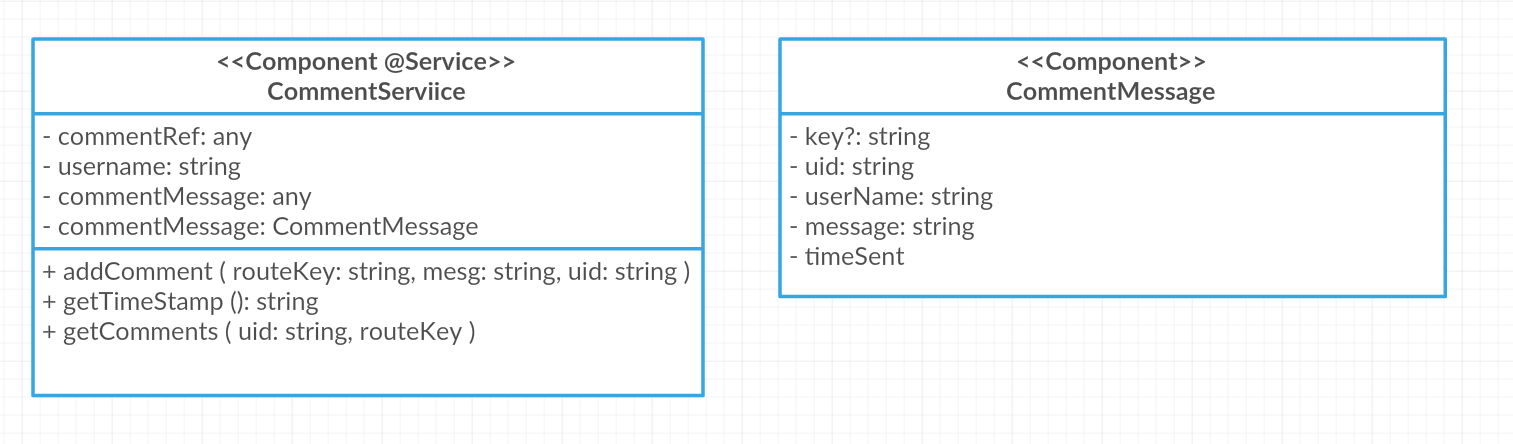


On the driver side, a method called **checkPendingNotifications()** is constantly checking for incoming notifications, and with the rider request will populate the drivers journey appropriately. Once accepted, a notification is sent back to the rider containing a string of “accepting”, which is also checked by **checkPendingNotifications()** and matches the rider to the journey, giving them access to the shared driver view.



## Messaging

The application contains two messaging functionalities, both of which share similarities in the way they are implemented. These are commenting in journeys between riders & drivers, and users being able to message each other privately. Both commenting and private messaging have separate services implemented.

CommentService is used for leaving comments in journeys, giving a public forum for the individuals involved to discuss the journey.

The above diagrams show the CommentService, along with the CommentMessage model component used for storing messages, with optional keys associated with them. To push a comment, a user needs to be accepted into the journey. The CommentService is then called from the JourneyView, passing in the key, comment and UID of the user commenting. The comment is retrieved using Angular Binding to a variable of type string.

TypeScript File: Comment:string = ‘ ’;

HTML: ([ngModel])=”Comment”

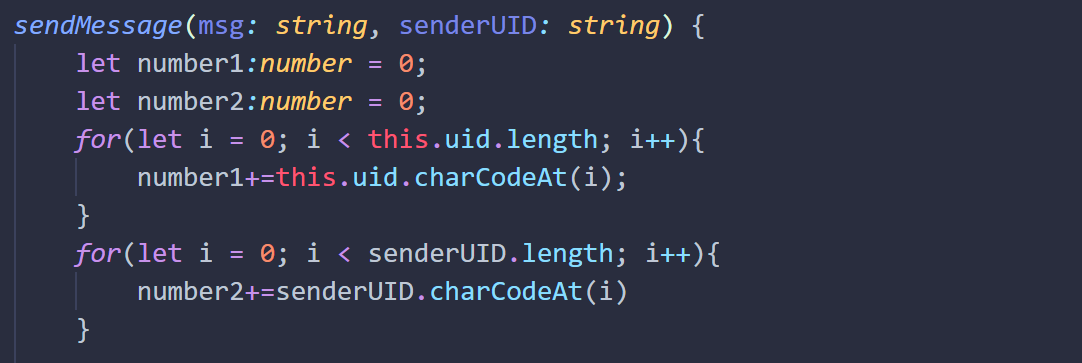


The comment is then pushed using the above addComment method, this creates an AngularFire reference to the comments section of the route the user is commenting on (obtained via the routeKey argument). A timestamp is associated before creating a CommentMessage object and pushing it to Firebase.

Comment Retrieval is handled via an observable array of type CommentMessage. This is then populated during the JourneyView initialisation, calling the CommentService **getComments()** method.

Once populated, an \**ngFor=”comment of commentList$”* Angular Directive can be created within the HTML Template like so:

Messaging between users is configured the same as the comment service, with the only difference being how message are stored and retrieved within Firebase. Messages must be stored uniquely for two users, and each user has a UID. So to create a unique identifier to store messages, I get both UIDs of each user messaging each-other and create a number based on the lexicographic value for each UID added together. This allows a unique root for messages.



## Testing

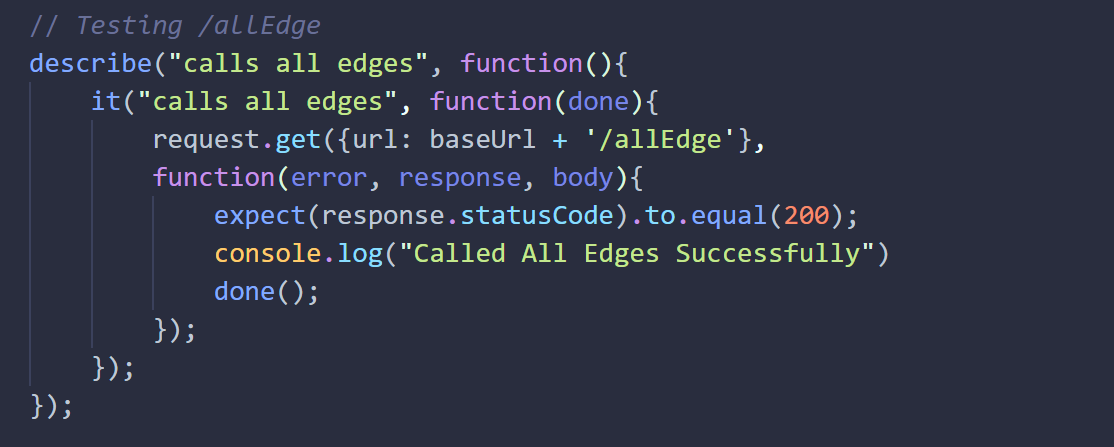
Testing is of vital importance when creating an application that is both mobile & web-based, as well as being targeted at a high amount of users. Multiple types of testing were taken into account during development and implementation of the application.

### REST API

The RESTful API was tested using Mocha & Chai. Mocha is a JavaScript test framework that allows testing on NodeJS programs, giving feature rich testing support and asynchronous testing and coverage. Chai is a BDD/TDD assertion library for NodeJS that provides excellent test components to test with.

I used both Mocha & Chai to test end-points of my API, making sure correct JSON responses were received, either with the correct status response code (200), or by checking specific JSON object attributes returned.

The following is an example of using **describe** & **it** to ensure the test returns all edges.



### Ionic2 Application

Testing for the application it-self was done via a mixture of unit testing using Karma & Jasmine, as well as traditional Black-Box Testing via using the application and the general user experience. I created a Black-box test document, outlining usability tests that would encapsulate activities that the user would take.

#### Device Testing

As the application was deployed onto mobile devices as well as web-based. I made sure that the Black-box test documentation included a section for hardware testing. This was testing on my personal phone, an android based phone.

Ionic also provides a very nice ‘Ionic Lab’ feature, which allows you to see what the application looks like across all three operating systems, this includes IOS, Android & Windows. Using Ionic Labs I could see any graphical issues that would arise when the application is viewed on a different device.

# **Results & Discussion**

## Summary of Work

The application was implemented to hit key functional requirements and provide an in-depth usage of routing algorithms applied to a Hybrid Web Application. The core requirements were taken and outlined, grouped into features that I split across each semester, which were outlined within the produced interim report. During each semester I split these features into atomic components, this would allow me to visualise them onto the Kanban board setup in my room.

For example, implementation of routing is and was a very broad requirement. Routing encompasses algorithm usage, data interpretation and storage, as well as data normalisation if any is required and using the results to display to the user. Using the techniques described above, I could meet requirements at each stage of development, and if I had difficulties with one requirement I could simply place it back into my project backlog and attempt it in the next 2 week sprint.

I feel the choices I made regarding technologies such as Ionic2, Firebase and NodeJS were extremely well received by myself and others who were shown the application. Ionic2 is a powerful framework with a very intuitive UI that allows use of components which mimic native mobile device components. Having access to UI elements that everybody can recognise was interesting, as I and other users who tried out the application could feel comfortable using it. The application design resulted in an intuitive feel which wasn’t confusing or cluttered.

Map Routing was particularly complicated for myself, and took some time to wrap my head around. However, once I had grasped the algorithms used and come to a conclusion on the technologies to facilitate implementation I found it a joy to complete and see work in the end. Map routing as a core requirement allows the application to have another layer of experience for the user, over sites such as BlaBlaCar.com which only have fields for ‘to’ and ‘from’, using map routing to display a route to both riders and drivers really does feel more involved. The map routing was a success in some senses, as implementation of Dijkstra’s worked, however A\* is limited in its ability to work on all roads. A\* worked enough to test speed and performance, but would require more work.

I would highlight some parts of the work completed that didn’t go as planned, nor were they appropriately planned for. Statistics and using an infographic page to show user history, such as journeys completed & money saved was originally an idea but due to time constraints and needing to focus more on functionality it was removed.

If I were to attempt a similar project again, I would prioritise testing at the same level of development during each stage. Testing was something completely lacking throughout the project and thus required basic Black-Box Testing. I feel Test Driven Development is a proven technique and widely recognised by professionals.

## Discussion of Social & Economic Impact

The current generation is becoming more and more mobile, with 85% of respondents in a recent study owning, or having access to a smartphone [14]. Millions of people own cars, and millions own smartphones, so creating an application that could be used to impact millions of lives in a positive manner is an aim of applications like mine.

A carpool application would reduce the cost of driving, especially commuting multiple days a week as petrol would be split between 2+ people minimum. Companies could enjoy a platform to advertise on, as the application could be open to host ads and generate profit for itself and others. Other benefits of the application include impact on health, as studies have shown that respondents were keen to take part in carpooling knowing it could reduce stress and save money [15].

Carpooling has been on the decline since the late 70s, yet with a higher cost of living, especially in cities such as London [16] it would be a great moment for drivers of Single Occupancy Vehicles (SOVs) to take part in carpooling services such as GreenCar@UoL, which could be aimed towards everyday users, or using a template system change depending on the area you’re using it in. For Example GreenCar@Ldn, or GreenCar@Mch for London and Manchester Respectively.

The technologies used to create the application kept everyday users in mind. As it was built using hybrid web technologies it can be used on any platform with a browser and internet connection, or if the user prefers they can download it as an App which would only increase functionality, as it opens up possibilities for push notifications and geolocation tracking. Hybrid Web Technologies aren’t brand new, but aren’t used as much compared to native mobile or straight web applications. Hybrid Web Applications can be slower because they have to translate JavaScript and Web programming languages into machine code to be read by mobile device for native-like app usage. GreenCar is not an intensive app however, with no need to render multimedia videos or take pictures. The application itself is light and doesn’t experience UI issues while running on mobile devices natively.

The final project is relevant in the commercial industry as it’s an application that could be refined, developed and would be marketed to the general public, giving it a real possibility to work as an application for users in the real world. Carpool applications that are mobile or web-based in the UK have limited accommodation for commuters, rather allowing journeys between major cities. Commuters are amongst the biggest car users in the UK, and giving a platform for them to share their ride, reduce cost, socialise and reduce the environmental impact of Single Occupancy Vehicles on the road is extremely beneficial. The final project is a perfect example of what should be happening within the UK. When searching on google for ‘carpooling UK’, top results include liftshare.com and BlaBlaCar.com, both of which only offer one time journeys. More emphasis on company carpooling and everyday commuting should be generated.

## Personal Development

During the development of this application I learned an incredible amount, including technical skills, organising myself and researching material and technologies. I purposefully picked Angular and Ionic2 as the core technologies for this project as they were interesting to myself and were languages and tools that had been recommended to myself by a work colleague while on my placement. Building up the technical skills to be proficient in both was really rewarding. I didn’t have a curriculum to follow as I would at university, but had to learn each concept by reading online material and mastering the languages through trial and error.

I felt I gained a greater ability to grasp material quicker and filter out information that wasn’t necessary. Using these developed skills I could implement features quicker and use it to refine my map routing usage. I gained a better understanding of breaking down tasks into manageable chunks, which gave me a greater appreciation of Agile Delivery of projects. I definitely used these skills during job applications and felt the project itself was the core reason I secured a graduate position.

# **Conclusion**

## Overview

The application has successfully implemented core requirements outlined by myself and the interim report requirements expectations, along with capturing key functionality that is lacking within alternative carpooling applications and producing it on a format that is more easily accessible by users. Hybrid Web Applications are an acceptable solution and provide fast implementation, allowing greater focus on functionality and meeting goals over having a truly native feel on mobile devices.

Throughout development of the application it has highlighted key concepts within Map Routing, showing that algorithms such as Dijkstra’s can outperform implementations of others like A\* in short distances, yet longer distances are better applied using A\*. Taking the map routing results highlight future improvements that could be made using more efficient routing algorithm techniques.

Implementation of storage using Firebase has been surprisingly pleasant. Firebase is a useful tool and includes great functionality that can be utilised to reduce the need to implement bespoke Authentication and data storage solutions. JSON storage is intuitive, fast to access and easy to manipulate. Having used Ionic2, which is based on Angular2 (a superset of JavaScript) has allowed easy integration with Firebase. JSON Objects are easily made within JavaScript & TypeScript, allowing pushing objects and data from the application to Firebase to be comfortable.

The applications design considerations were taken into account throughout all stages, using HCI and UI design techniques to produce a minimalistic interface that would fit in with most web and mobile based applications today. I chose a minimalist approach, aiming to make the information available part of the design, such as styling borders around information and creating buttons that would add to the overall UI experience of the application.

## Further Development

The application could benefit from further development considerations, such as more efficiently implementing mapping functionality. Working with both Dijkstra’s and A\* to produce a hybrid of the two, or using alternatives such as Bi-directional Dijkstra’s. Working with OpenStreetMap Data to produce a graph that can be searchable via PostCode or street address would have felt more intuitive, however using tools such as Googles Geocoding would provide extra functionality.

I believe that development on an Admin Interface would have been feasible, however Firebase only allows an Admin API to be run as a Server, which would require extra work on architecture. Proxy services calling the API from the application are a possible solution for this.

The potential for applications such as this are very wide, giving extra social functionality and incentive systems, possibilities of reward systems for using the application are a viable concept and would allow revenue for whomever is running the application. Systems that implement reward features for successfully achieving a goal or completing a purchase are viable business decisions for this type of project and would incentivise more use. Rewarding users after 6 completed journeys with some promotional deal or content would allow more user enjoyment and could increase the amount of traffic to the system, as more people would want to take part, saving money and earning rewards all at the same time.

# **Bibliography**