

# Technical Manual

Participants: Afolabi Fatogun (20409054), Adrian Irwin (20415624)

Project Name: Carpool2DCU

Finish Date: 24/02/2023

<b>Introduction</b>	<b>2</b>
Overview	2
Glossary	2
<b>System Architecture</b>	<b>3</b>
<b>High-Level Design</b>	<b>5</b>
Data Flow Diagram	5
Context Diagram	6
<b>Problems and Resolution</b>	<b>6</b>
<b>Installation Guide</b>	<b>7</b>
Prerequisites	7
Step 1	7
Step 2	7
Step 3	7
Step 4	7
Step 5	7
Step 6	7
Step 7	7
Step 8	7
<b>Testing</b>	<b>8</b>
Snapshot Testing	8
Integration Testing	8
System Testing	9
<b>Appendix</b>	<b>12</b>

# Introduction

## Overview

Carpool2DCU is a mobile carpooling application that is operational on both Android and IOS devices. It aims at getting DCU students to campus while reducing their digital footprint, as well as decreasing transportation costs throughout the university. The application was developed with ease of use in mind as well as respect for individual constraints, which is why students can filter their ideal partner out of available ones.

The application works by allowing registered students to create a profile, which includes their name, email address, and preferences such as the preferred route and time of travel. Drivers can then offer rides to other students who have matching preferences.

## Glossary

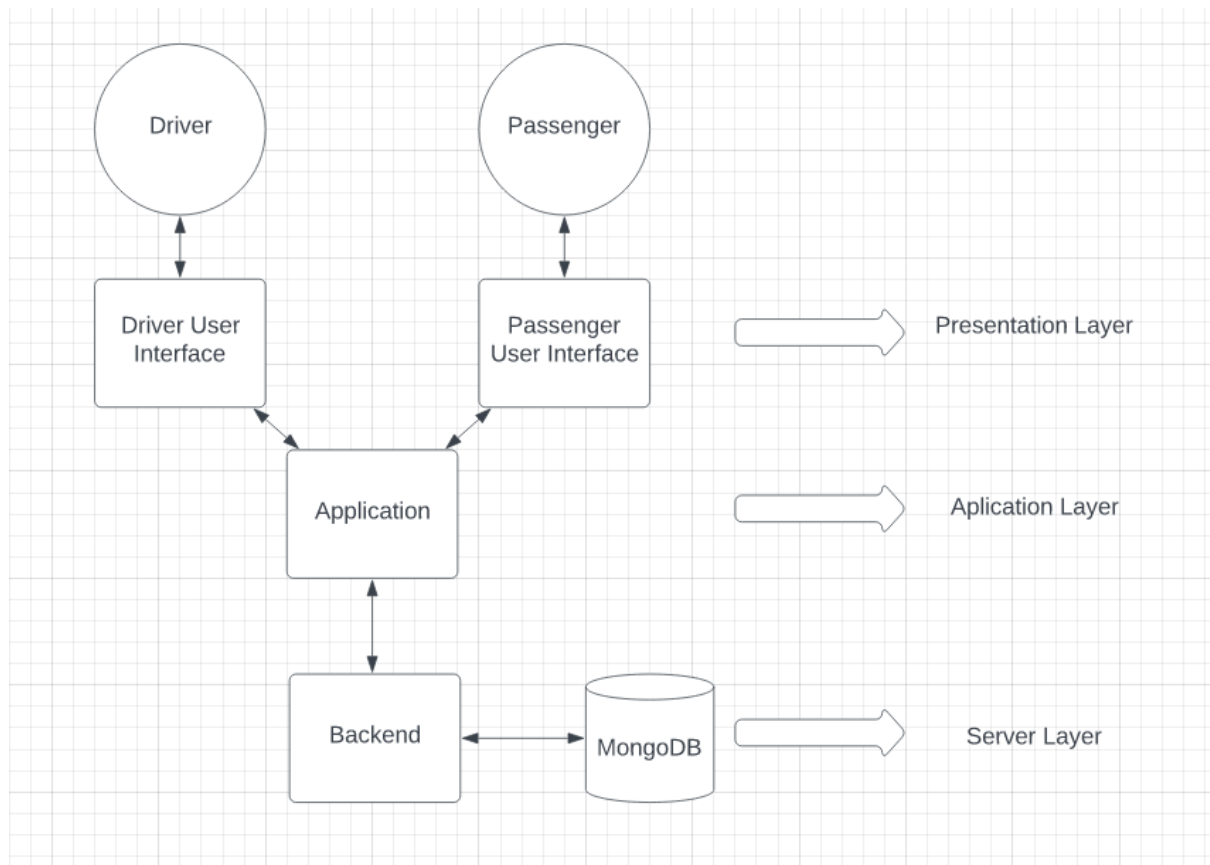
- Carpooling: The practice of sharing a car journey with one or more other people, typically to reduce costs and/or carbon emissions.
- DCU: Dublin City University, an Irish university located in Dublin.
- Driver: A student who is offering a ride to other students.
- Passenger: A student who is looking for a ride to DCU.
- Preferences: Specific requirements or requests made by a user, such as the preferred route or time of travel.
- Profile: A collection of personal information and preferences associated with a user's account.
- Android/IOS: Mobile operating systems

## System Architecture

Carpool2DCU is a client-server application, with the client being a mobile application user interface, and the server is a database that uses a web application framework.

It mainly follows the below diagram and has had no new iterations since it was designed. The application's architecture is designed in 3 layers as can be seen in the below chart.

- The Presentation layer: This is implemented using react-native and written in javascript. This layer handles all user inputs to the system and displays information to users
- The application layer: This layer is an express app also implemented in javascript and is a middleman between the presentation layer and the database layer. It processes user input and interacts with the database.
- The server layer: This layer is also implemented using MongoDB and code written in javascript. This layer is responsible for storing and receiving data from the application layer.

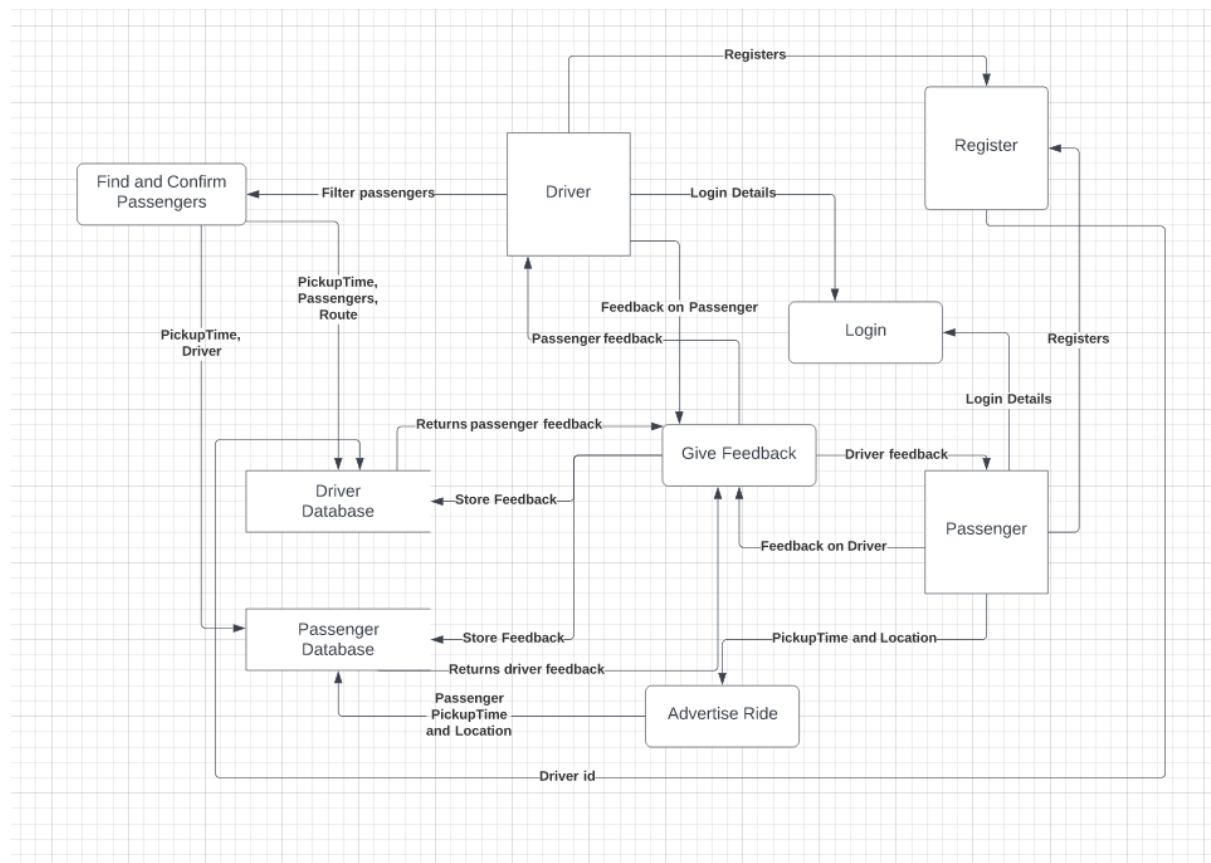


The application has been designed to be scalable, with the ability to add additional application and database servers as needed.

# High-Level Design

## Data Flow Diagram

The data flow diagram describes the dynamic travel of data throughout the system and the response to internal and external actions.

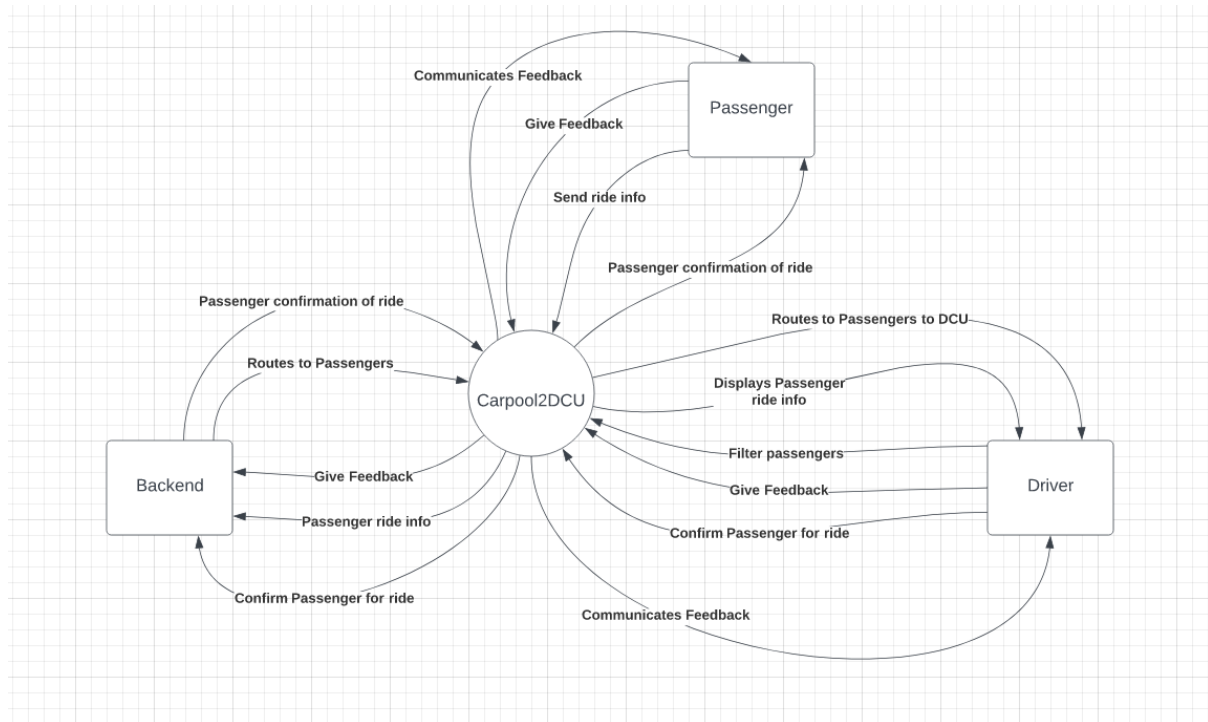


In this revised diagram there are additions and deletions to certain components. There is an addition of a 'Register' interaction as this was introduced to the system. Also, a deletion to the 'Rides database' component as the system does not make use of a specific rides database. Instead, there was a reroute of data from the 'Advertise Ride' interaction to the Passenger database.

Finally, the 'Give Feedback' interaction will also send feedback to the component from which it came.

## Context Diagram

The context diagram describes the interaction between the two user classes, the backend, and the application. It shows all the functions of the system components.



In the context diagram, there are fewer iterations from the initial one described during the functional specification. There is an addition of a 'Filter Passengers' function which goes from the driver to the application as well as an addition of a 'Communicates Feedback' function from the app to both user classes.

## Problems and Resolution

**Problem:** When using the Nominatim - OpenStreetMap geocoding API we found that we were either getting inaccurate or no results.

**Solution:** To fix this issue we switched to the Bing Maps API which was more accurate in providing location coordinates. The Bing Maps API is a location-based service that Microsoft provides mapping, geocoding, and routing capabilities to developers similar to the OpenStreetMap API.

# Installation Guide

## Prerequisites

- NodeJS LTS + npm
- Expo Go mobile application from either the IOS or Android store.
- Two terminals
- MongoDB connection string
- OpenRouteService API Key
- Bing Maps API Key

## Step 1

Clone the repo: ``git clone`

`https://gitlab.com/computing.dcu.ie/irwina7/2023-ca326-carpool.git``

## Step 2

In one terminal, go to the frontend folder: ``cd src/Carpool`` and run ``npm i``

## Step 3

Create a ``.env`` file using ``.env.example`` and fill in the API key environment variables.

## Step 4

Run the frontend using ``npm start``

## Step 5

In a second terminal, go to the backend folder: ``cd src/backend`` and run ``npm i``

## Step 6

Create a ``.env`` file using ``.env.example`` and fill in the MongoDB connection string environment variable.

## Step 7

Run the backend using ``npm start``

## Step 8

Scan the QR code in the first terminal using your camera to open the application.

## Testing

During the implementation and design of Carpool2DCU, we used different forms of testing.

Some of the tools we used were:

- Jest
- Jest-Expo
- React Native Testing Library
- react-test-renderer

### Snapshot Testing

This is a type of testing where a snapshot of a rendered component is compared to a reference snapshot file. The test passes if there are no differences between the snapshot and the file.

We have used snapshot testing to confirm that the components are rendered correctly. See an example below

```
describe("TransitionCard", () => {  
  it("renders correctly", () => {  
    const { toJSON } = render(<TransitionCard />);  
    expect(toJSON()).toMatchSnapshot();  
  });  
});
```

### Integration Testing

This is used to combine individual pieces of code and test them together to ensure that all components are working together as expected.

In our implementation we utilized integration testing to see that other components are properly implemented into App.js, e.g. TextInputField is properly rendered on the login screen that is implemented into App.js. See the example below.



```

import React from "react";
import { fireEvent, render } from "@testing-library/react-native";

import App from "../App";

describe("App", () => {

  it("renders correctly", () => {
    const { toJSON } = render(<App />);
    expect(toJSON()).toMatchSnapshot();
  });

  it("renders Login screen by default", () => {
    const { getByText } = render(<App />);
    expect(getByText("Login")).toBeTruthy();
  });

  it("renders Register screen when Register button is pressed", () => {
    const { getByText } = render(<App />);
    const registerButton = getByText("Register");
    fireEvent.press(registerButton);
    expect(getByText("Register")).toBeTruthy();
  });

  it("renders email text input", () => {
    const { findByTestId } = render(<App />);
    expect(findByTestId("Email")).toBeTruthy();
  });

  it("renders password text input", () => {
    const { findByTestId } = render(<App />);
    expect(findByTestId("Password")).toBeTruthy();
  });
});

```

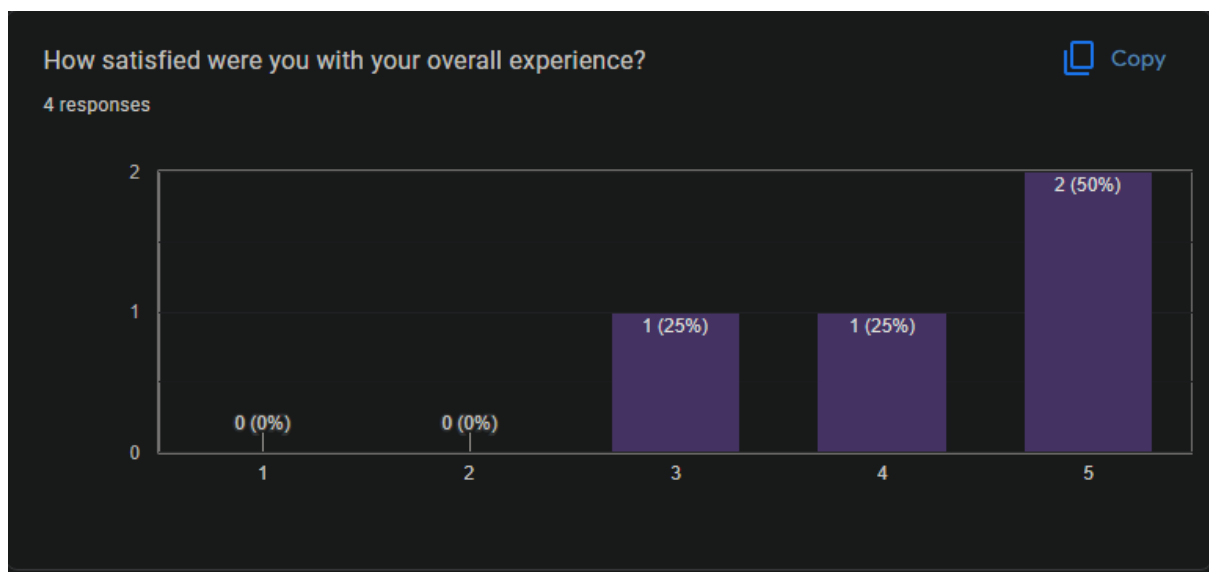
## System Testing

This is a form of testing in which a tester verifies the integrity of a “completed” software application. We recruited some testers for this testing phase under directed guidelines and ourselves in developer situations. Under system testing, we carried out the following methods;


- Load testing: In which we tested how Carpool2DCU handled under intense load. To do this we added 100 - 150 passengers to the passenger list, a number that is higher than what the system

would be expected to handle regularly. We monitored that the system handled this fluently and without setbacks. We also tested running many routes at the same time on the Driver end to which the system performed satisfactorily.

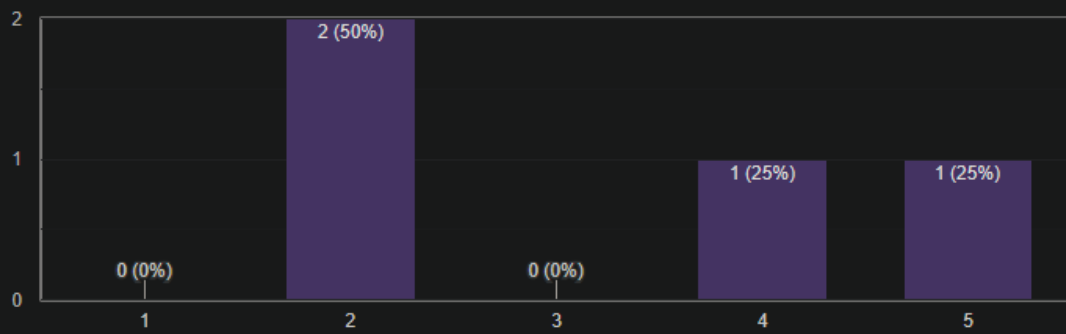
- Usability testing: In this method we implored the test to check the ease of use of the system and provided them with a questionnaire to fill out upon testing the software. Below are images of the feedback relating to usability.



How easy did you find using the feedback section?

 Copy

4 responses

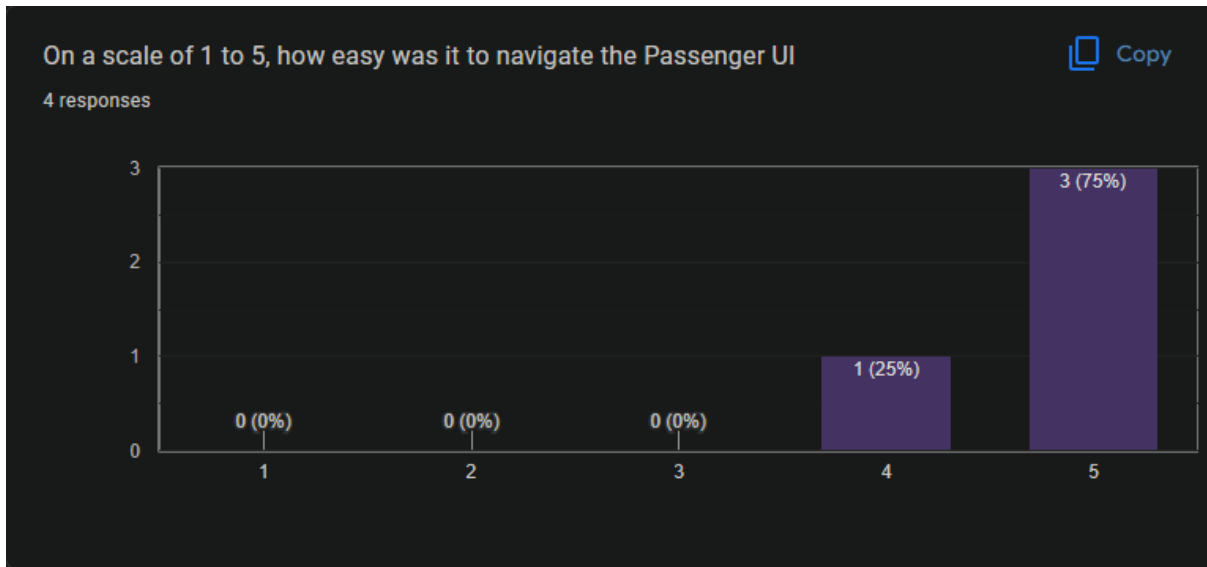


On a scale of 1 to 5, how easy was it to navigate the Driver UI

 Copy

4 responses





From the above responses, it was obvious that the test users were more than satisfied with the ease of use of Carpool2DCU

- Recovery testing: In this method, we measured the software's ability to recover from a crash. To do this we introduced wrong information into the database and tested if the system could properly respond to updating information while running. We recorded that Carpool2DCU had no issues updating the correct information.

## Appendix

- Link to all collected responses from user testing:

[+ Usability feedback on Carpool2DCU \(Responses\)](#)