Venato Technical Specification

Names: Aaron Cleary, Joao Pereira

Student Numbers: 19495324, 19354106

Supervisor: Michael Scriney

Date Created: 01/03/22

1. Overview

Our project, Venato, is a GPS tracking app for bikes. Venato is designed with the goal of providing security and peace of mind to fellow cyclists. The users of Venato will therefore be cyclists who want to prevent any potential theft of their bikes. Venato will be of significant value to our users, as our product will allow users to know exactly where their bike is at any given time. Venato will establish a direct connection between our users and their bikes, which will provide users with a strong feeling of assurance and security, knowing that their bikes are safe.

Our project is an all-in-one bike device and companion app that will allow users to keep their bikes secure. Users will be able to interact with our mobile app, which will enable them to connect to our device and easily access the range of tracking features that we will provide.

Our Venato app allows users to register their own personal account, enabling them to easily sign in and out of the app. The user chooses their unique username, a password and provides their email address for verification. In our system, each user has a unique user ID, allowing them to be easily authenticated by our user database.

Another feature of our app is the user profile. Each user has their own profile, containing important information about themselves. The user profile contains their name, Instagram handle, location, phone number and email. Furthermore, there are interactive features such as help, settings, terms and conditions and a sign out option.

The most important feature of our app is the ability for users to view the location of their bike. Users can see the current location of their bike through our GPS tracking system, which is then conveniently displayed on a map within our app. This map is the central tab on our app, which was a conscious design choice that emphasises the importance of this feature. This key feature is the selling point of our product, and is also what provides our users with peace of mind by knowing that they have an accurate, real-time location of their bike.

2. Motivation

Last summer, we cycled to Dun Laoghaire together to go for a swim. Unfortunately, Aaron's bike was stolen in broad daylight while we were swimming in the water. When we later realised that the bike had been stolen, we immediately recognised how useful it would be if we had a way to track the bike down. Having already decided to partner for our 3rd year project, we thought this could be a great potential idea for the project.

Shortly after starting the college year, it eventually came time to choose an idea for our 3rd year project. We remembered the idea that we had come up with months ago in Dun Laoghaire, and thus decided to research the feasibility of it. After realising it was possible to create a GPS tracker for a bike, we committed to this idea for our project. We named our project Venato, meaning 'tracker' in Latin.

Having both had our bikes stolen, we know that the safety of one's bike is of paramount importance, which gave us motivation to ensure that Venato could safeguard fellow cyclists from going through the same tough experience as we did. We believe that our project has a real, relevant use case that can provide enormous benefit to potential users.

3. Research

Throughout the course of our project, we have had to conduct research on several different components of the project. We have extensively researched components including programming languages, frameworks and IDEs. We will now break down the vast amount of research that was involved in our project.

3.1 Device Research

When we initially started the project, we needed to find a device that was capable of transmitting GPS coordinates to a program. We had also decided on using a GSM module to transmit the data, meaning that we required a device that could support a SIM card. We found that we could use an Arduino for this purpose, so we then went about researching for the optimal device.

We eventually decided on using the Maduino Zero A9G IoT Microcontroller for our project, as it had all of the capabilities that we would need. We also had to buy a SIM card for this device in order to receive the GPS data, and we identified the Arduino SIM card as the one we needed. Later in the project, as we elaborate on in the 'Problems and Solutions' section of the report, we had to buy a replacement device after our original one broke. Unfortunately, the original device was out of stock, meaning that we were forced into researching alternative devices. We eventually found the SIM808 GSM Module, which we then decided to buy after finding that it could effectively replace our original device.

3.2 Arduino Research

In order to get the necessary data from our device, we had to research how to write Arduino code that could provide an output to a serial monitor. Neither of us had any prior experience with using Arduino, meaning that we both had to watch several tutorials and familiarise ourselves with the Arduino IDE. Furthermore, we had to write our code for the Arduino in C++, which is a programming language that neither of us had used before. We therefore had to research how we could code in C++ in order to correctly implement a program that could retrieve coordinate data from our device..

3.3 Alternative Research

After countless hours of extensive testing with our device, we realised that we simply could not rely on it to produce the coordinate data we needed, which we elaborate on later in the ‘Problems and Solutions’ section of the report. We were incredibly frustrated that in spite of all our hard work, the device just would not work properly. However, we didn’t want our app to not be able to display coordinates because of this, so we decided to look for an alternative way to send coordinate data to our app.

Our initial idea was to create a Python script that could generate random coordinates within the DCU area. We chose to use Python because we are both experienced in using it and knew that we could easily implement such a script. After creating it, we then had to research a way to get the output of this script into React Native. We performed extensive research on the most efficient way to do this, eventually deciding to use a AWS Gateway REST API. Neither of us had any prior experience with AWS, so there was another learning experience as we familiarised ourselves with both the Lambda and Gateway API services and how they interacted with each other.

3.4 Front-End Research

We then had to research what framework we were going to use for the front-end of the project. Since we were going to be making an app for the first time, we wanted a framework that was smooth and had a responsive user interface. We found that the React Native framework and the Android Studio IDE would allow us to achieve these objectives, in addition to significantly reduced load times. Neither of us had used the React Native framework before, so we again had to follow various tutorials as we tried to familiarise ourselves with React Native. Since React Native is a JavaScript framework, we obviously had to use the JavaScript language when developing our app.

We chose to use Android Studio so that we could get an Android emulator that would allow us to test both the user interface and functionality of our Venato app. Our research showed that Android Studio could be easily integrated with React Native, meaning that using Android Studio would allow us to achieve what we were looking for.

3.5 Back-End Research

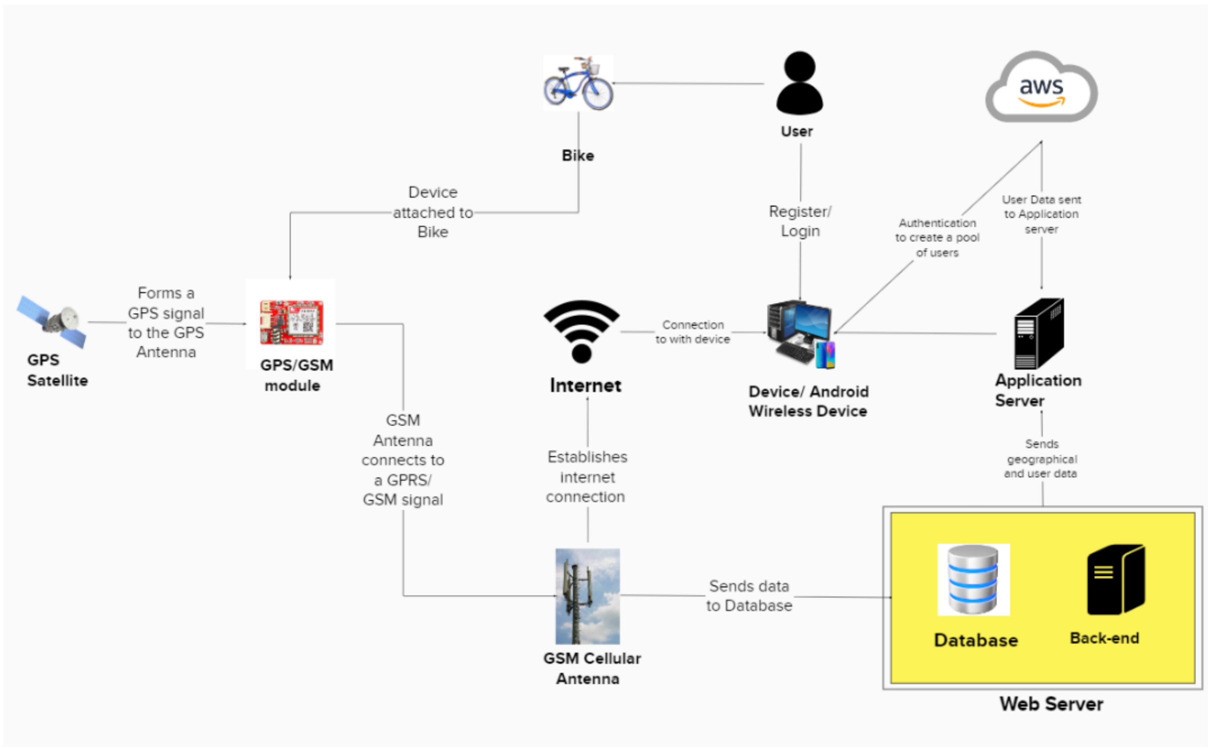
Upon starting the project, we knew that we would need a server and a database that contained the data we would receive from our GPS device. When researching our options, we made sure to factor in that neither of us were familiar with using an Arduino, meaning that a tutorial we could follow was ideal. We looked for tutorials for storing data with our specific device, however since it is a fairly niche device, there were only a handful of options. After researching all of them, we formulated our original plan for the back-end of our project, which was to have the device send data to a PHP web server that connected to a MySQL database. We decided to use a PHP web server because PHP is a server-side language that makes hosting local servers easy. For our database, we chose to use MySQL because we both have prior experience with it and it is relatively straightforward to connect to a PHP server.

As we elaborate on later in the ‘Problems and Solutions’ section of the report, our device issues meant that we could not get our device to reliably send coordinate data. This forced us into making the decision to find another way of producing coordinate data for our app. Our new plan involved a backend of an AWS Gateway API, in addition to AWS DynamoDB for our database. After implementing this, we realised that since we were going to be generating our own coordinates, there was no need for a database, so our final backend was just our AWS Gateway REST API.

4. Design

This section details the system architecture of our project. As seen below, there is a system architecture diagram, a data flow diagram and a use case diagram. The system architecture diagram contains the key components in our system and explains how they interact. The use case diagram and data flow diagrams are used to further illustrate the high-level design of our project.

4.1 System Architecture Diagram



Bike
User
aws
Device attached to Bike
Register/ Login
Authentication to create a pool of users
User Data sent to Application server
GPS Satellite
Forms a GPS signal to the GPS Antenna
GPS/GSM module
Internet
Connection to with device
Device/ Android Wireless Device
Application Server
GSM Antenna connects to a GPRS/ GSM signal
Establishes internet connection
Sends geographical and user data
Sends data to Database
GSM Cellular Antenna
Database
Back-end
Web Server

4.2 GPS Satellite -> GPS Module

The GPS/GSM module has a GPS antenna integrated into its hardware. This GPS antenna needs a GPS signal in order to be able to retrieve the coordinates of its current location. This required GPS signal is broadcasted by a GPS satellite located in space. Once the GPS signal is received by the device, the device’s GPS antenna can begin broadcasting its own location.

4.3 GSM Module -> GSM Cellular Antenna -> Internet

For any tracking device, we require a GSM antenna/module in order to be able to form a connection to networks such as 2g, 3g, 4g, bluetooth and Wi-Fi. The GSM antenna in our device is integrated with a sim card and when both are operating together, they form a network connection to the Internet to then furthermore have the ability to store and send data.

4.4 Device/Android Wireless Device

Our Venato app was built through a React Native application and an Android Studio emulator. In order for a user to access Venato's application, they must first have access to an Android device (as our app is currently built for Android only).

4.5 User, Bike

The aim of the Venato is for a user to have the ability to track their bike through the use of a GPS device and an application. This means that in order to track their bike, the user must first securely mount their GPS device onto their bike so that it can be tracked.

4.6 Application

Venato's front-end is created with React Native CLI, this is where we designed the user interface that the user will see when they are running the application. The application has access to the web server, as well as AWS, in order to retrieve important data such as user information and device data. This data is then utilised within the application to display necessary GPS tracking information to the user.

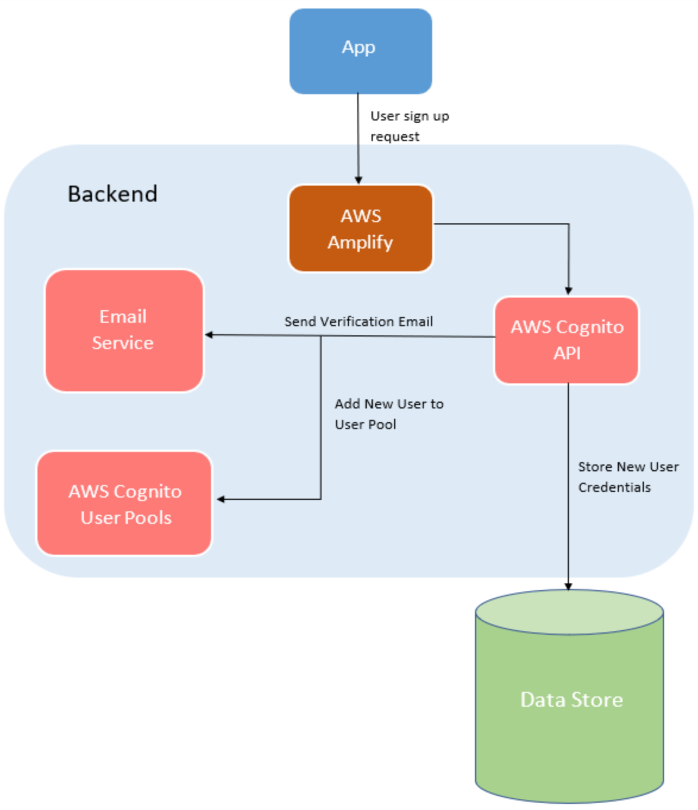
4.7 Web Server

The web server consists of both a database and a back-end server. This is where the behind-the-scenes of the application is operated. Data gets sent to our web server before then being stored within our database, and this data then runs through our back-end and is filtered to then be sent to the application.

4.8 AWS

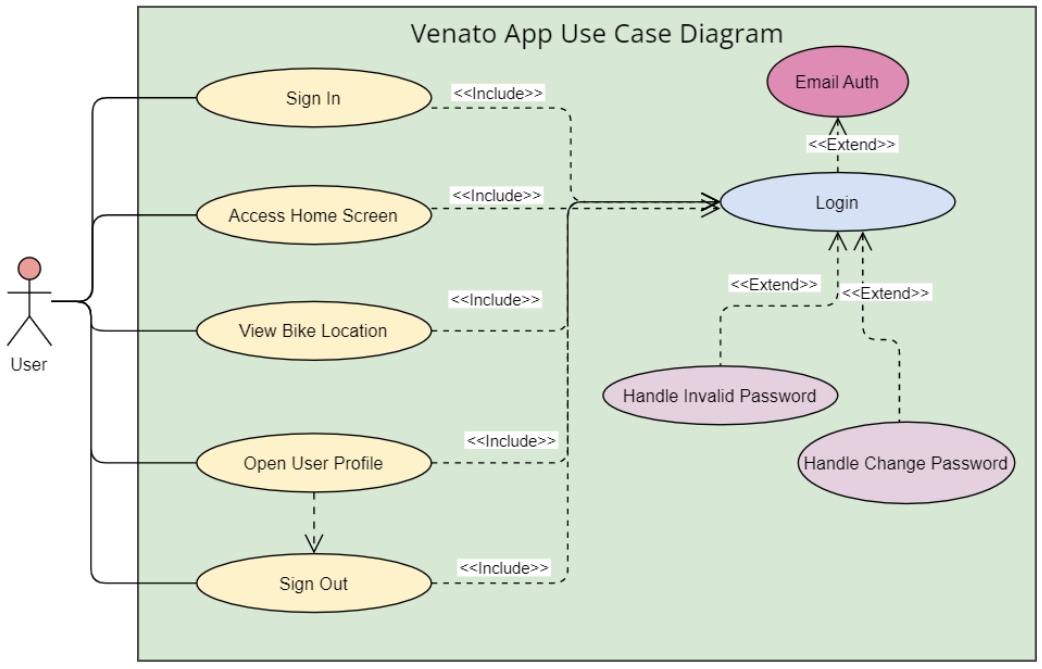
Throughout the project, we decided to use AWS and the services that it provides in order to create a functioning and effective authentication system for our users. Firstly, we used AWS Amplify to handle user registration/login/signout. We also used AWS Cognito to create and view user pools that contain our users' provided information.

4.9 Data Flow Diagram



The data flow diagram above illustrates the process behind the user registering for a new account. Our app forms a signup request based on the credentials that the user inputs on their screen. This request is then sent to our app's AWS backend. We used the AWS amplify framework in order to make a serverless backend. After a user's details are successfully validated by Amplify, the payload is sent to the AWS Cognito API. The Cognito API then sends a verification email to the email address that the user provided. After user verification, a new user is created in the Cognito User Pool, and the user data is stored to Amplify.

4.10 Use Case Diagram



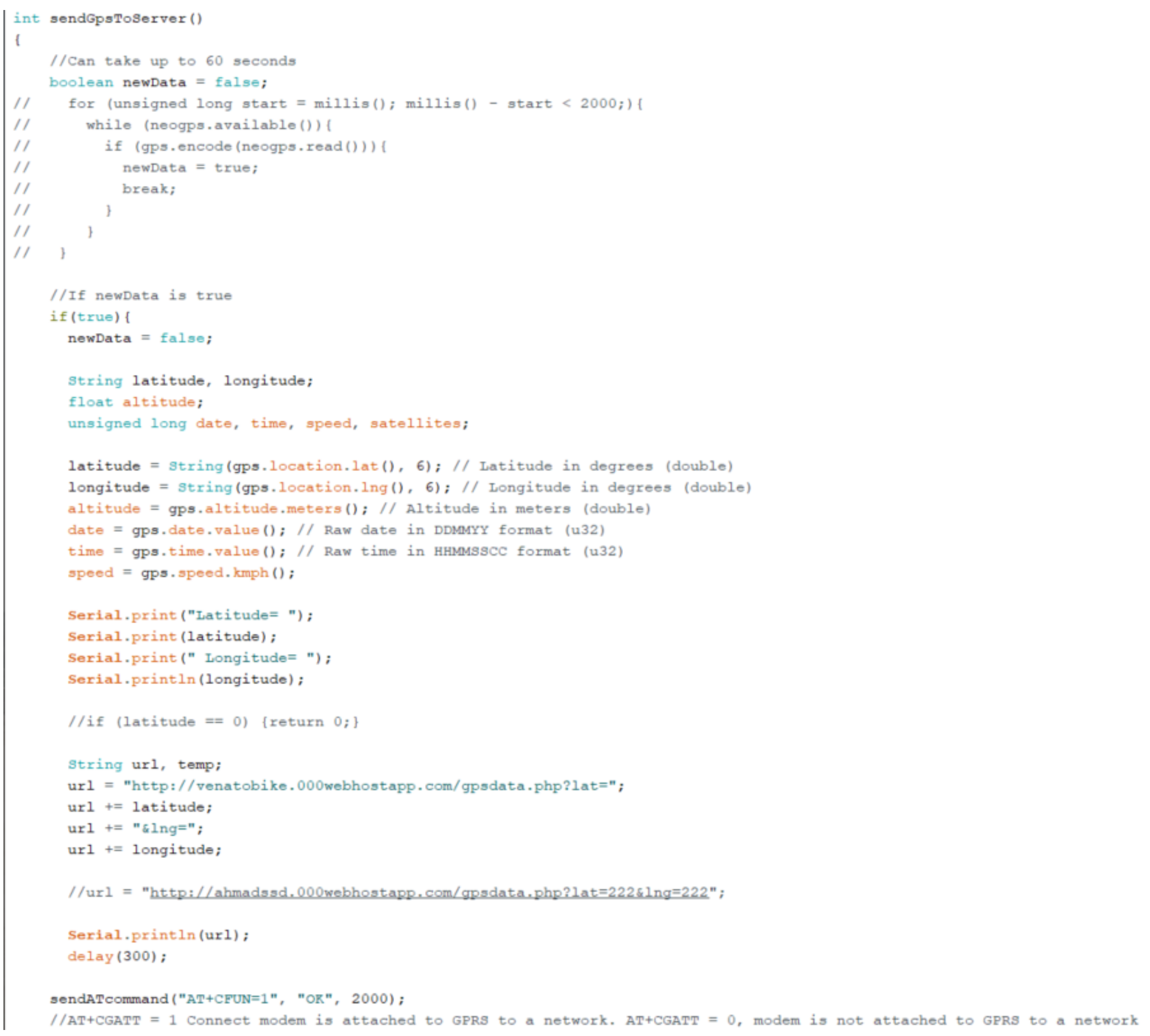
The use case diagram above illustrates how a user will interact with our Venato app system. Upon opening the app, the user must first sign in. This is dependent on the login feature of our app, whereby a user must first be signed up. If they are not signed up, their login is dependent on them receiving their email verification code, which then authorises them to access our app. When a user is logging in, our app is also able to handle invalid credentials (e.g. password) as well as a change password option.

There are other remaining use cases of our app that become available to the user after signing in, which include the ability to access the home screen, viewing their bike location, opening their user profile and finally signing out. These use cases are all dependent on the user first signing into the app, as illustrated in the diagram through the dotted line dependencies. The sign out use case is also dependent on the user opening their user profile, as that is where the sign out button is located.

5. Implementation

5.1 Arduino Code

The GPS/GSM devices we utilised required code in order to display the necessary GPS location data that we needed for our app. We used Arduino's IDE (written in C++) to perform AT commands on the device. Each command had a different function, such as turning on the device, retrieving GPS co-ordinates or to establish a HTTP connection. The full code we implemented utilised the Tiny GPS library in order to facilitate the retrieval of the location of the GPS. Furthermore, in our final implementation the Arduino code would get the latitude and longitude of the device and send that data to our PHP database to store it.

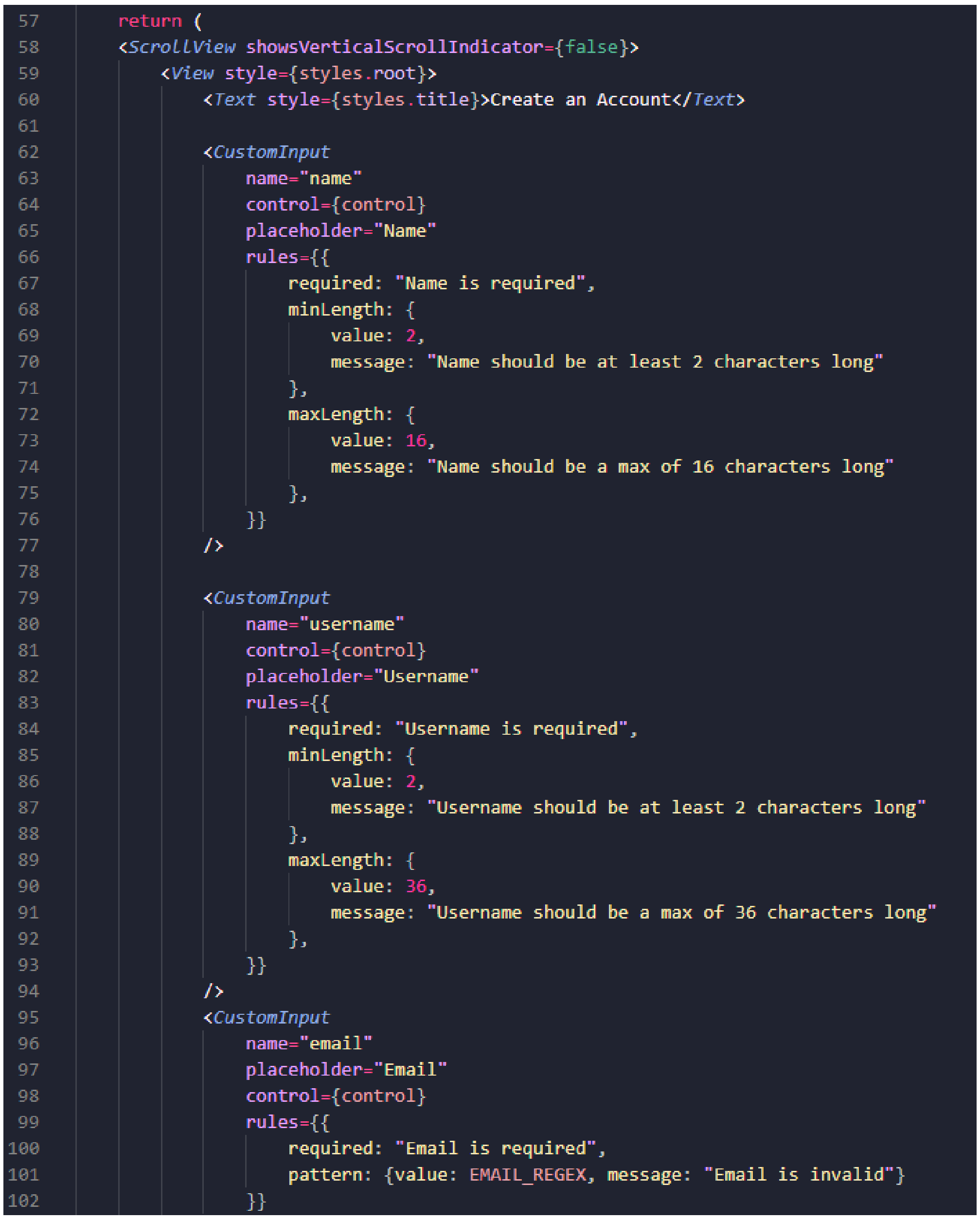


```
int sendGpsToServer()
{
    //Can take up to 60 seconds
    boolean newData = false;
//    for (unsigned long start = millis(); millis() - start < 2000;){
//      while (neogps.available()){
//        if (gps.encode(neogps.read())){
//          newData = true;
//          break;
//        }
//      }
//   }

    //If newData is true
    if(true){
      newData = false;

      String latitude, longitude;
      float altitude;
      unsigned long date, time, speed, satellites;

      latitude = String(gps.location.lat(), 6); // Latitude in degrees (double)
      longitude = String(gps.location.lng(), 6); // Longitude in degrees (double)
      altitude = gps.altitude.meters(); // Altitude in meters (double)
      date = gps.date.value(); // Raw date in DDMMYY format (u32)
      time = gps.time.value(); // Raw time in HHMMSSCC format (u32)
      speed = gps.speed.kmph();

      Serial.print("Latitude= ");
      Serial.print(latitude);
      Serial.print(" Longitude= ");
      Serial.println(longitude);

      //if (latitude == 0) {return 0;}

      String url, temp;
      url = "http://venatobike.000webhostapp.com/gpsdata.php?lat=";
      url += latitude;
      url += "&lng=";
      url += longitude;

      //url = "http://ahmadssd.000webhostapp.com/gpsdata.php?lat=222&lng=222";

      Serial.println(url);
      delay(300);

    sendATcommand("AT+CFUN=1", "OK", 2000);
    //AT+CGATT = 1 Connect modem is attached to GPRS to a network. AT+CGATT = 0, modem is not attached to GPRS to a network
```

5.2 React Native (Part 1 - Authentication)

For authentication purposes we opted for AWS Amplify to be our service which deals with who registers/logs in/sign outs within our application. We provide a 'Sign In' screen where users are able to sign in and then navigate to the main pages of Venato, a 'Sign Up' screen where users are able to create an account with the use of their name, username, email and password. This code for this screen is illustrated below. Once users follow all guidelines and create an account they proceed to a 'Confirm Email' screen where they have to input the verification code that was sent to their provided email in order to confirm their account.

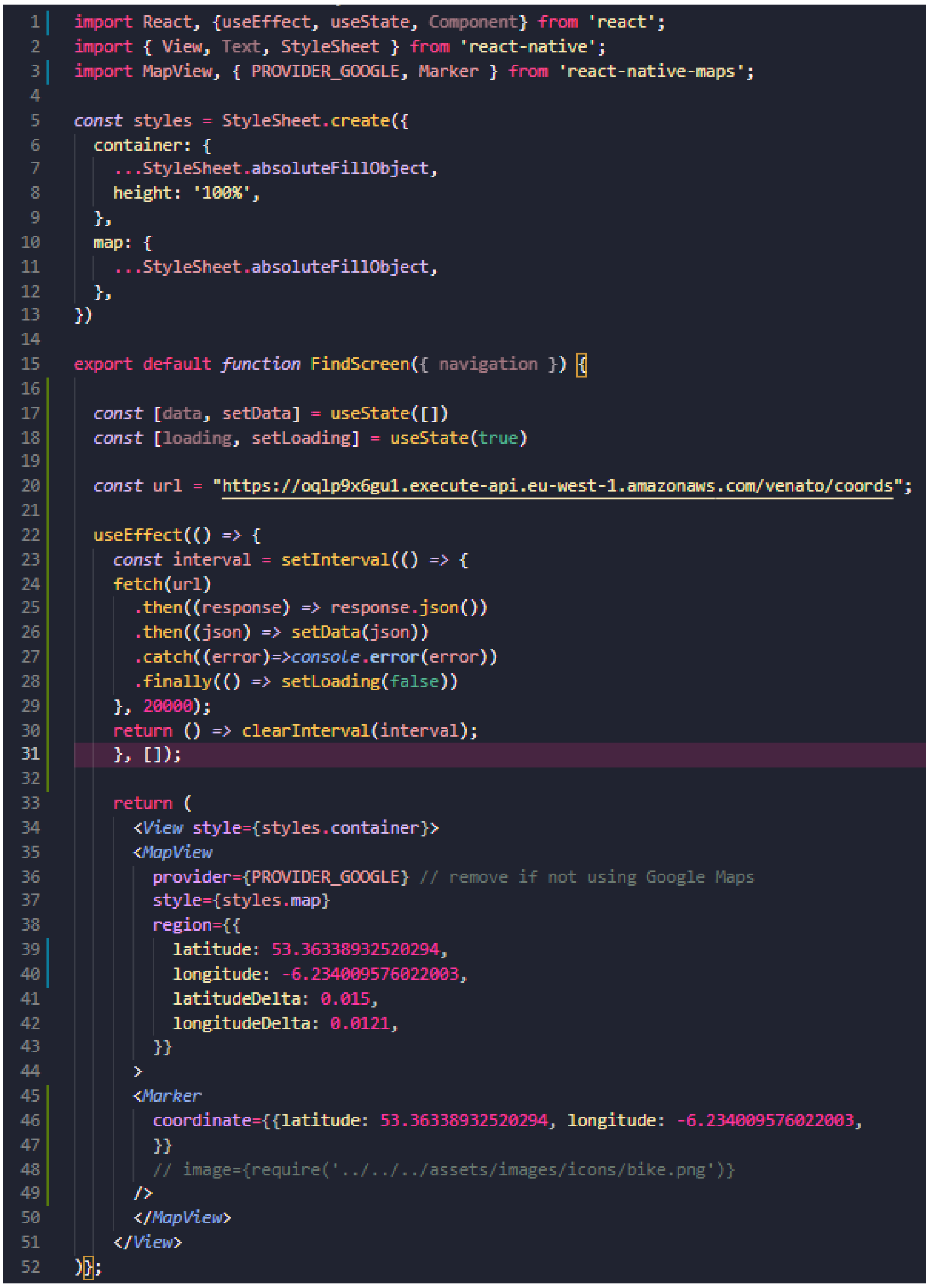
Another feature of authentication we provided is the ability to reset your password with a 'Forgot Password' screen. AWS Amplify has a log of all the users that already exist. Our React Native CLI application in turn handles the UI components as well as the states upon users logging in and out and accounts being created. For the libraries and installations that we required, we used React Native Navigation and specifically Stack Navigation to deal with routing between screens, AUTH as the authentication packages necessary for AWS Amplify and React Hook Forms to handle all the forms we used.



```
    return (
    <ScrollView showsVerticalScrollIndicator={false}>
        <View style={styles.root}>
            <Text style={styles.title}>Create an Account</Text>

            <CustomInput
                name="name"
                control={control}
                placeholder="Name"
                rules={{
                    required: "Name is required",
                    minLength: {
                        value: 2,
                        message: "Name should be at least 2 characters long"
                    },
                    maxLength: {
                        value: 16,
                        message: "Name should be a max of 16 characters long"
                    },
                }}
            />

            <CustomInput
                name="username"
                control={control}
                placeholder="Username"
                rules={{
                    required: "Username is required",
                    minLength: {
                        value: 2,
                        message: "Username should be at least 2 characters long"
                    },
                    maxLength: {
                        value: 36,
                        message: "Username should be a max of 36 characters long"
                    },
                }}
            />
            <CustomInput
                name="email"
                placeholder="Email"
                control={control}
                rules={{
                    required: "Email is required",
                    pattern: {value: EMAIL_REGEX, message: "Email is invalid"}
                }}
```

5.3 React Native (Part 2 - Maps)

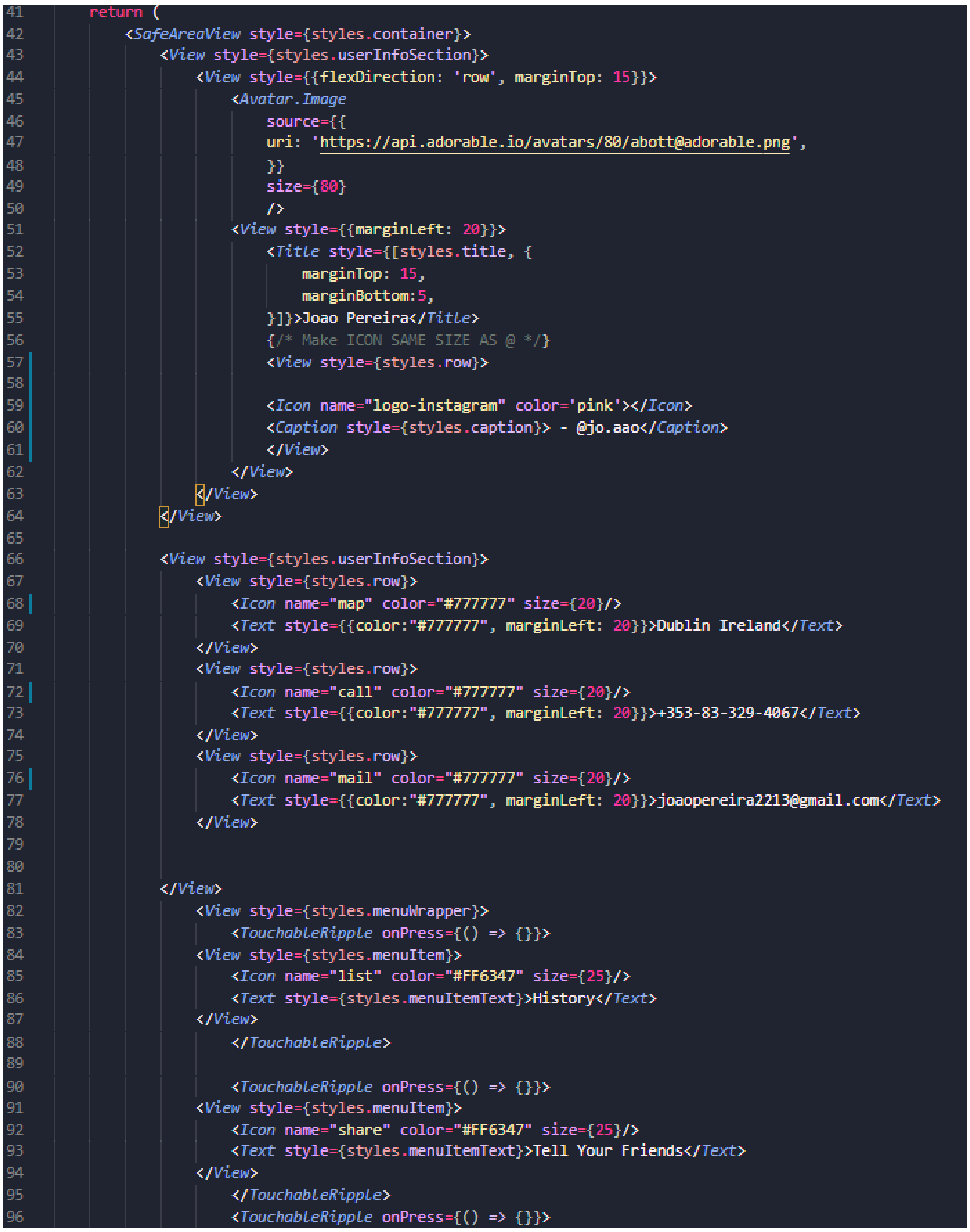
The Venato app’s main feature is the map that displays the current location of a user’s bike. In order to display the map in our React Native CLI app, we chose to use the React Native Maps library. We then integrated Google Maps onto a screen with the use of a generated API Key. Within our map screen, we have a useEffect hook which performs the fetching of the API link and then has the link refresh every twenty seconds. The screen retrieves the API data and displays it on the map within a “<Marker…” tag. However due to device/technical performance issues, our code is not fully up to date to a GPS/GSM device, however due to testing purposes, we instead created a Python script which in some similarity acts like the device in order for the maps to be able to display coordinates on a map, such as the device would when fully optimised.



```
import React, {useEffect, useState, Component} from 'react';
import { View, Text, StyleSheet } from 'react-native';
import MapView, { PROVIDER_GOOGLE, Marker } from 'react-native-maps';

const styles = StyleSheet.create({
  container: {
    ...StyleSheet.absoluteFillObject,
    height: '100%',
  },
  map: {
    ...StyleSheet.absoluteFillObject,
  },
})

export default function FindScreen({ navigation }) {

  const [data, setData] = useState([])
  const [loading, setLoading] = useState(true)

  const url = "https://oqlp9x6gu1.execute-api.eu-west-1.amazonaws.com/venato/coords";

  useEffect(() => {
    const interval = setInterval(() => {
    fetch(url)
      .then((response) => response.json())
      .then((json) => setData(json))
      .catch((error)=>console.error(error))
      .finally(() => setLoading(false))
    }, 20000);
    return () => clearInterval(interval);
    }, []);

    return (
      <View style={styles.container}>
      <MapView
        provider={PROVIDER_GOOGLE} // remove if not using Google Maps
        style={styles.map}
        region={{
          latitude: 53.36338932520294,
          longitude: -6.234009576022003,
          latitudeDelta: 0.015,
          longitudeDelta: 0.0121,
        }}
      >
      <Marker
        coordinate={{latitude: 53.36338932520294, longitude: -6.234009576022003,
        }}
        // image={require('../../../assets/images/icons/bike.png')}
      />
      </MapView>
    </View>
);};
```

5.4 React Native (Part 3 - User Profile, Home Screen)

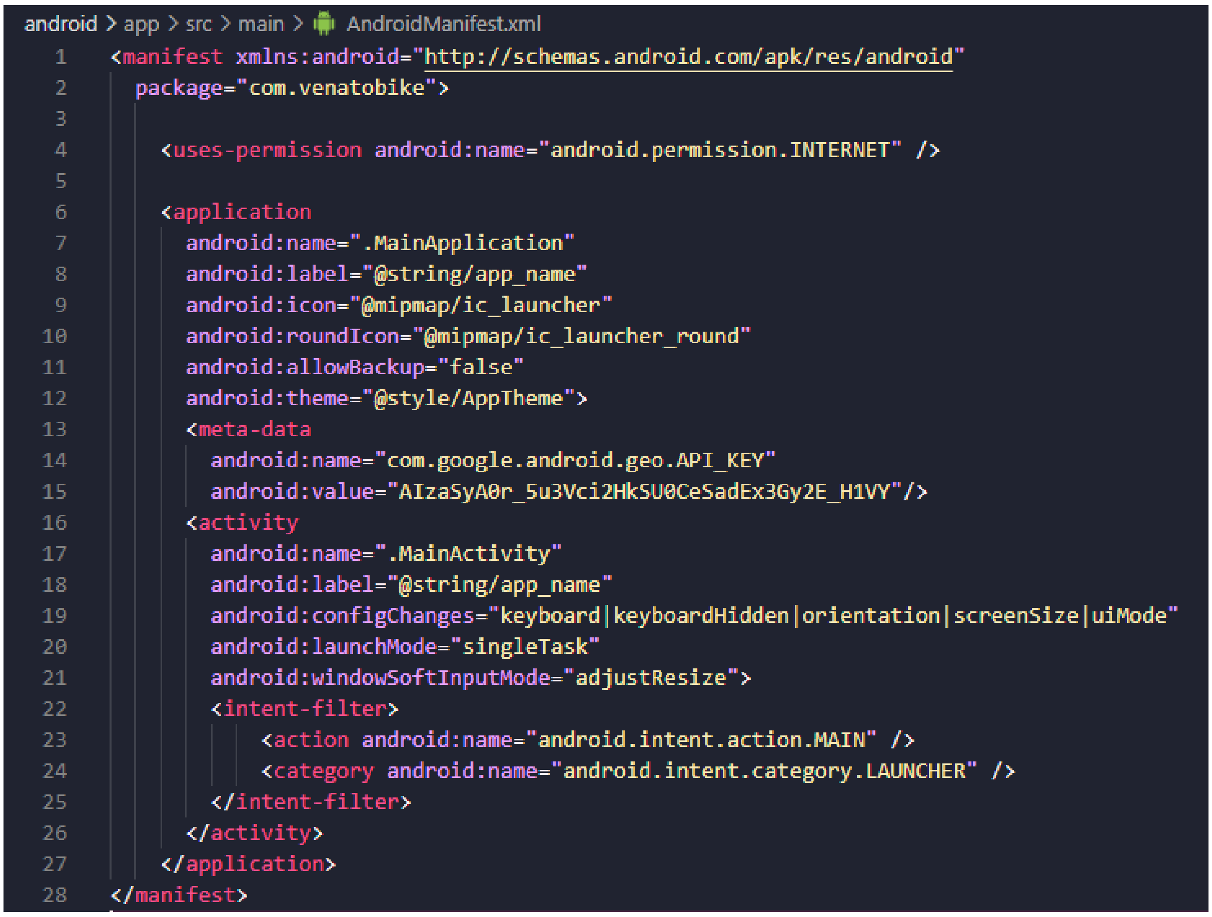
The last section of our application is the ‘User Profile’ and ‘Home’ screen. Both of these screens are prototypes of what the final deliverable would be like if the app was to be released to a market. In our code, the ‘User Profile’, ‘Home’ and ‘Map’ screens are all based on a Bottom Tab Navigator. The ‘Home’ screen consists of a bubble/border with a ‘Welcome to Venato’, a paragraph on how to get started and finally a dummy link to a tutorial. Meanwhile the ‘User Profile’ screen consists of a user’s information such as their name, Instagram handle, location and phone number. In addition, the ‘User Profile’ screen also provides the user with a functional ‘Sign Out’ button. The remaining options include ‘Terms and Conditions’, ‘Help’, ‘Edit Profile’ and ‘Settings’. These are dummy options however, which would only provide functionality if we were to release our app to the market.



```
    return (
        <SafeAreaView style={styles.container}>
            <View style={styles.userInfoSection}>
                <View style={{flexDirection: 'row', marginTop: 15}}>
                    <Avatar.Image
                        source={{
                        uri: 'https://api.adorable.io/avatars/80/abott@adorable.png',
                        }}
                        size={80}
                        />
                    <View style={{marginLeft: 20}}>
                        <Title style={[styles.title, {
                            marginTop: 15,
                            marginBottom:5,
                        }]}>Joao Pereira</Title>
                        {/* Make ICON SAME SIZE AS @ */}
                        <View style={styles.row}>

                        <Icon name="logo-instagram" color='pink'></Icon>
                        <Caption style={styles.caption}> - @jo.aao</Caption>
                        </View>
                    </View>
                </View>
            </View>

            <View style={styles.userInfoSection}>
                <View style={styles.row}>
                    <Icon name="map" color="#777777" size={20}/>
                    <Text style={{color:"#777777", marginLeft: 20}}>Dublin Ireland</Text>
                </View>
                <View style={styles.row}>
                    <Icon name="call" color="#777777" size={20}/>
                    <Text style={{color:"#777777", marginLeft: 20}}>+353-83-329-4067</Text>
                </View>
                <View style={styles.row}>
                    <Icon name="mail" color="#777777" size={20}/>
                    <Text style={{color:"#777777", marginLeft: 20}}>joaopereira2213@gmail.com</Text>
                </View>


            </View>
                <View style={styles.menuWrapper}>
                    <TouchableRipple onPress={() => {}}>
                <View style={styles.menuItem}>
                    <Icon name="list" color="#FF6347" size={25}/>
                    <Text style={styles.menuItemText}>History</Text>
                </View>
                    </TouchableRipple>

                    <TouchableRipple onPress={() => {}}>
                <View style={styles.menuItem}>
                    <Icon name="share" color="#FF6347" size={25}/>
                    <Text style={styles.menuItemText}>Tell Your Friends</Text>
                </View>
                    </TouchableRipple>
                    <TouchableRipple onPress={() => {}}>
```

5.5 Android Studio

Upon installing Android Studio and establishing an emulator to our React-Native CLI front-end, an Android folder was installed within our repo. This Android folder contains several dependencies and important files to ensure the android emulator is up to date with operating alongside our application. This Android folder was not messed around with too much, apart from on certain occasions where we had to implement a few lines of code in order to update the latest packages that were being installed in our app. For example, when React Maps was installed, "AndroidManifest.xml" had to be altered in order to retrieve an API key for the maps to be fully functioning, as illustrated below.



android > app > src > main > AndroidManifest.xml

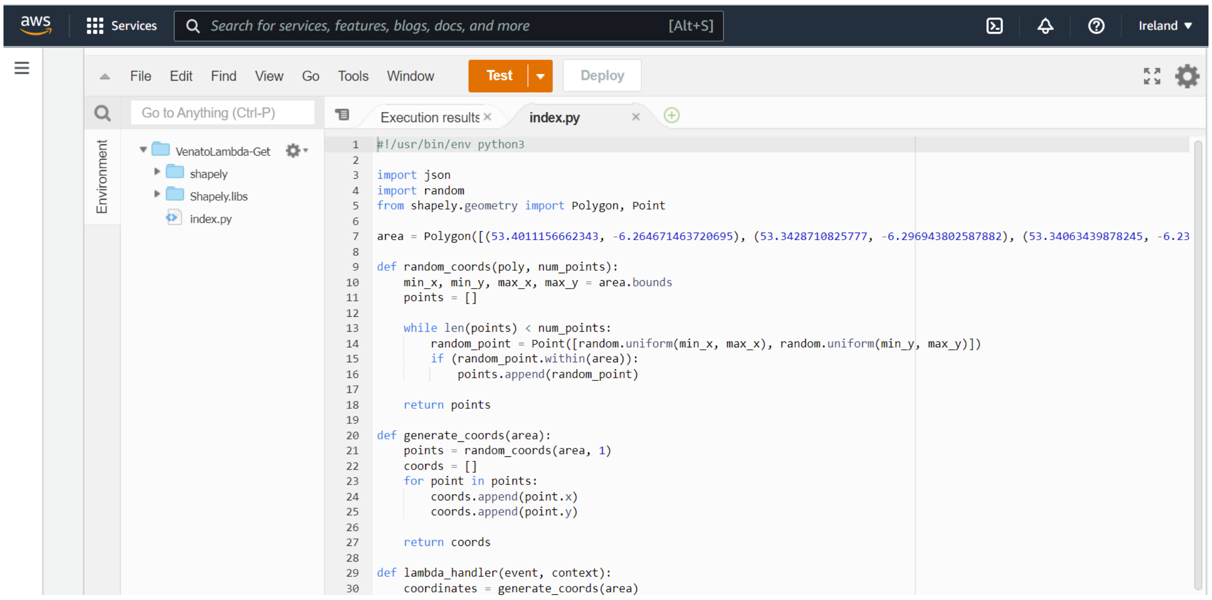
```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.venatobike">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
      android:name=".MainApplication"
      android:label="@string/app_name"
      android:icon="@mipmap/ic_launcher"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:allowBackup="false"
      android:theme="@style/AppTheme">
      <meta-data
        android:name="com.google.android.geo.API_KEY"
        android:value="AIzaSyA0r_5u3Vci2HkSU0CeSadEx3Gy2E_H1VY"/>
      <activity
        android:name=".MainActivity"
        android:label="@string/app_name"
        android:configChanges="keyboard|keyboardHidden|orientation|screenSize|uiMode"
        android:launchMode="singleTask"
        android:windowSoftInputMode="adjustResize">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
    </application>
</manifest>
```

As you can see above, within the "<meta-data…" tag, we had to implement an API key.

In addition, the "buildscript" within build.gradle was altered to support React Maps. Another change to the Android folder was the addition of a fonts file in order to have IonIcons be able to display icons within the application. The rest of the Android code implementations/changes were within gradle files in order to deal with and attempt to fix errors that the application faced. These implementations were often very small.

5.6 AWS Lambda Function

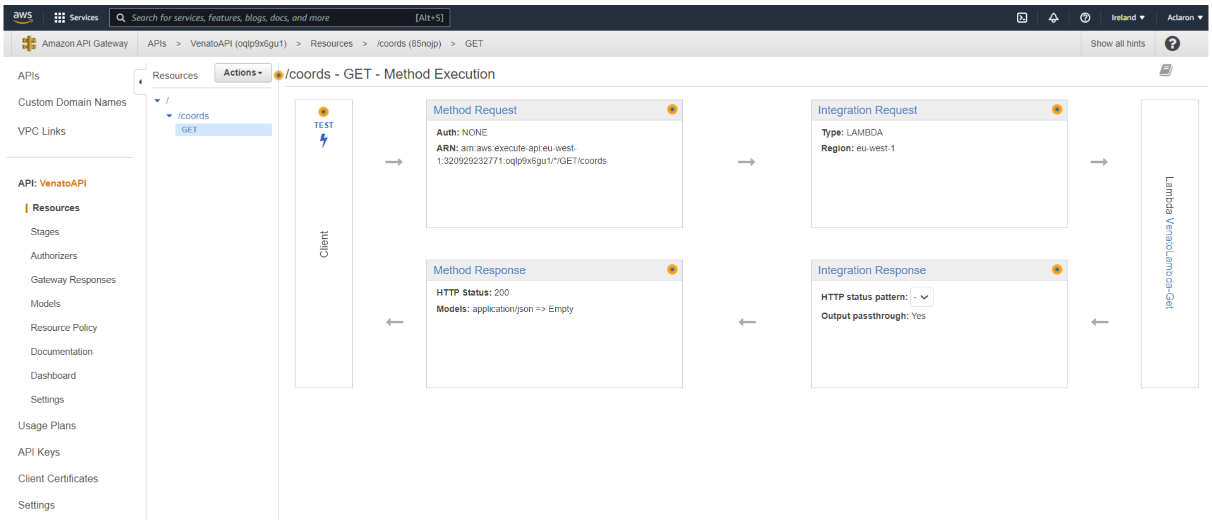
After deciding to write a Python script that could generate random coordinates, we had to find a way to get this coordinate data into React Native. The script was uploaded to the AWS Lambda service and turned into a Lambda function, which can then be called by an API. As seen in the picture below, we also had to upload the packages for the Python module Shapely since this module is an external library in Lambda. When we run this function, we receive a JSON coordinate object with a random latitude and longitude.



```python
#!/usr/bin/env python3

import json
import random
from shapely.geometry import Polygon, Point

area = Polygon([(53.4011156662343, -6.264671463720695), (53.3428710825777, -6.296943802587882), (53.34063439878245, -6.23

def random_coords(poly, num_points):
    min_x, min_y, max_x, max_y = area.bounds
    points = []

    while len(points) < num_points:
        random_point = Point([random.uniform(min_x, max_x), random.uniform(min_y, max_y)])
        if (random_point.within(area)):
            points.append(random_point)

    return points

def generate_coords(area):
    points = random_coords(area, 1)
    coords = []
    for point in points:
        coords.append(point.x)
        coords.append(point.y)

    return coords

def lambda_handler(event, context):
    coordinates = generate_coords(area)
```

5.7 AWS Gateway REST API

We created a REST API using the AWS Gateway API service. We called it VenatoAPI and it has only one method, as seen in the picture below, which is a GET method. When we deploy the API, our React Native code contains a fetch request that calls this GET method. Our Lambda function is integrated with our API so that when the API's GET method is called, the Lambda function is invoked and returns a JSON coordinate object.



/coords - GET - Method Execution

Method Request
Auth: NONE
ARN: arn:aws:execute-api:eu-west-1:320929232771:oqlp9x6gu1/*/GET/coords

Integration Request
Type: LAMBDA
Region: eu-west-1

Method Response
HTTP Status: 200
Models: application/json => Empty

Integration Response
HTTP status pattern: -
Output passthrough: Yes

6. Problems and Solutions

6.1 Device Delays

The device we needed for our project, a Maduino Zero A9G IoT Microcontroller, was out of stock for several months. Eventually, after regularly checking availability, we were able to buy it, and it arrived in mid-January. We immediately met up and got to work on starting our project, unfortunately however, within an hour of starting to use our device, the pins in the micro SIM card connector broke.

We took the device to specialist technicians in a repair shop in order to fix it, however the damage was irreversible, rendering our device useless. We overcame this problem by ordering a new replacement device, however this exact device was again out of stock. We therefore had to find and order a new device, a SIM808 GPS module, which we received in mid-February. We then realised that we had to order a battery and a charger for it, which took another week to arrive. Eventually, by the end of February and after spending €115, we were finally able to begin connecting our device with our existing front-end and back-end setup.

6.2 Device Issues

When we initially discussed our project with our supervisor, Michael Scriney, he told us that sensor projects can involve troubleshooting and unforeseen issues with the hardware. Unfortunately, this advice proved to be incredibly pertinent, as we faced a host of problems with not only the hardware, but our software as well.

Upon initially attempting to retrieve GPS data from our device using a C++ program in the Arduino IDE, we could retrieve all data except the most important piece - the GPS latitude and longitude coordinates. We became incredibly frustrated after unsuccessfully spending several hours making multiple new programs trying to fix this. We eventually decided to try moving the device to a different location, hoping this would solve the problem. Incredibly, we managed to receive GPS coordinate data and seemingly fix the issue after standing in the back corner of Joao's garden. This proved to be a false start however, as our device would randomly choose when to send the coordinates. We exhausted ourselves for hours trying to resolve this issue, but ultimately we realised that it was out of our control and decided to simulate the device through a AWS Lambda Python function.

6.3 AWS Account Permissions

Having decided to simulate our device by creating a Python script that could generate random coordinates, we set about getting the coordinate data into React Native by integrating our script with an AWS Gateway API through the AWS Lambda service. Despite having never used AWS before, we had success with making a Python Lambda function and integrating it with the Gateway API that we created. Unfortunately however, we kept getting a 403 "message: forbidden" error when testing our API through Postman. We created several new test APIs with even the easiest of functions, but every time we ran into this error. Eventually, we decided to create a new AWS account, as research had made us suspect that the 403 error had something to do with permissions. The new account was created in exactly the same way as our original account, yet we managed to successfully deploy the API, perform requests and get the coordinate data through an API fetch in React Native.

6.4 React Native Difficulties

During the development of the front-end application and integration with the back-end, we ran into several problems, most of which were errors from React Native CLI. Some of these errors were, for example, ":app:compileDebugJavaWithJavac FAILED" and "Response error code: 500". These are just some quick examples, however we ran into these issues far too many times to keep count on. They would usually occur when we were trying to install a package or dependency and the app would then fail with errors, or if one piece of code was changed ever so slightly and there was an error, we would then be challenged to fix that issue with unknown resources.

A lot of these errors set us back regarding time, as sometimes we would spend hours to even two days searching for a solution. At one stage, we formed a commit which broke the project depo itself, meaning that we were required to create a new repo and have the project transferred over. We were able to overcome these React Native issues through sheer persistence and constant debugging. We were able to learn from these errors as we got further through our app, meaning that we eventually made less mistakes and were able to more efficiently develop our app.

6.5 Android Studio Problems

Upon the initiation of the project, one of our early goals was to get an Android emulator working in order to test the React Native front-end. This took a week to get up and running as our lack of experience with both the React Native framework and the Android Studio IDE meant that we made several errors throughout the process. We struggled to fix these errors, meaning that we were unable to create a stable connection between the project and the emulator and thus could not move on to designing our app.

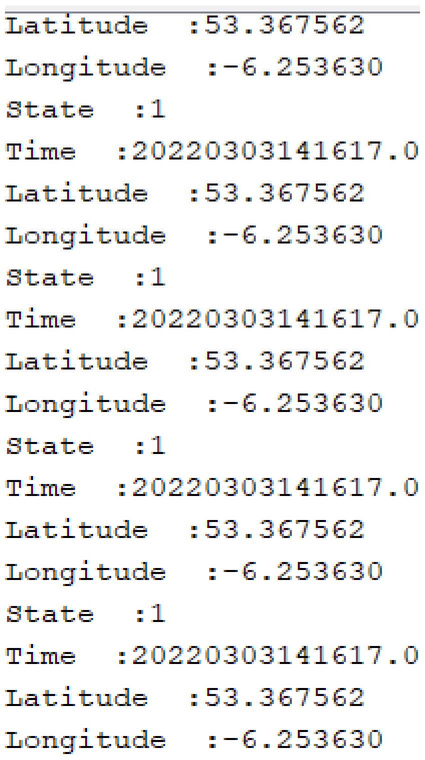
After many frustrating attempts to establish a connection, we finally came up with a potential solution. We decided to create a new project and transfer all of the files and dependencies over, before then deleting all of the Android folders and retrying from the beginning. This solution proved to be a successful one as we were then able to create a stable connection. Furthermore, our debugging efforts had made us more skilled with using Android Studio and React Native, meaning that we were making less errors and were working more efficiently.

7. Testing

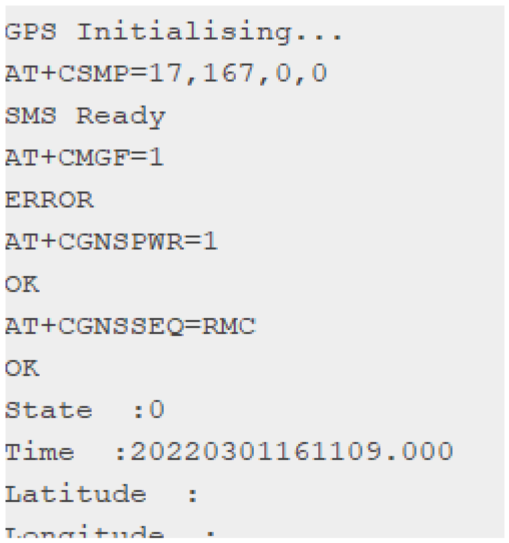
7.1 Unit Testing

We employed unit testing when we were trying to read the data from our GPS device into the Arduino serial monitor. After we had written our Arduino code, we started with the most basic of tests by only attempting to get the time data from the device. When this worked, we incrementally added more data requests, including state, latitude and longitude. We were also only running one test at a time at this point, so once we could receive the data, we moved on to continually running our program.

It was thanks to our unit testing that we realised that we could not rely on our device for data. The image below on the left shows what it looks like in the Arduino serial monitor when our device was working as intended. The image on the right shows what happened most of the time, where the latitude and longitude data is missing. Our program would only get initial coordinate data roughly 10% of the time, but would always eventually stop sending data when we performed continuous testing.



```
Latitude   :53.367562
Longitude   :-6.253630
State   :1
Time   :20220303141617.0
Latitude   :53.367562
Longitude   :-6.253630
State   :1
Time   :20220303141617.0
Latitude   :53.367562
Longitude   :-6.253630
State   :1
Time   :20220303141617.0
Latitude   :53.367562
Longitude   :-6.253630
State   :1
Time   :20220303141617.0
Latitude   :53.367562
Longitude   :-6.253630
```

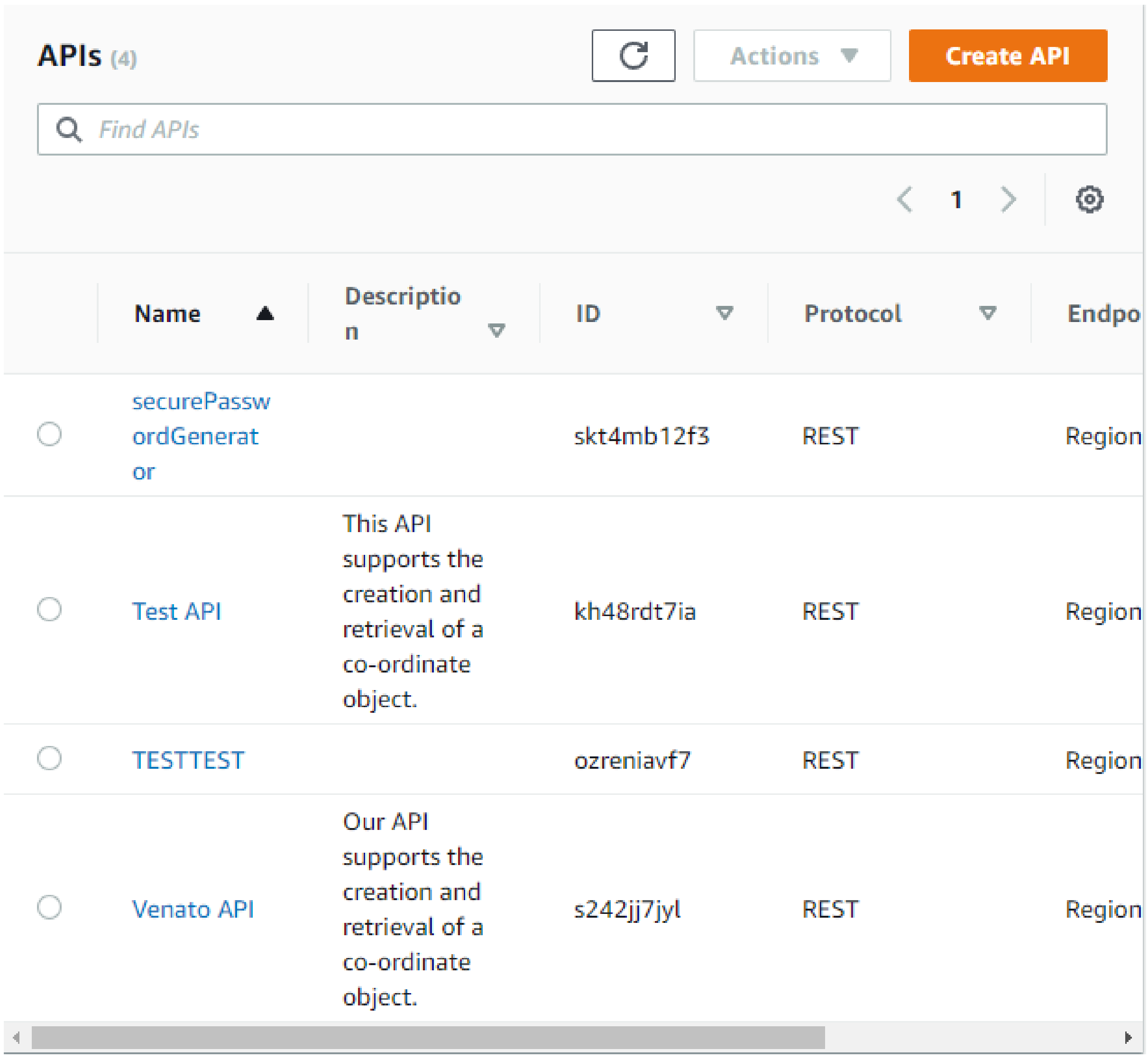


```
GPS Initialising...
AT+CSMP=17,167,0,0
SMS Ready
AT+CMGF=1
ERROR
AT+CGNSPWR=1
OK
AT+CGNSSEQ=RMC
OK
State   :0
Time   :20220301161109.000
Latitude   :
Longitude   :
```

7.2 Integration Testing

Our first use of integration testing was through AWS Amplify and React Native. We performed integration testing by checking if registered users appeared on AWS Amplify/Cognito as confirmed/unconfirmed. Initially users would have the ability to sign up, however Amplify/Cognito was unable to both retrieve their information and process that the register event had even occurred. Once we solved this issue, we had our next problem where all the users appeared to be 'unconfirmed'. This issue was then shortly fixed with a few lines of code and AWS Amplify was integrated with React Native CLI and our testing phase for this stage was terminated.

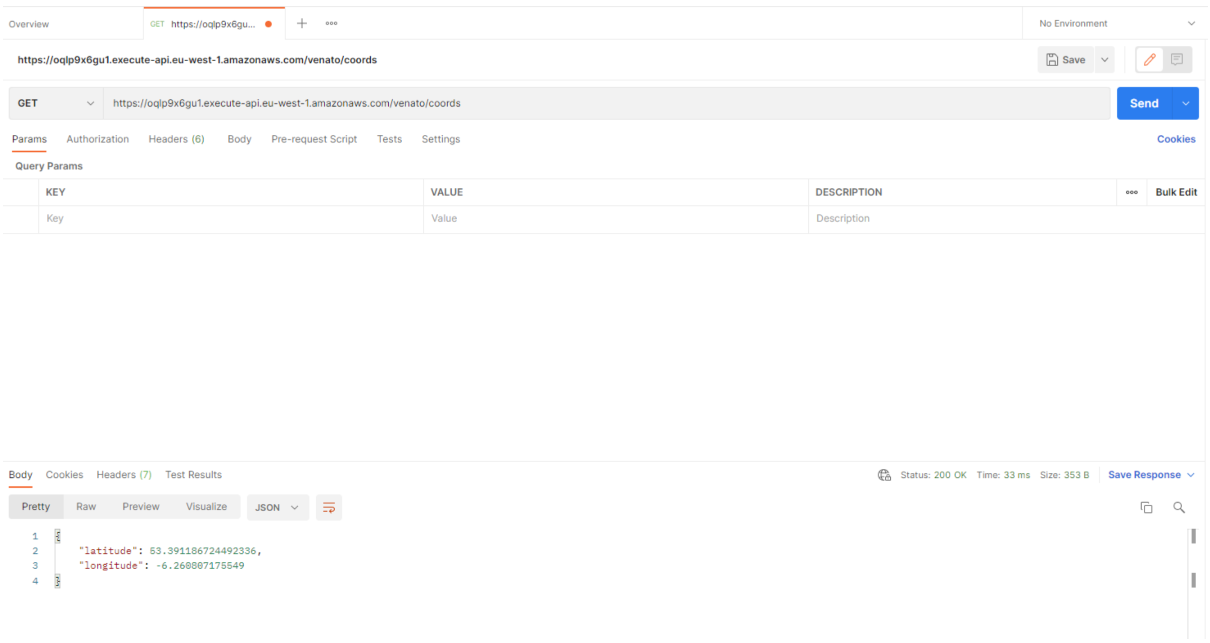
Another form of integration Testing was based on AWS Lambda, Gateway API and React Native. We integrated a Python script with Lambda, creating a Lambda function that sends coordinate data in JSON format to the Gateway API, which is called by a fetch GET request in React Native. Initially, this system seemed quite easy to operate on however it proved to be more complex than we had first thought. We performed Integration Testing by taking each step at a time and ensuring each individual component was performing its purpose under no fault. Once each component was optimised to the required standard, we then pieced them one by one until they were all operating together in the manner that was required. For example, our Lambda function was providing the response we required, however the Gateway API was not communicating with the Lambda function and we would receive error messages within the API. We then fixed this problem and performed multiple tests with the use of different accounts, Lambda functions and Gateway API setups as seen below.



Name	Description	ID	Protocol	Endpo
securePasswordGenerator		skt4mb12f3	REST	Region
Test API	This API supports the creation and retrieval of a co-ordinate object.	kh48rdt7ia	REST	Region
TESTTEST		ozreniavf7	REST	Region
Venato API	Our API supports the creation and retrieval of a co-ordinate object.	s242jj7jyl	REST	Region

7.3 System Testing

We used the Postman API testing client in order to conduct system tests on our AWS Gateway API. Our testing involved hitting our API endpoint with GET requests, as pictured below. Our system testing was also a form of black-box testing, as we were only concerned with validating the API responses rather than what was actually happening behind the API server. The desired response was a JSON object showing randomly generated latitude and longitude coordinates, which can be seen in the response body pictured below. Our system testing allowed us to prove that our AWS Lambda function was fully integrated with our API and ensure that there was no irregularity between them.



https://oqlp9x6gu1.execute-api.eu-west-1.amazonaws.com/venato/coords

GET https://oqlp9x6gu1.execute-api.eu-west-1.amazonaws.com/venato/coords

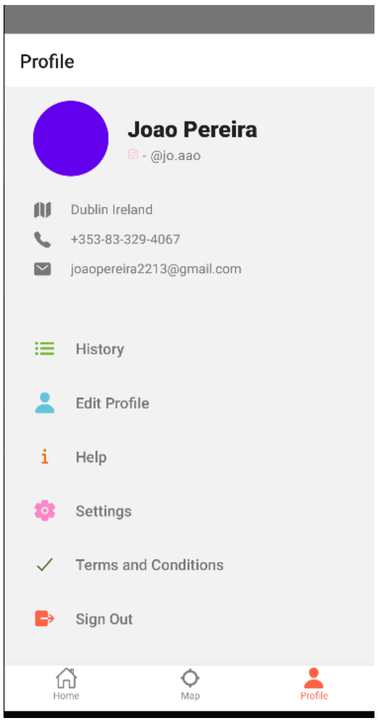
KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 33 ms Size: 353 B

```
{
    "latitude": 63.391186724492336,
    "longitude": -6.260807175549
}
```

7.4 UI Testing

The main goal of our UI testing was to ensure that our Venato app was usable. We wanted to ensure that our app has a sleek UI that would allow our users to perform essential tasks. As a part of our User Testing (see 7.5), we wanted to see how our users would interact with the interface. We needed to ensure that there were no bugs or obstacles that prevented users from intuitively using the app in the way that we intended. As shown in the picture below, we incorporated feedback from users in order to add 3 tabs at the bottom of the screen to easily navigate between screens.



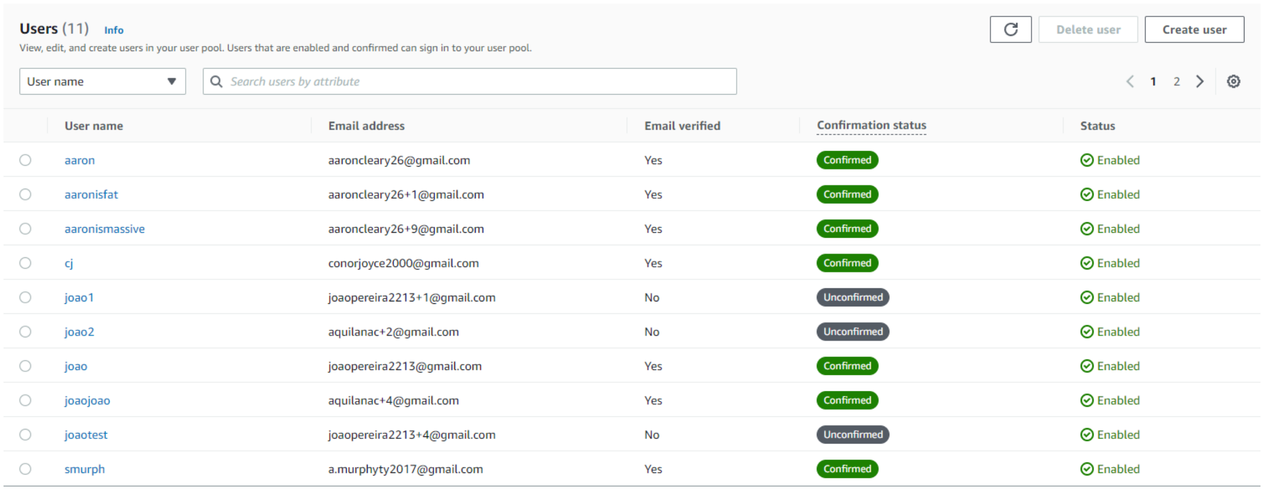
Profile

Joao Pereira
- @jo.aao

Dublin Ireland
+353-83-329-4067
joaopereira2213@gmail.com

7.5 User Testing

We employed user testing after we had finished designing our app. The goal of our user testing was to have our app tested by real users in order to evaluate the usability of the app. We had friends and fellow cyclists, as shown in the picture below, perform certain tasks in realistic conditions, which allowed us to decide whether our app has a functional design that appeals to real users. We gave our user testers basic tasks to perform, such as signing up for the app, interacting with the map and navigating the user profile. The outcome of our user testing was that we were able to verify that we had successfully implemented a functional app that allowed users to intuitively operate it and take advantage of all the app features.



Users (11) Info

View, edit, and create users in your user pool. Users that are enabled and confirmed can sign in to your user pool.

User name	Email address	Email verified	Confirmation status	Status
aaron	aaroncleary26@gmail.com	Yes	Confirmed	Enabled
aaronisfat	aaroncleary26+1@gmail.com	Yes	Confirmed	Enabled
aaronismassive	aaroncleary26+9@gmail.com	Yes	Confirmed	Enabled
cj	conorjoyce2000@gmail.com	Yes	Confirmed	Enabled
joao1	joaopereira2213+1@gmail.com	No	Unconfirmed	Enabled
joao2	aquilanac+2@gmail.com	No	Unconfirmed	Enabled
joao	joaopereira2213@gmail.com	Yes	Confirmed	Enabled
joaojoao	aquilanac+4@gmail.com	Yes	Confirmed	Enabled
joaotest	joaopereira2213+4@gmail.com	No	Unconfirmed	Enabled
smurph	a.murphyty2017@gmail.com	Yes	Confirmed	Enabled