Vicinity
Technical Specification - CA400
Friday, 15th May 2020

Eimhin Dunne 16386406 Adam McElroy 16345753 Supervisor: Paul Clarke

Table of Contents

1.1 Overview

Vicinity is an Android application which enables users to find new places to visit that they otherwise may never have known about. The goal of the application is to take some basic information from a user such as what sort of places / activities they are interested in and from this construct daily recommendations of places which fall into their chosen categories and display these on their homepage. From here they can also select a certain distance radius which they want these places to be within and refresh their recommendations within this radius. Recommendations are refreshed daily at 7:00 am so users can wake up to a new set of places. Besides the home page users can also manually filter what type of places they want to search for. There is a filter page which contains every kind of activity the app has to offer and also a categories page where a user can pick from 10 categories and find only places which fall under this.

1.2 Motivation

Both members of the team moved to Dublin in first year from the country not knowing much about Dublin and the things to do there. In the country there isn't much to do and you usually know most of what there is to do in your area. Dublin is the complete opposite with a massive amount to do and you get overwhelmed with the choice. We wanted to make an app that let users find the things they are interested in in their area. This extended into the rest of the country and we ended up finding new things in our area that we didn't know existed.

1.3 Glossary

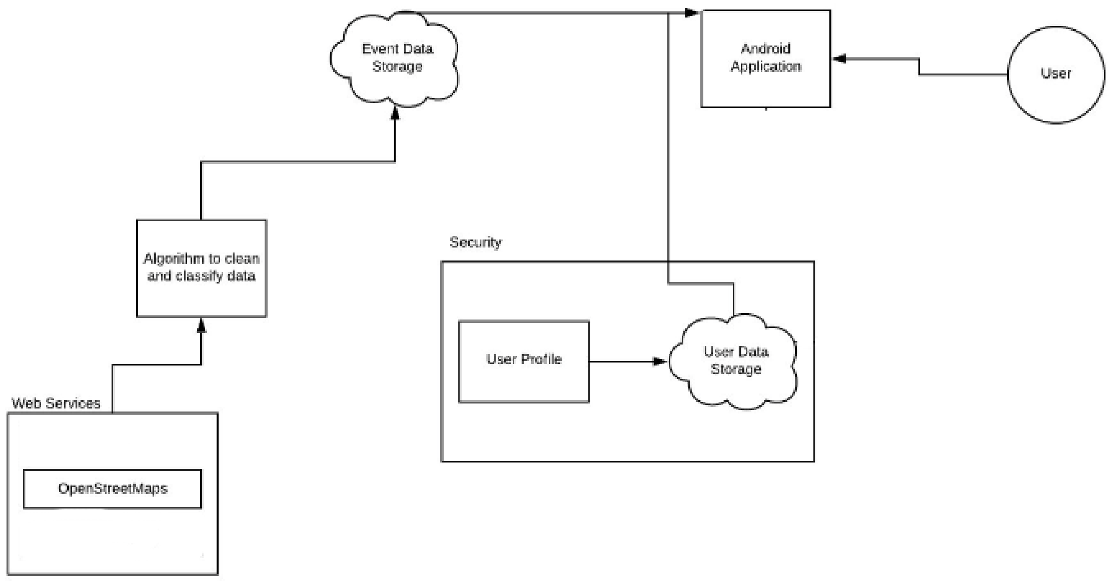
Android Studio - IDE for programming Android apps in Java.
API - Application Programming Interface
UI - User Interface
JSON - JavaScript Object Notation
Git - Version Control System Used
Cloud Functions - Cloud based JavaScript functions which can be triggered by certain events.
Firestore - Data Storage Service provided by Firebase
Firebase - Google's Backend as a Service
NLP - Natural Language Processing
IDE - Integrated Development Environment

1.4 Research

Before the project was approved we did a lot of research on the technologies we could use, as well as similar apps to Vicinity. We found the closest thing to be google maps but this came up short on many of the things we wanted to achieve with Vicinity. The main thing was finding things you didn't know that exist. To find things on google maps you have to have an idea what you are looking for.
When we were happy this was an original idea we had to make sure there was enough technical complexity to make it a project we could possibly do well in. We planned to use firebase as our database and the backend to the app. We also planned on making an android app as well as a web app. A recommender system was also something we thought we would need to include. We both had an interest in using this project as an opportunity to learn about machine learning so we wanted to include a NLP to determine how positive a review was.

Sadly not everything we planned to include was implemented fully but the components were still learned about or partly implemented in the process of the project. Time restraints prevented us from finishing.

2. System Architecture



2.1 Users

Most of the users tasks are automatically done for them by the app. It retrieves data for them and does most of its functionality in the backend. Desired data is then displayed to the users in an easy to read list. The users can make some contributions however. They are allowed to display reviews on different events which can then be seen by other users to better help them understand about the place they are viewing. Users interests are also anonymously used to help make recommendations to other users by comparing their interests to the current users.

2.2 Database and Tools

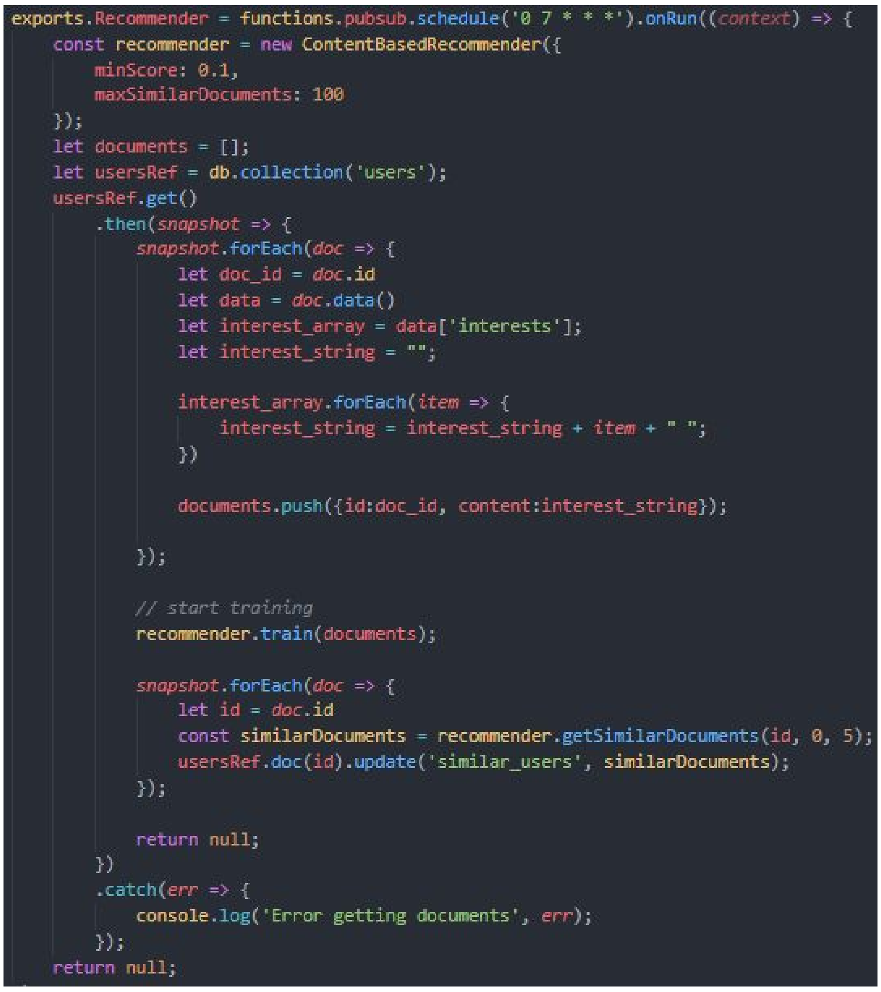
We decided to use Cloud Firestore as our datastore for the app. Firestore integrates seamlessly with Android Studio so it seemed like the best option for us. Firestore has a unique way of storing data but makes it very easy to work with. It consists of collections, documents and fields. Collections are a group of documents. Documents contain information about a single thing and this information was stored in fields. For example we had a collection called "Restaurants". Each document in this collection was an individual restaurant. The fields of each restaurant were data about it such as ID, address, longitude, latitude. These fields could then be queried from within the app.

2.3 Technologies Used

2.3.1 Recommender System

We used a simple recommender system in order to generate new recommendations each day for every user. What the system did was take every user's set of interests and turn them into a large string of interests. From this it computed a similarity score between each sentence for each user. From this we were able to set a threshold of how similar we wanted users to be in order to consider the sufficiently similar. From this we took the top 5 results and assigned these as “similar_users” under each user in Firestore. Their interests were used as the categories to recommend each day so the current user may see some different types of places appearing in their feed.

In order to execute this recommender function daily we used Firebase Cloud Functions. This allowed us to write a node.js script which made the recommendations. From this we could add a schedule to it using crontab syntax which is a method of scheduling events to be triggered at a specific time of the day. We used 7:00 am. This function was then deployed to our Firebase project so it could perform its task in our instance of Firestore.

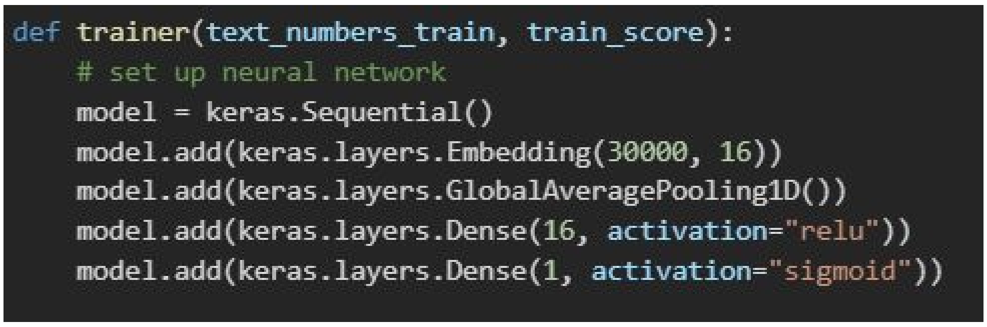


```
exports.Recommender = functions.pubsub.schedule('0 7 * * *').onRun((context) => {
    const recommender = new ContentBasedRecommender({
        minScore: 0.1,
        maxSimilarDocuments: 100
    });
    let documents = [];
    let usersRef = db.collection('users');
    usersRef.get()
        .then(snapshot => {
            snapshot.forEach(doc => {
                let doc_id = doc.id
                let data = doc.data()
                let interest_array = data['interests'];
                let interest_string = "";

                interest_array.forEach(item => {
                    interest_string = interest_string + item + " ";
                })

                documents.push({id:doc_id, content:interest_string});

            });

            // start training
            recommender.train(documents);

            snapshot.forEach(doc => {
                let id = doc.id
                const similarDocuments = recommender.getSimilarDocuments(id, 0, 5);
                usersRef.doc(id).update('similar_users', similarDocuments);
            });

            return null;
        })
        .catch(err => {
            console.log('Error getting documents', err);
        });
    return null;
```

2.3.2 Natural Language Processing

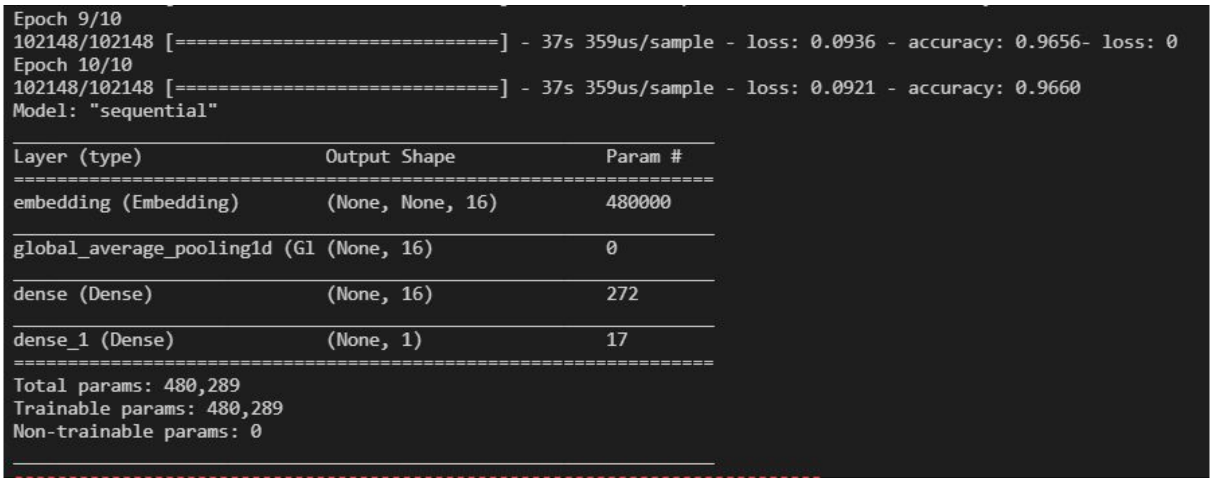
To generate more usable data from reviews we decided to use a NLP approach to determine if the review was positive or negative. This would give an idea if the event was positive or negative and could be used in the recommender system as well as when data was filtered to return higher rated events before the lower rated ones. To do this python was used with tensorflow to build and train the model. It was trained using 500,000 hotel reviews and when trained was returning an accuracy over 94% when tested against unseen data. The NLP was never fully implemented with the app due to version errors and time restraints and is explained in more detail in section 4.3 Problems and Resolutions.

The model:



```
def trainer(text_numbers_train, train_score):
    # set up neural network
    model = keras.Sequential()
    model.add(keras.layers.Embedding(30000, 16))
    model.add(keras.layers.GlobalAveragePooling1D())
    model.add(keras.layers.Dense(16, activation="relu"))
    model.add(keras.layers.Dense(1, activation="sigmoid"))
```

The output:



```
Epoch 9/10
102148/102148 [==============================] - 37s 359us/sample - loss: 0.0936 - accuracy: 0.9656- loss: 0
Epoch 10/10
102148/102148 [==============================] - 37s 359us/sample - loss: 0.0921 - accuracy: 0.9660
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 16)          480000
_________________________________________________________________
global_average_pooling1d (Gl (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 16)                272
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 480,289
Trainable params: 480,289
Non-trainable params: 0
_________________________________________________________________
```

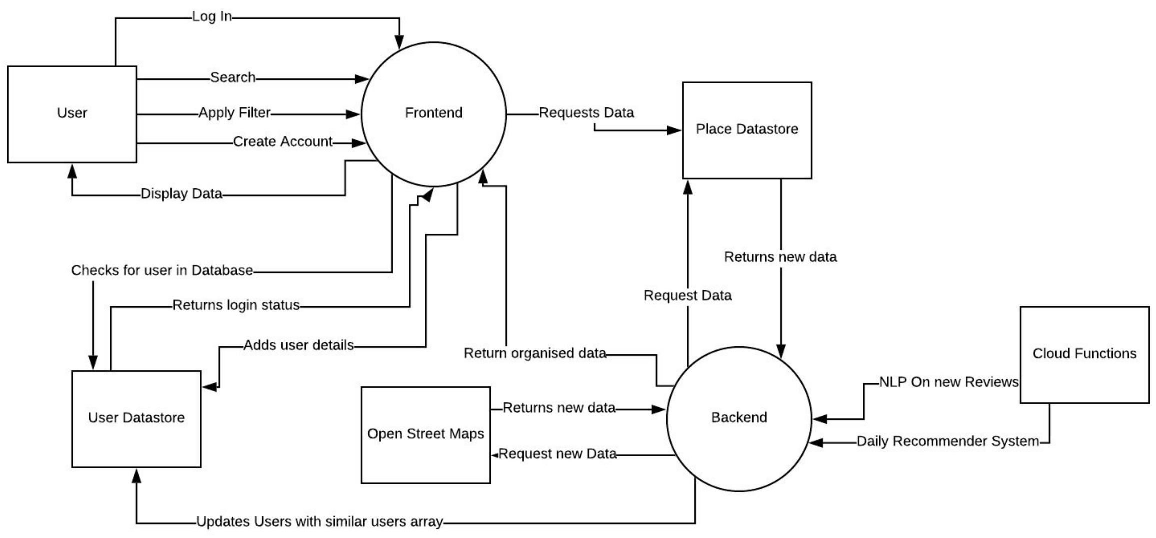
2.3.3 Android Studio

We decided to build the app using android studio as we had previous experience with it. We would have liked to make it as an ios app as well but we did not have access to a Mac, so decided to stick to the one android app. Android studio allowed for testing on a virtual android device, this worked to our favour as not both members of the team had access to an android phone. Android studio was also a prefered IDE as it allowed seamless integration with firestore (2.3.4).

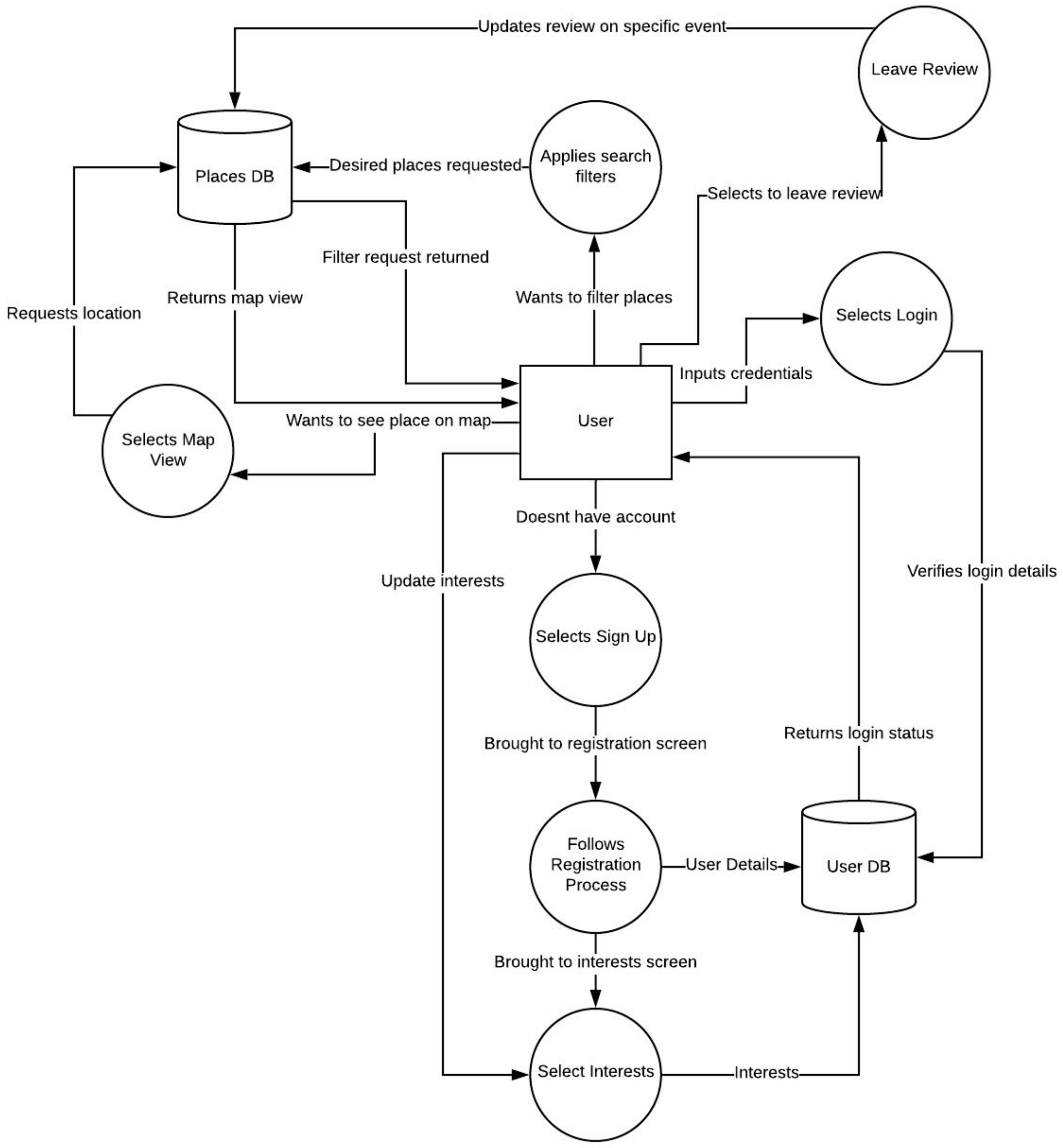
2.3.4 Geohashing

When working with firestore we found that geohashes were the recommended solution to querying the data, this can be seen under problems and resolutions (4.5 - Filtering map by distance). Both coordinates are taken and turned into one geohash. The difference with geohash and coordinates are coordinates give a point where geohash gives a square bounding box that the points fit into. The longer the geohash the more precise the location is. Each letter that gets added to the box makes the box 32 times smaller. These can be 0 to 9 and A to Z, excl "A", "I", "L" and "O".

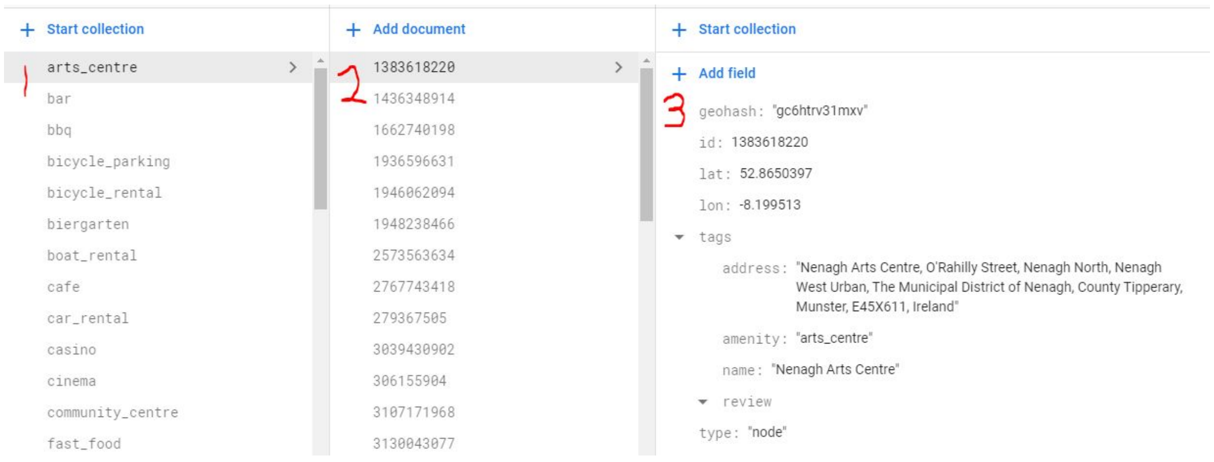
3.1 Context Diagram



3.2 Data Flow Diagram



3.3 Firestore Layout



+ Start collection

```
arts_centre
bar
bbq
bicycle_parking
bicycle_rental
biergarten
boat_rental
cafe
car_rental
casino
cinema
community_centre
fast_food
```

+ Add document

```
1383618220
1436348914
1662740198
1936596631
1946062094
1948238466
2573563634
2767743418
279367505
3039430902
306155904
3107171968
3130043077
```

+ Start collection

+ Add field

```
geohash: "gc6htrv31mxv"
id: 1383618220
lat: 52.8650397
lon: -8.199513
▾ tags
    address: "Nenagh Arts Centre, O'Rahilly Street, Nenagh North, Nenagh
              West Urban, The Municipal District of Nenagh, County Tipperary,
              Munster, E45X611, Ireland"
    amenity: "arts_centre"
    name: "Nenagh Arts Centre"
  ▾ review
type: "node"
```

Above is our Cloud Firestore layout. As mentioned previously in the document Firestore is broken up into Collections, Documents and Fields. The number 1 column in the image is the collections section. This is a snippet of a few of the collections we have gathered from OpenStreetMaps. Each of these flows into the number 2 column which is all of the documents contained in that collection. When retrieving the data from OpenStreetMap each and every place is given an ID. This is what we used to ID each document in Firestore. Finally in column number 3 we can see the fields for the documents. Some of this is data which was retrieved from OpenStreetMap such as id, lat, lon. Others we added ourselves such as geohash which we used to better filter within distances.

4. Problems and Resolution

4.1 - Missing Data

There were many cases of inconsistent data when retrieving it from OpenStreetMap. Every place on the map is guaranteed to have an ID, latitude and longitude. But then there was data assigned to many places but not all such as name, type, website URL etc. We wanted to use names on the display of each event so we had to overcome this. Address was another field present in essentially most of the places retrieved besides about 20 of the total 18,000. We noticed how the name of every place was actually part of the address so we wrote a Namify() function to extract names from addresses which helped in displaying names. This is just one example of a few we had to overcome with inconsistent data. We also chose to only take the fields which we felt were needed. Some places had vast amounts of data most of which we didn't need so we cleaned this before adding it to our datastore.

4.2 - Cloud Functions

When using Cloud functions we encountered a regional problem. Cloud Functions and Cloud Firestore are both connected to different servers but in order to work together their regions must match. When creating our Firestore project the server we used was US-East as the EU was not available at the time. Later on in the project lifecycle when creating the Cloud functions to run on a schedule we attempted to use the EU server to run them. However this caused our functions to break and after a few days spent troubleshooting we discovered that they needed to match the Firestore servers. As a result the function takes approximately 500ms longer to execute each day but we didn't think of this as an issue as it is not a function which is called by the user who expects a response quickly. It is called in the background daily and they don't have to have the app running at the time so the 500ms isn't an issue.

4.3 - Natural Language Processing

When the model was trained in python using tensorflow it was saved with the .h5 extension. This is how tensorflow expects their models to be saved. To use this model that we had trained it had to be converted to a .tflite file. To do this should be an easy task and the code to do so can be seen at (2020-ca400-dunnee49-mcelroa9\src\nlp\nlp_training\convertToTfLite.py). When the .h5 file was tried to convert there were errors with multiple things but the main thing was version errors with python, tensorflow and pip as well other modules. Many hours were spent trying to resolve these issues but it was never resolved due to time restraints.

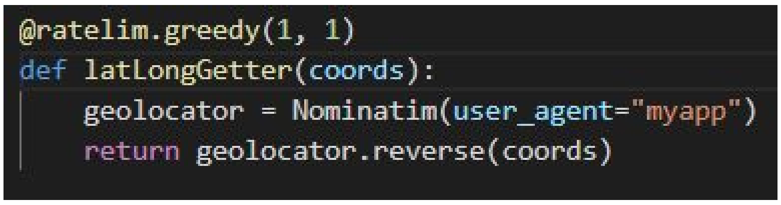
A work around we had found was to find a tflite file online already trained for determining if a review was positive or negative but because of time restraints and exams this was never done.

```
ValueError: Input 0 of node sequential/embedding/embedding_lookup was
passed float from
sequential/embedding/embedding_lookup/Read/ReadVariableOp/resource:0
incompatible with expected resource
```

4.4 - Getting addresses from coordinates

When working with OSM it was easy to return a lot of data but usually the data was missing the majority of fields. We wanted to add to the data and one of the things we noticed was every event had a longitude and latitude. This meant we could look up addresses based on these coordinates. To do this we used a module called Nominatim. It worked perfectly when you were only searching for a few coordinates for addresses but when we were looking for thousands at a time it would restrict access and you would have to wait a few hours before you could run it again. To resolve this we used dictionaries to store the addresses and would only use Nominatim when a new coordinate was needed.

This worked fine but was very manual as you kept having to run the code and waiting for it to time out or throw an error when several new coordinates were searched for after each other. The solution to this was to slow down the API calls to Nominatim using ratelim and calling one coordinate every one second. This slowed down the processes but resolved the problem and we just let it run for several hours.

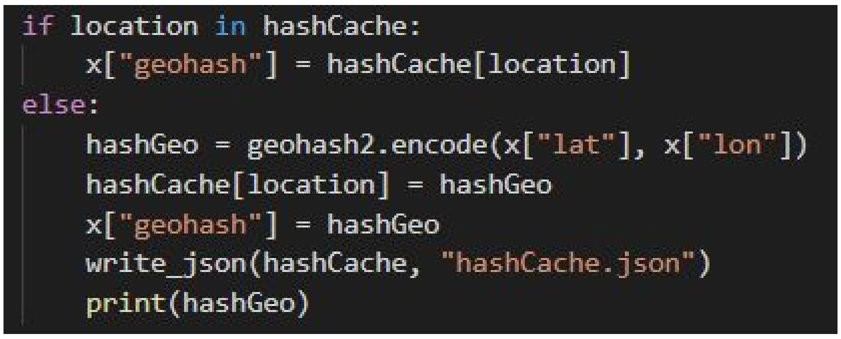


```python
@ratelim.greedy(1, 1)
def latLongGetter(coords):
    geolocator = Nominatim(user_agent="myapp")
    return geolocator.reverse(coords)
```

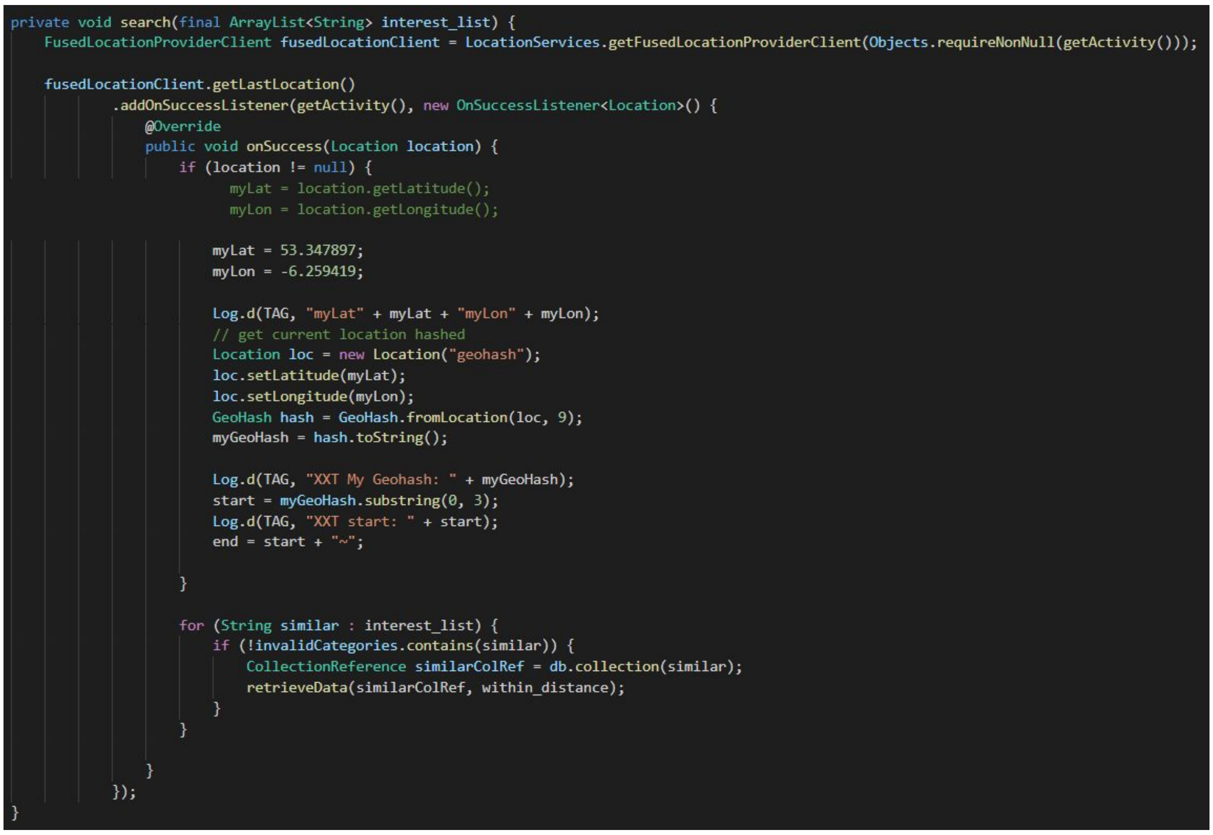
4.5 - Filtering map by distance

Our original plan to filter the map by distance was to draw a box and filter between two X and two Y coordinates which would work if we had used a different database but firestore did not allow this. Firestore does not allow this as they want to be just as quick filtering through small and large amounts of data. This means the ability to use more than one greater than or less than sign is not available. We looked into fixes for this and found geohash was the most efficient way to solve this problem. This created a new problem as we already had all our data in the database but none of it

had geohashes. We decided to delete the events from the database and reupload the events from OSM with the geohash code and used this as an opportunity to clean the data more before uploading it.
Geohashes are square boxes so to find events in a radius of eg. 5km you filter the database to return these boxes that are more than 5km from the server side and then calculate the distance from your current position to each point on the client side. We found this to work well and it was what firebase recommended.



```python
if location in hashCache:
    x["geohash"] = hashCache[location]
else:
    hashGeo = geohash2.encode(x["lat"], x["lon"])
    hashCache[location] = hashGeo
    x["geohash"] = hashGeo
    write_json(hashCache, "hashCache.json")
    print(hashGeo)
```



```java
private void search(final ArrayList<String> interest_list) {
    FusedLocationProviderClient fusedLocationClient = LocationServices.getFusedLocationProviderClient(Objects.requireNonNull(getActivity()));

    fusedLocationClient.getLastLocation()
            .addOnSuccessListener(getActivity(), new OnSuccessListener<Location>() {
                @Override
                public void onSuccess(Location location) {
                    if (location != null) {
                          myLat = location.getLatitude();
                          myLon = location.getLongitude();

                        myLat = 53.347897;
                        myLon = -6.259419;

                        Log.d(TAG, "myLat" + myLat + "myLon" + myLon);
                        // get current location hashed
                        Location loc = new Location("geohash");
                        loc.setLatitude(myLat);
                        loc.setLongitude(myLon);
                        GeoHash hash = GeoHash.fromLocation(loc, 9);
                        myGeoHash = hash.toString();

                        Log.d(TAG, "XXT My Geohash: " + myGeoHash);
                        start = myGeoHash.substring(0, 3);
                        Log.d(TAG, "XXT start: " + start);
                        end = start + "~";

                    }

                    for (String similar : interest_list) {
                        if (!invalidCategories.contains(similar)) {
                            CollectionReference similarColRef = db.collection(similar);
                            retrieveData(similarColRef, within_distance);
                        }
                    }

                }
            });
}
```

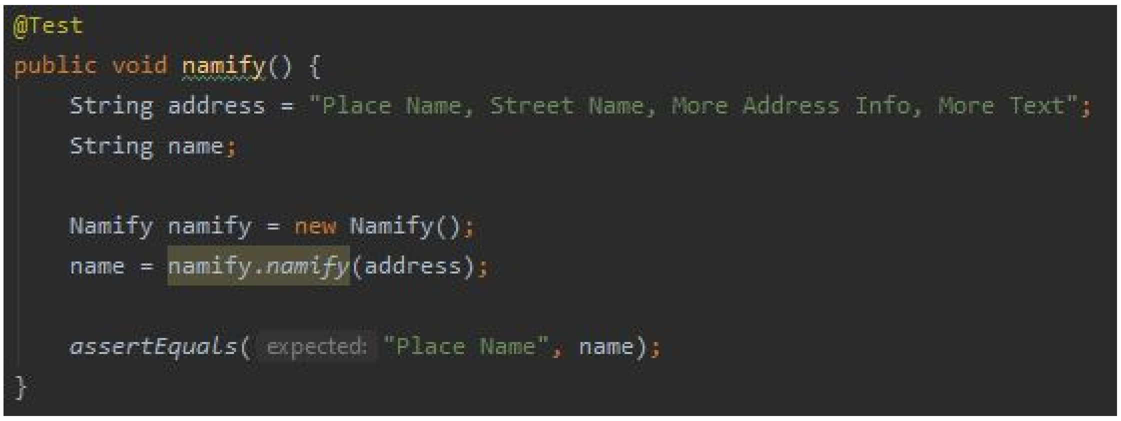
5. Future work
The project is still a work in progress and we would like to continue working on it into the future. We would like to add the ability to have friends in the app as well as having built in messaging and event sharing. These features were not implemented as they would have needed ethical approval which we left too late to try and get.
The other feature we want to add is user levels to gamify Vicinity. This feature will add the ability for leveled up users to add and update events including tags and other information on events that might

be needed. We decided to leave this out for the project as we thought it did not have the technical difficulty as the user recommender system and the NLP.
There are two things we didn't get to finish during the project that we would like to continue working on after, the Web app and the NLP.

6. Testing

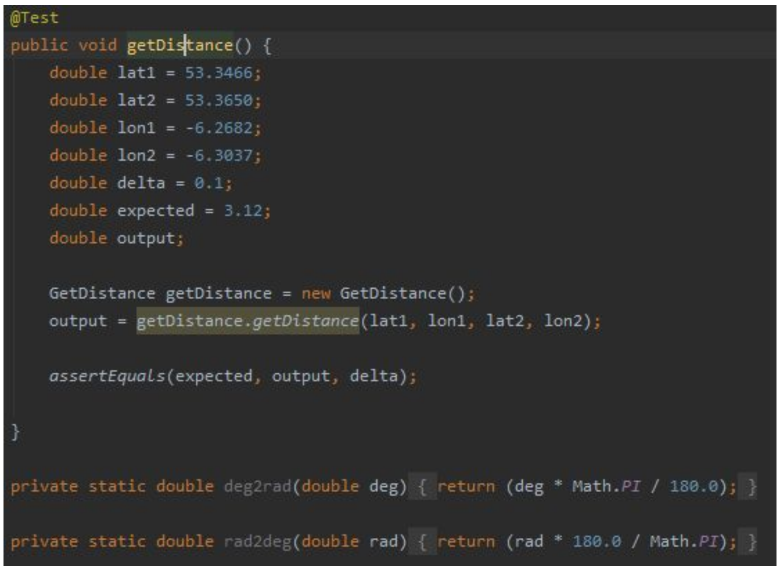
1. Unit Testing

We carried out unit tests as we developed our app to ensure certain components were working before committing them. This allowed us to have confidence that each individual part was working before proceeding with adding new features as we did not want to add a lot of features and try to run the app only to find that a lot of the features were not working. Unit tests enabled us to validate that the application was performing as expected without having to manually crawl through the app to test the point which was in development. This saved us a lot of time. We have included some examples of our unit tests below.



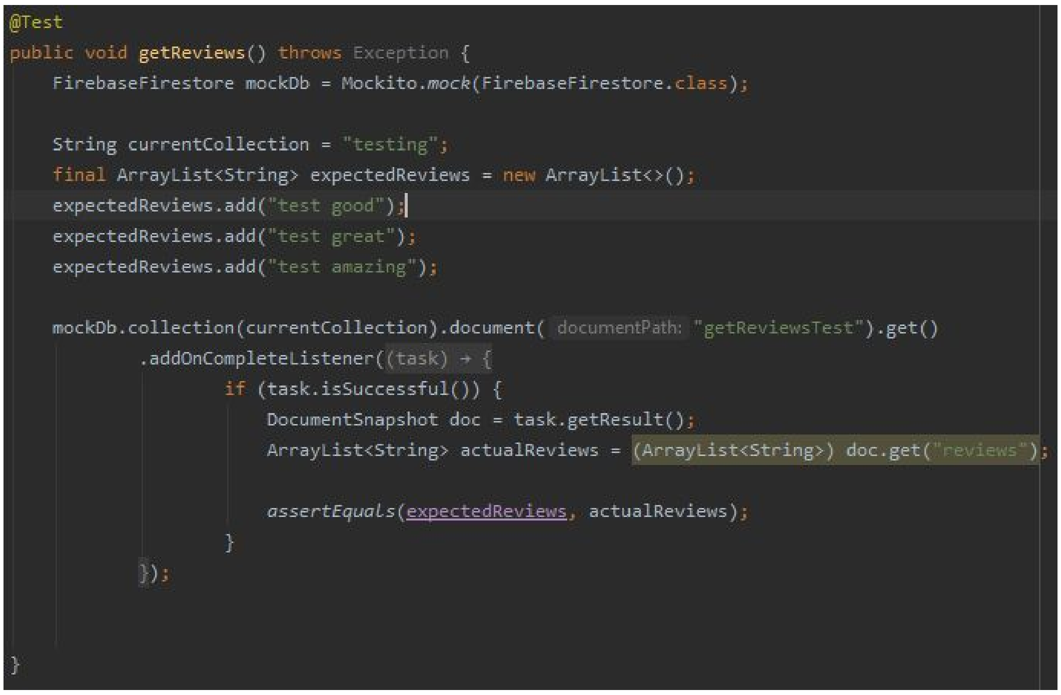
```java
@Test
public void namify() {
    String address = "Place Name, Street Name, More Address Info, More Text";
    String name;

    Namify namify = new Namify();
    name = namify.namify(address);

    assertEquals( expected: "Place Name", name);
}
```

1. This is the namify() test. When we were retrieving data from OpenStreetMaps not all places retrieved had a name tag. We wrote a function which uses the latitude and longitude of each place to retrieve the address as a string. Fortunately the name was included in this string so using Regular Expressions we were able to extract the name for each place using this. That was the purpose of the namify() function. The above test validates that given an address the function correctly returns the name of the place.



```java
@Test
public void getDistance() {
    double lat1 = 53.3466;
    double lat2 = 53.3650;
    double lon1 = -6.2682;
    double lon2 = -6.3037;
    double delta = 0.1;
    double expected = 3.12;
    double output;

    GetDistance getDistance = new GetDistance();
    output = getDistance.getDistance(lat1, lon1, lat2, lon2);

    assertEquals(expected, output, delta);

}

private static double deg2rad(double deg) { return (deg * Math.PI / 180.0); }

private static double rad2deg(double rad) { return (rad * 180.0 / Math.PI); }
```

2. This is the getDistance() test. The purpose of get distance was to return the distance in kilometres from the user's current location to individual locations of places within the app. This function was very important as it allowed us to filter out places within a certain radius of the user. The function took 2 latitudes and 2 longitudes and compared them to find the distance using some functions from the math library. Since it was dealing with decimal values we included a delta parameter in the test which allows for 0.1 of an error either side of the distance.



```java
@Test
public void getReviews() throws Exception {
    FirebaseFirestore mockDb = Mockito.mock(FirebaseFirestore.class);

    String currentCollection = "testing";
    final ArrayList<String> expectedReviews = new ArrayList<>();
    expectedReviews.add("test good");
    expectedReviews.add("test great");
    expectedReviews.add("test amazing");

    mockDb.collection(currentCollection).document( documentPath: "getReviewsTest").get()
            .addOnCompleteListener((task) → {
                    if (task.isSuccessful()) {
                        DocumentSnapshot doc = task.getResult();
                        ArrayList<String> actualReviews = (ArrayList<String>) doc.get("reviews");

                        assertEquals(expectedReviews, actualReviews);
                    }
            });


}
```

3. This is the getReviews() integration test. The getReviews() function checked that the app successfully connected to FirebaseFirestore where data such as reviews of each place was stored. In order to do this we had to create a Mock of the FirebaseFirestore class using the Mockito Framework. This allowed us to then connect and make calls to Firestore. From this we validated the ArrayList which was returned as equal to the one which we provided as expected.

2. User Testing

We performed some user tests in order to better gauge how well users could interact with the app in order to understand where it could be improved. We wrote out a number of tasks and timed some users time to completion of each task. We designed the questions so they weren't a walkthrough of the app but rather a test of how intuitive the design and layout of the app was to an average user. These were our results:

Task	User 1 Time (Seconds)	User 2 Time (Seconds)	User 3 Time (Seconds)
1	50	31	27
2	23	9	20
3	44	32	36
4	9	10	12
5	19	29	43
6	25	13	42
7	32	20	56
8	50	23	31
9	3	8	5

Task 1 - Create an account.

Task 2 - Select 3 main interests from the interests screen.

Task 3 - Select 5 interests which fall into these categories.

Task 4 - Look through you recommended events for the day.

Task 5 - Filter your recommended events to only include those within 25km of you.

Task 6 - Filter only cinemas within 15km of you.

Task 7 - Use categories to get places that are only competitive activities.

Task 8 - Change your interests to be 3 different ones from what you picked when signing up.

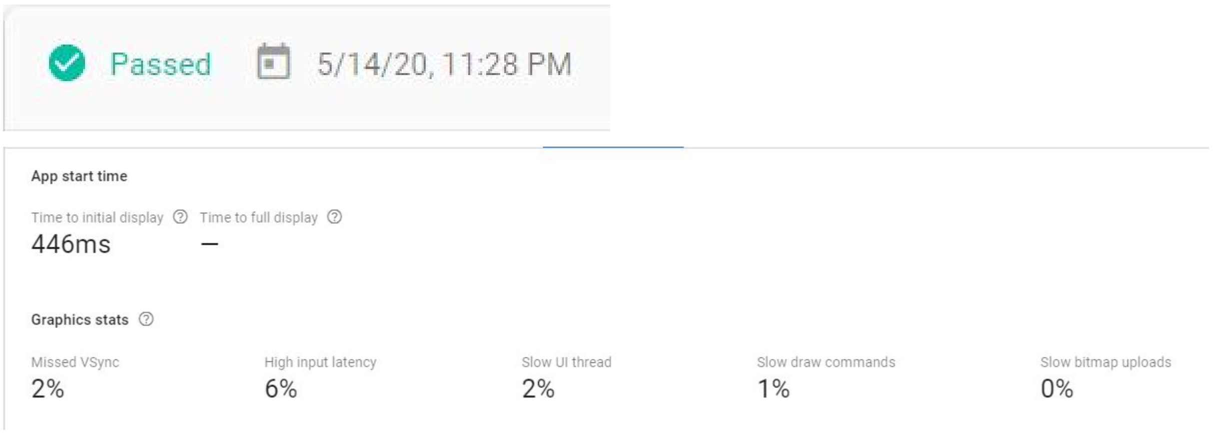
Task 9 - Log Out.

3. Code Reviews

When we were in college programming was usually done near each other in the labs. While we were working on something we would share our thought process and ideas how we were going to solve something. This lets us maximise our learning as not only one of us worked on something and we both learned about it. If we didn't do pair programming we did code reviews after to try stop errors being added. This let us use a constant coding standard though the project that we both could follow. When we started working at home due to covid-19 it made us have to change our approach to peer programming and code reviews became more important. We used Zoom as well as we could to keep in contact and work together to improve our knowledge in each other's work. Code reviews were always carried out before any branch was merged to master.

4. Firebase

Firebase provides a testing framework to make it easy to perform system tests on Android apps. This is called TestLab. It allows you to run up to 5 tests per day on a real Google Pixel device located in America and provides you with a lot of feedback on how it performed. It provides a crawl graph showing how it went from page to page and what actions were carried out. It also provides a video of how it went through the app for viewing. It also provides metrics such as network performance, latency, memory used and CPU used. The only problem we had was that since the device used was located in america it couldn't pick up any events as they were only in Ireland but it did prove to us that there were no faults in using the system and that each user input field such as text fields and buttons worked as expected.



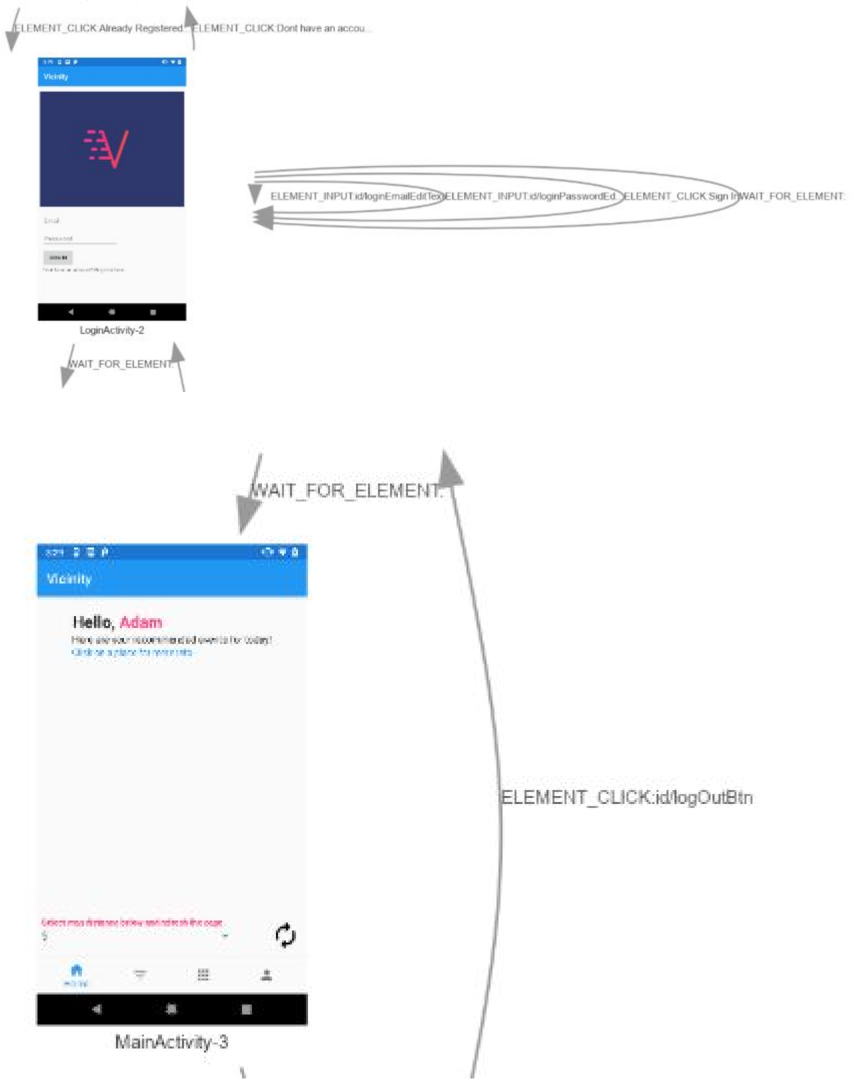
Passed 5/14/20, 11:28 PM

App start time

Time to initial display	Time to full display
446ms	—

Graphics stats

Missed VSync	High input latency	Slow UI thread	Slow draw commands	Slow bitmap uploads
2%	6%	2%	1%	0%



ELEMENT_CLICK:Already Registered...
ELEMENT_CLICK:Dont have an accou...
Vicinity
ELEMENT_INPUT:id/loginEmailEditTex
ELEMENT_INPUT:id/loginPasswordEd...
ELEMENT_CLICK:Sign In
WAIT_FOR_ELEMENT:
LoginActivity-2
WAIT_FOR_ELEMENT:
WAIT_FOR_ELEMENT:
Vicinity
Hello, Adam
ELEMENT_CLICK:id/logOutBtn
MainActivity-3

5. UI Testing

UI testing was mostly done using Ad Hoc testing as well as using Firebase TestLab, user testing also played a part in this for getting feedback on things that people didn't fully understand what it did. This let us use a more descriptive and better laid out UI as well as picking colours and buttons. To pick how the UI looked we followed the Android Material Design convention as well as picking a recommended colour palette. We also used some of Nielsen's Heuristics. The TestLab let us find errors in the code where things broke or didn't work as expected. This testing was done in the later stages of the project when we had an app that could be navigated and used data from Firestore.

6. Ad Hoc testing

This was one of our preferred types of testing when testing the UI, it let us be productive while also trying to find errors in the code. After we would write a piece of code we would try and figure out different ways to break it and how we would implement a fix. This lets us find simple errors like spamming buttons are inputting strings with numbers and symbols. The lack of documentation made this simple but also the lack of structure made it difficult to keep track of what had been tested. It also meant that after something new was implemented it would require a manual test again so the same tests might not have been implemented twice. We found Ad Hoc testing to be suitable for what we were doing and helped us find errors that we may not have found with structured testing.