

Onion Ltd. Final Report

Ross Starritt, Samuel Warfield,
Stephen New, and Dallas Andersen

Table of Contents

Introduction	2
High-level design	2
Requirements	2
High-level description/vision	2
Functional requirements	2
Non-functional requirements	3
Potential project risks:	3
Definition of Done	3
System Architecture	4
App	4
Database	4
Algorithm	5
Overall System Integration	5
Technical Design	5
Firebase	5
Pager	8
Design Decisions	10
Fragments	10
Fragment v4	10
Android app	11
Swipe to choose	11
Firebase	11
Photos	11
Login/Tokenization	11
Writing in Java	11
Default theme	11
Match presentation and storage	12
Results	12
Testing	12
Features not implemented	12
Lessons learned	12
Future work	13

Introduction

High-level design

We were tasked by Onion Ltd, just Brian Webber, to implement a pet project idea for a dating app named Kuppler. Kuppler has been designed to be a Android app inspired by other apps in the current market like Tinder or Bumble. On startup the user is prompted to login and verify their identity, this is done through Firebase cloud service. The user then can either edit their profile or match make. If the user chooses to matchmake, they are first taken to choose who they are matchmaking for. Then they are shown possible suitors for the selected profile. See Figure 1 below for full detail.

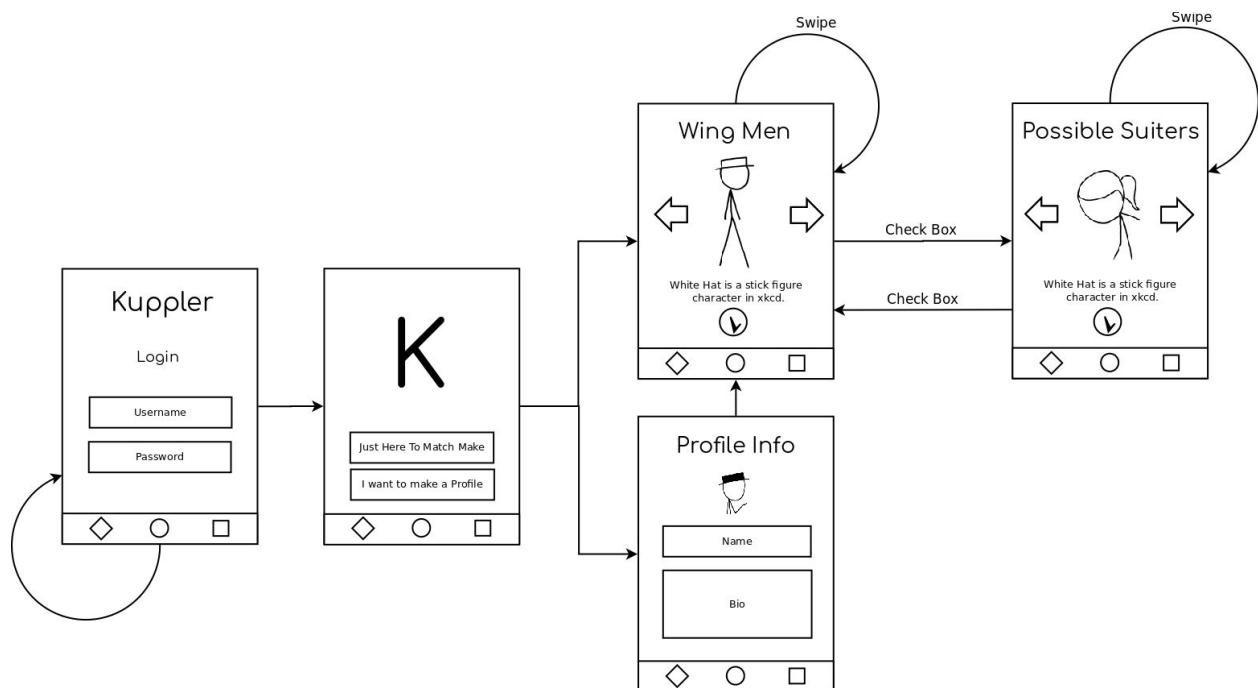


Figure 1: App Layout

Requirements

High-level description/vision

The final product is a social dating app in which users play matchmaker. The app has a simple UI in which the matchmaker is presented with a profile to choose matches for. The potential matches are then displayed to the matchmaker, and they get to vote if they would be a good match. The owner of the profile is then presented with their matches, based on aggregate votes, from an option in the overflow menu on the matching page. User data is stored in Firebase's realtime database. Computation is performed in Firebase.

Functional requirements

1. Login page with token verification
2. Create profile or just be a matchmaker
3. Pick a profile to play matchmaker for
4. Accept or reject profiles, display when match occurs

5. The program must create a selection/sort algorithm based on profile data
6. The program must have a convenient user interface that allows users to understand and use easily
7. The user is able to log in or create a profile, choose who to matchmake for, and then is presented with possible matches.

Non-functional requirements

1. Program must be an Android Mobile application
2. Database must be Firebase
3. Use of GitHub repo for version control
4. App will be written with Java
5. Program IDE will be Android Studio 3.1.2
6. Code will follow Google Java Style Guide
7. App must be engaging to end user

Potential project risks:

A. Technology Risks

1. We may encounter issues as how to connect the app and the database
2. Firebase interface will be tricky
3. Database management and population in general
4. Learning to use AWS to integrate with Google FireBase

B. Skill Risks

1. Only one member is very familiar with Android app development, others would need to self-learn
2. All members are generally unfamiliar with Firebase

Of the aforementioned requirements, all was met except for implementing AWS into our app. This is because Firebase provided all the functionality we desired and learning AWS after learning Firebase would take too much time given the six weeks.

Definition of Done

The client, Onion Ltd., is looking for the minimum viable product. In the case of the Kuppler App this means a a mobile interface with all basic functionality for the dating app. This includes profile creation and browsing, and an intuitive UI/UX design. This includes a working backend and an algorithm of our choice to best match suiters. The app should allow users to login and matchmake for another user.

For backend resources the client has made it clear that they would like to use a cloud service provider for computation and database management. To meet this requirement we are using Firebase by Google as it allows for easy database and authentication. By the end of the project the client would like to have the back and front end easily deployable by a new team if needed. Users with a completed profile in the matchmaking pool can view matches on login based on aggregate votes.

After the six week period, we have successfully met these goals in creating the minimum viable product, and have thus considered this project done by our client's perspective. Code has been delivered via GitHub.

System Architecture

App

After the user logs in, they are presented with a choice to select matchmaking or profile creation. When the profile is created, the user then match-makes. The algorithm selects profiles for the user to matchmake for. After selecting a profile, the app presents the user with profiles which are potential matches (see Algorithms section for more details). The user then decides if the two profiles match. At this point the main usage loop has been reached, and the user repeats the actions of matchmaking indefinitely. See Figure 2 below for app flow and Figure 3 for backend integration details.

Engaging the user is a big part of the app. Engagement will come across in the main idea of matching for people, playing on the sense of control. Keeping interest of the user will come specifically with new people to match for, and a chance to create someone else's happiness is a motivation for continuous use of the app.

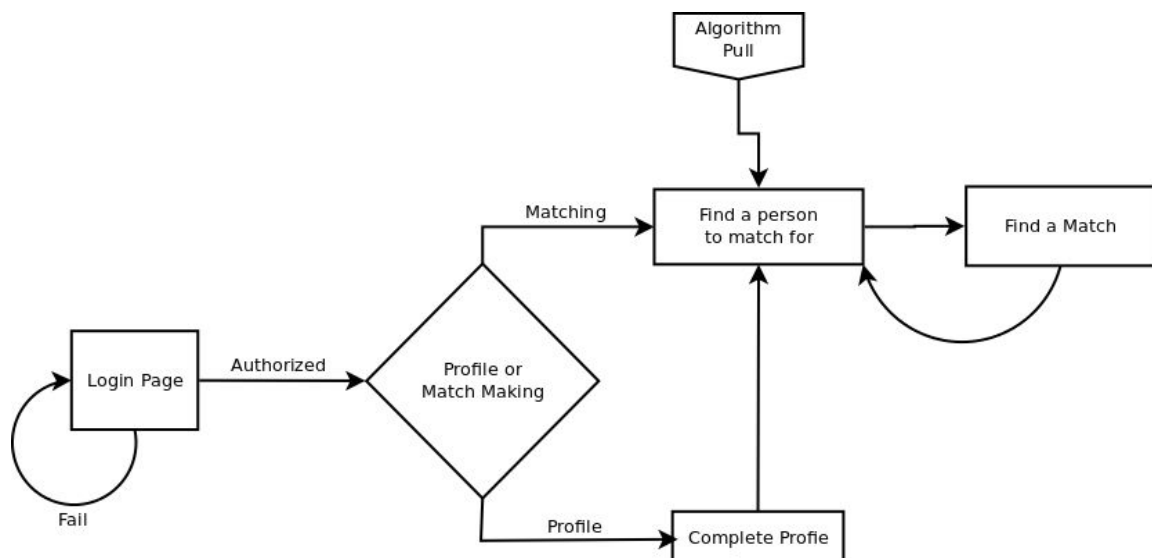


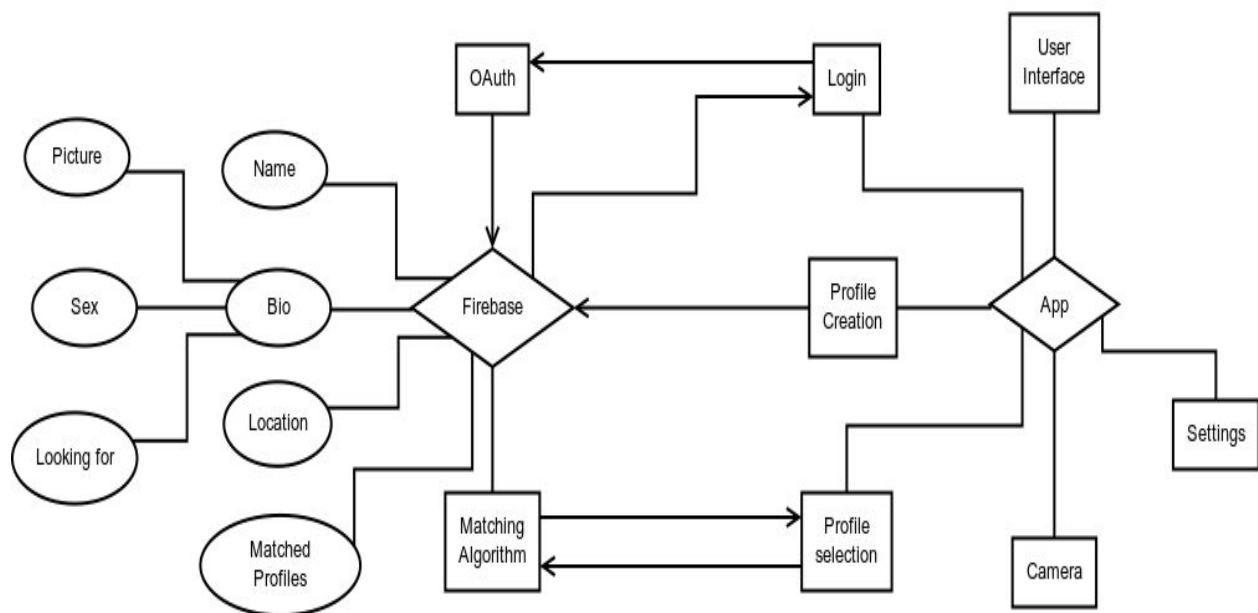
Figure 2: General Application Usage Loop

Database

Google's Firebase Realtime Database is our database software. It handles user login authentication, user and match data storage, and computations. Particularly, for each registered user in the matchmaking pool, Firebase will store name, bio, gender, seeking gender, ZIP code, matches as UUIDs, and a profile picture. Any logged in user may read any user data, but only the actual user can modify their own information and other's matches and report flag.

Firestore will also be handling the matchmaking algorithm, determining what profiles to show to users. As each user in the matchmaking pool provides a ZIP code, our initial implementation will yield all profiles with the same ZIP and desired gender as the profile they are matchmaking for. A match will be added to a user's account on match once, it is possible for one person to show up multiple times as a sign they might be compatible. Currently as flat-results are not yet dealt with, the current threshold to be displayed as a match is 1. This algorithm is extremely naive and open for extension in the future.

Overall there are three points of contact between our Firebase cloud service and the local device. They are as follows: user login services, profile creation and editing and selecting profiles to match with. This can be seen below in Figure 3, the diamonds represent the two different systems where as the squares represent functions and actions that each system can take. Lastly the circles represent information, which is completely stored off device until it is requested by user devices.



Technical Design

Firestore is a comprehensive mobile development platform provided by Google. For the scope of the project, the usage remained in the free tier. Thus scalability for

Firebase only requires a credit card. Integrating Firebase adds functionality like user authorization and management, real-time databases, and cloud storage for photos.

Rather than dealing with user tokenization ourselves, Firebase handles login verification and possible email verification (not implemented). Currently our application supports generic email/password and Google sign-in, see Figure 4. Upon account creation, Firebase automatically assigns the user a unique user ID (UUID). Firebase's authorization functionality ensures that the only logged in users can access the application. User management is also included: in the Firebase console admins may reset passwords, delete accounts and create accounts.

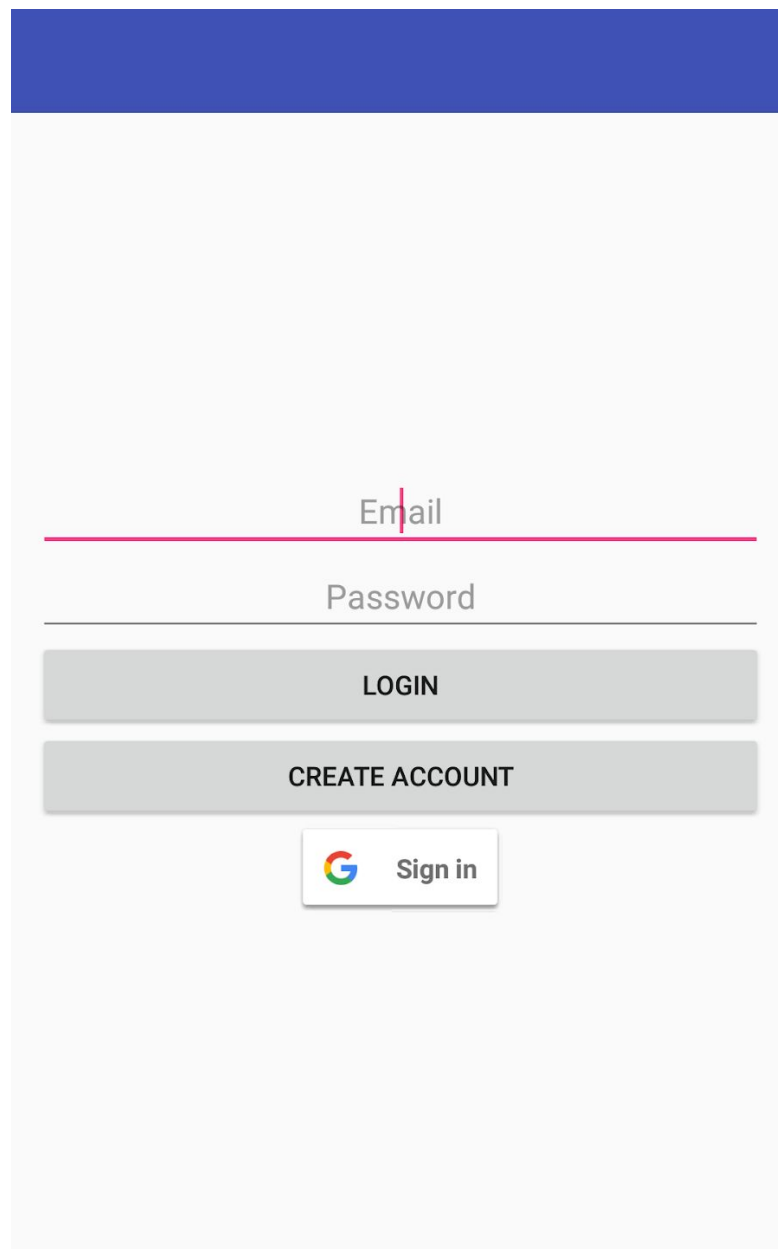
The image shows a login page with a blue header bar at the top. Below the header, there are two input fields: the first is labeled 'Email' and has a pink underline, and the second is labeled 'Password' and has a grey underline. Below these fields are two large grey buttons: 'LOGIN' and 'CREATE ACCOUNT'. At the bottom, there is a 'Sign in' button featuring the Google 'G' logo.

Figure 4: login page

Once a user is logged in and creates a profile, the user's information is stored in Firebase's realtime database. Only the logged in user may write to their information stored in the database, but by nature of a dating app, any logged in user may read any profile in the database. The structure of the database is one JSON file (extra databases may be obtained upon purchasing). The current structure can be seen below in Figure 5. Note that the first key under users is the UUID of the account and below the UUID is the profile information. In addition to storing user specific information, the profile's matches are also stored under mUnseenId.



Figure 5: Firebase Database structure

To display the profiles to matchmake for and the potential profile-specific matches, a query to Firebase is required. Logged in users may read all information in the database. As Firebase is a real-time database, accessing information requires `onDataChange` Listeners. Additionally, such listeners run as async tasks, meaning that load time must be baked into the app to have profile accessible when prompted by the user. There are two places where queries are called preemptively. First, after a user logs in, the app queries the database in the background for profiles to match for. Second, once the profile to matchmake for has been selected, before launching the page showing more information about the selected profile, the database is queried for profiles that match the desired gender and ZIP location.

As each profile mandates a profile picture, images are stored with Firebase's Cloud Store: a way to store images in a Google Cloud Bucket. Once again, retrieving the image as a byte array is also an async task. The two aforementioned preemptive Firebase queries also allows enough time to retrieve the respective profile pictures. By matter of profile query, any logged in user can retrieve any profile picture.

Pager

The pager is how the application displays profiles to the user. Each profile in the pager is its own fragment: a piece of an activity that contains some user interface and functionality of a particular screen in an app. After profile information has been retrieved from Firebase, a fragment is created for each profile for displaying in the pager. Paging between fragments of a profile required many moving parts, see Figure 6 for an example profile in the pager.

Swiping is handled innately by the pager activity, which calls the next appropriate profile. The pager's `OnPageChangeListener` function allows the app the determine the current profile displayed for reference in both Firebase and the pull algorithm. As the pager is recycled multiple times, starting the pager activity's intent is crucial. An intent is a call to the Android OS to move to another activity while carrying extra information between the two activities. For instance, once a profile has been chosen to match for, the intent launching the pager carries along the profile's UUID for ease of recording matches in the database.

Boolean information is also bundled with the intent to determine which stage of matchmaking has been called and what function the pager will take on. Similar handling of intents logic was applied to view the user's matches. Particularly, the profile does not have a button at the bottom of the screen, see Figure 7 for a matched profile.

The overflow menu, the three dots in the upper right-hand corner, is also included in the pager. On click, the overflow menu provides the following options: Report Account, Edit Profile, Sign Out, and View Matches. Adding intents to the menu lends its functionality. To manage memory, once an intent is called, the activity stack, a record of previously visited activities, is cleared.

Kuppler

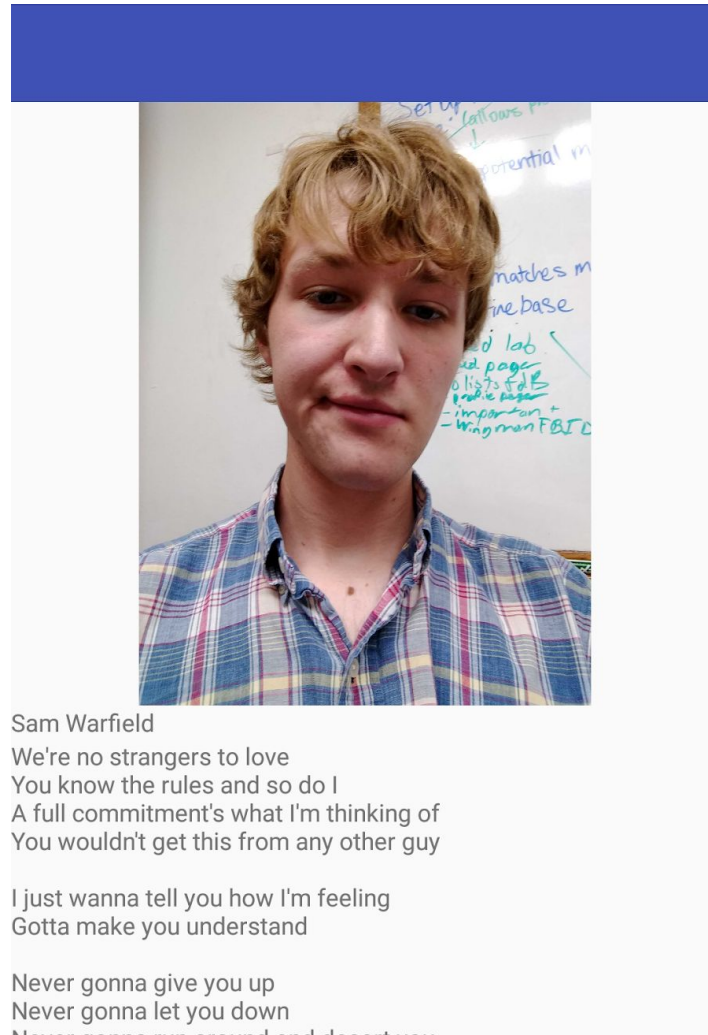


Captain Falcon

I just want someone to show them my moves. My favorite animal is a falcon. Golden eagles scare me though. I am a master of the punch attack and the Gentlemen.

MATCH MAKE FOR THIS PERSON

Figure 6: Pager Activity Matchmaking Choice



Sam Warfield

We're no strangers to love
You know the rules and so do I
A full commitment's what I'm thinking of
You wouldn't get this from any other guy

I just wanna tell you how I'm feeling
Gotta make you understand

Never gonna give you up
Never gonna let you down
Never gonna run around and desert you

Figure 7: Matched Pager

Design Decisions

Fragments

The choice to use fragments instead of just activities is a choice made to conform to the industry standard. Using fragments also allows for more degrees of flexibility and design in how the app will function and even look. Using just activities to house functionality is unnecessarily restrictive and can be too large to effectively design and retain readability.

Fragment v4

Importing and using fragments v4 is the preferred method to capture users below the APK level of 19 which we are designing for. This does limit cutting edge use of new technologies and prevent features that could make the app more flashy and new without thinking too much towards market and compatibility.

Android app

As a team we decided that the Android platform was the best option for the Kuppler project. We reached this decision after taking into account several factors such as alternative platforms and past experience. Ultimately after researching the market we found that of the mobile market over 85% of global users use Android devices. In addition a team member, Dallas Andersen, had past experience in Android development. We decided to avoid an Electron app due to no one in the group having web development experience.

Swipe to choose

We chose to use a pager for displaying profiles because we feel it is more important to show more profile information to the users rather than show several profiles at once. The purpose of the app is matchmaking, so personal information like the bio is very important in choosing good matches.

Firebase

For the velocity required to achieve MVP in these six weeks, mobile development platforms were highly desired. Of all the alternative, such as meteor, Firebase was the clear choice for its extensive functionality in providing user authorization and management, real-time database, and cloud storage. All of these features were ultimately vital in creating the MVP.

Photos

Photo management proved quite difficult. Rather than uploading images directly, the application accesses the camera and collects the taken picture. Before uploading to the Google Cloud Bucket via Firebase's Cloud Storage, the image is compressed to about half a megabyte and rescaled to less than 4096 by 4096 pixels.

Login/Tokenization

For all users of the application, they must log in even if they only wish to matchmake. The decision to force all users to create an account stems from incentivising user activity, and recognition for any good matchings they have.

Writing in Java

We decided to develop with Java in Android Studio 3.1.2. The alternative was Kotlin. All members of our team are familiar with Java, so choosing Kotlin would have slowed down the development pace. Additionally most of the Firebase documentation for Android is written in Java. Thus a proper understanding of Kotlin is needed to translate and understand the documentation. The overall base experience and documentation informed our language choice.

Default theme

Using standard themes and overall layouts are the easiest solution to meet the goal of a "minimum viable product" detailed in the design document. It's not outstanding but usable and serving the purpose of housing the app. Using another theme requires

some refactoring at this point in time and using other themes is not necessary, making our own is just too much to focus on while functionality is still on the line.

Match presentation and storage

We are storing matches as a list of UUID's as strings of whomever has been matched, the simplest identifying information in the database. The list is stored under the user as an attribute in Firebase. Users viewing matches will be presented with a pager of the profiles they have been matched with. With list's natively supported in Firebase it made parsing it in the app easier than other supported data types, like sets.

Results

Testing

The app has been tested and works on the following phones: a Google Nexus 5, a Google Nexus 6, a Google Pixel 2, and a Motorola Moto Z Play. These ran Nugget, Oreo, Kitkat, Lollipop and had different resolutions to view different layouts and their feelings on flow. All Android version KitKat and forward are supported, the largest share of the fragmented Android market. There is a hardware issue exemplified with the Pixel 2 where all of the pictures are forced into a landscape orientation: there is not much from a software perspective to fix it without much more time and resources. Despite efforts of image compression and some memory management, usage on devices with minimal RAM leads to an out of memory error due to image loading, asking for more memory from the OS is a possibility but limiting the extent is out of scope for the project and timescale.

User authentication was verified through manual testing and configuration, though not with Firebase's confirmation system. We opted for the standard and familiar Android layouts. As such, our UI reacts just as one would expect; our client Brian Weber only wanted to make changes in the order that our app presents the user with information. Mr. Weber approved each individual page layout, remarking on the intuitive button layout.

Features not implemented

Flat data was not explicitly dealt with in the pull algorithm. Distance from location was also not considered in the algorithm, and use of a zip code api will make this process work well. No other contact information besides email was received profiles.

Lessons learned

- Making a unified front end that is user friendly and looks nice for an app is a near impossible task, even if relative units of DPs are used and look fine on some devices other issues always arise on different sized devices.
- Firebase is a wonderful tool full of features devs don't want to implement like verified login and non-local storage and pulling in JSON format. It also is easy to use from both front end and back end development is also a giant boon to a short timescale of the project.

- However, as Firebase is a real-time database, accessing data requires listeners: an async task. Therefore pulling information from Firebase requires some load time.
- Memory management is as important a design philosophy cornerstone as others such as power conservation and greedy processing avoidance
- Reusability is an important design feature to use for large projects, like reusing our pager xml and some functionality is able to be reused and cut down on repeat code

Future work

For the future, the profile pull algorithm could adopt a rating system based on the number of user matches to better match profiles of similar ranking. Additionally, the algorithm will more precisely consider variables such as age and location. In largely populated areas, flat results are possible; to avoid this, the algorithm must adapt and prioritize matches with a small amount of matches to increase the signal. To combat the difficulty of image memory management, libraries such as Facebook's fresco would help better manage images and the memory. For scalability, a restructuring of the database is required, particularly storing users based on geographical location will allow for more efficient processing and accessing.