BandPal

An Android Social Media App

By Samuel Commander

Student No: **1328472**
Supervisor: **Rami Bahsoon**
MSc Computer Science

UNIVERSITY OF
BIRMINGHAM

Abstract

Post-pandemic there is a renewed hunger within many music communities for in-person connection and collaboration. However, this can be difficult for those with less time for networking or those who live in rural and sparse areas. BandPal seeks to help facilitate these connections within an Android mobile social media environment. By allowing musicians to fill their profile with details of their passions, the application then matches likeminded users on these passions they share with others, all within a specific locale. Drawing on several APIs for artist and instrument data, allowing users to connect and message, and utilising a gamification system for aiding user-retention, the application seeks to build local music communities across the globe. This paper documents the process and decisions made in order to realise this application.

Acknowledgements

I would like to give my heartfelt thanks to my supervisor, Rami Bahsoon for overseeing and guiding this project, and Martín Escardó, for inspecting it along the way. My partner Anna for her undying patience, support and advice and finally my study group of Matt, Jups, Safwa, Will, Ruby, Asha, Kris, David, Aimee and Dom, who without I'm not sure I would've gotten through even the first semester.

Table of contents

Introduction

The aim of the work described in this report was to provide a software tool with which like-minded local musicians could meet and form communities. As a lifelong musician, member of several bands and gig promoter since my teens, I've always sought out like-minded people who share these passions. One of the greatest elements of music is that regardless of genre, technical ability or instrumental palette music can always be made with any number of people. In fact, a lot of music is often improved by direct collaboration and created within genre-confines that necessitate multi-person performance.

Regardless of whether you're into Black Metal, Baroque or Balearic House, sharing your tastes with others can lead to improved music, friendship and fun, especially when your bandmates are in the same room, barriers broken down and not at the whim of an internet connection. Collaborating over the internet can be invigorating, especially given that over 60% of the planet are online[1], but in the post-pandemic landscape there's a hunger for IRL connections and collaboration.

However, meeting people and developing a local music community can be difficult when not everyone has the time to organically network through gigs and festivals, or lives in rural or poorly funded areas where the arts are less accessible. In fact the impetus for this project was a conversation I had with a childhood friend, bandmate in several projects over the years and recording artist. We both grew up in the same small rural town being into the weirder genres and instruments the world offers. But despite living in such a lowly populated area (and having such specific and aligned interests) we didn't realise each other existed for years.

This got me to thinking of a program that could facilitate the same collaborations websites like Reddit and Bandcamp do, but unlike those services, focus on creating IRL connections within a user's local area rather than strung out over many miles of fiberoptic cable.

The goal therefore was simple, create a program that had my friend and I had access to at the time, would've facilitated our years of musical collaborations much sooner. Naturally this would be classed as a social media program[2], as its principal aim would be to connect users and allow them to share personal information about themselves.

With no coding experience prior to the course, an academic background in the humanities rather than STEM and four years working in the music industry this felt like an appropriate task for the time and scope of the project. The first questions to arise would be of what format the program should take, which would then indicate the tools used, its architecture and design, the GUI layout, how to connect database, server and user and finally decisions made to retain user attention. This report documents my decision making processes in creating this program.

The report begins with **Project Specification**, analysing the problem and specifying requirements, then in **Solution Design** learning achievements and the high-level architecture of the application are handled. **Implementation** covers class descriptions, activity analysis and database design. Then **Testing** shows the debugging and analysis techniques used to make the software functional. **Evaluation** outlines the achievements and shortcomings of the project, supplying fixes to some of the weaker aspects, and **Conclusion** draws all previous analysis to a close. Finally there is the **Bibliography** and then the **Appendix** section, which provides additional insight into the project journey not present in the main text.

---

[1] https://datareportal.com/reports/6-in-10-people-around-the-world-now-use-the-internet#:~:text=As%20we%20revealed%20in%20our,the%20start%20of%20April%202021.

[2] https://marketbusinessnews.com/financial-glossary/social-media-definition-meaning/

Project Specification

Problem Analysis - Choosing a Format: Web, Desktop or Mobile?

Once the project's topic and purpose had been established, the immediate question was what form should the program take? Here I had to be pragmatic yet ambitious. I wanted to utilise the software engineering and programming skills I had developed on the course, but adapt them to a new challenge. The task needed to be far enough away from what we had done in class that I could assert myself, learn new skills and progress as a programmer, yet not be so far away that my relative youth in coding and engineering prevented me from creating a cohesive piece of software.

On the course we have principally coded in Java, and focused first on simple command line processes, before progressing to desktop applications using JavaFX for the GUI and SQL for the database. This lead me to set four stipulations:

1) Challenging myself to handle a new environment as opposed to a desktop GUI.

2) Making sure that the format of this application would allow for widespread usage.

3) If the principal language of the project was to be Java, then it must go past the uses of the language we covered in class. Otherwise I should challenge myself with a new language, and similarly advanced techniques.

4) I didn't get on so well with PostgreSQL, so finding a different way to handle the database would be optimal.

With desktop applications declining in popularity[3] and social media applications in particular having been aimed at web and mobile for years now[4] it made sense for me to look at these two platforms to pass stipulations one and two.

I first looked at web-based apps. I liked that there was so much learning material out there for JavaScript, HTML and CSS, and plenty of room for personal ingenuity, with few framework limitations. However a reservation held me back.

I wanted my prospective program to be able to reach as many people as possible, and a quick look at recent figures for website requests showed that *"68.1% of all visits came from mobile devices"*[5] compared to just 28.9% from desktops. This would mean if I wanted to maximise scope I would need to make my site accessible via both hardwares. Although suitably challenging, this would mean dividing my time and resources between two formats, and I thought this split-focus may come at the cost of features. This lead me to my final destination: mobile applications.

Having used Apple products as my go-to for both desktop and mobile for years, I first thought of making an iOS application using Apple's relatively new Swift language and Xcode IDE. All the fundamentals were there (challenging, ambitious, yet supported by adequate learning materials), but on playing around with the language I found it sluggish and frustrating. This led me to an Android Application.

---

[3] https://productcoalition.com/is-developing-for-desktop-dead-f8d7ba9fd180

[4] https://www.smartinsights.com/mobile-marketing/mobile-marketing-strategy/from-desktop-mobile-why-matters/

[5] https://www.perficient.com/insights/research-hub/mobile-vs-desktop-usage#:~:text=Globally%2C%2068.1%25%20of%20all%20website,total%20time%20on%20site%20globally.

The first stipulation of a new environment was fulfilled by virtue of an Android mobile application being entirely different to formats we had covered on the course. With the global market share split between iOS at 26.99%, Android at 72.2% and less than 1% of other OSs[6] (at time of writing) the popularity of Android fulfilled my second stipulation. The third and fourth stipulations of programming language and database posed more questions.

I first thought to adopt Java as my principle language for the application, as traditionally it's what has been used[7]. However, with Google making Kotlin their officially preferred language for Android development back in 2019[8], I was intrigued. With less boiler-plate code, easier debugging due to shorter code lines and crucially, simplified database updates via streamlined asynchronous programming, I thought it would make enough of a challenge for the scope of the project, and fulfil stipulation three.

Lastly there was the question of database and server. I settled on Google's Firebase platform, as this could act as both a database and home for server code, which allowed for authentication, realtime updates and storage for photos. It was also designed to sync up cohesively with the Android Studio IDE I chose to develop the application in.

Problem Analysis - User Interviews

Having settled on the format, programming language and database, I needed next to cement the features necessary for a compelling application. The main aim of the program was clear: to connect people within a specific local area to people who share their taste in music and want to play together. But how exactly to do this wasn't clear.

Having been active in the UK music scene both in an amateur and professional capacity since my teens, I know a lot of musicians. So, as potential users / stakeholders in my application, I wanted to get their first-hand opinions on what features they'd like to see, so that I could streamline the system's design requirements.

I selected five of my network for this, all chosen for their unique perspectives on the UK music scene, with various genres, instruments and professions represented. I gave them a simple brief, just saying that the program would be a mobile social media application that linked users' profiles to each other based on shared interests and location. Their comments were then summarised down to a paragraph.

---

[6] https://gs.statcounter.com/os-market-share/mobile/worldwide

[7] https://www.netsolutions.com/insights/best-programming-languages-to-write-develop-android-apps/#:~:text=Frequently%20Asked%20Questions-,1.,and%20Basic%20are%20also%20used.

[8] https://tcrn.ch/2vJMMO2

Information	User 1	User 2	User 3	User 4	User 5
Name	J.Cunningham	R.Commander	S.Chahad	E.Martin	A.Rocchi
Age	29	67	29	27	33
Profession	Music Therapist	Luthier	Teacher	Folk musician	University Administrator
Instruments	Drum kit, Piano, Cello	Viola da gamba, Lira da braccio, Violin	Vocals, Guitar, Production	Vocals, Guitar	Drum kit
Genres	Jazz, Metal	Early Music, Classical	Electronic, Techno, Grime	Folk, Psychedelic	Dub, Reggae, Indie Rock
Location	Bristol	Bridgnorth, Shropshire	Sheffield	London	Birmingham
Comments	I'm a bit old school when it comes to technology, an accessible application that didn't overload me with information and features would be my preference. Although, I wouldn't want it to compromise on accuracy because of this.	I've used apps like this in the past, but they've always failed me due to my slightly obscure instrument choices and influences. These other apps often wouldn't have the instruments I play listed and instead would generalise them too much - I don't play a Cello, I play a Viol. This led to me being encouraged to connect with musicians in completely different styles to the ones I'm interested in. They also often wouldn't match me on influences, only genres, so this would be a nice touch.	I hate it when apps lock off their best content behind paywalls or subscription tiers, I'd want the app to be free and not spam my phone with notifications.	Playing live is a big thing for me - the feeling of interplay with others when I'm up on stage. If the app could include videos of user's live performances that would be great. I wouldn't want to connect with anyone who wasn't at a similar skill on their instrument to me.	I'm a very private person, so no application gets on my phone unless it takes my privacy seriously. I wouldn't want the application constantly tracking my location and I wouldn't want lots of connections from people outside my own age range.

User Requirements

See **Appendix A - Use Case Diagram**.

System Requirements

Fifty system requirements were specified for this application. They have been omitted from the main text, but to view them please see **Appendix B - System Requirements**.

Solution Design

Learning

Significant practices and skills were needed to complete this project that went past what I had learnt on the MSc. The most notable of these are listed here:

- **Kotlin**

  As the central programming language of the project, I had to get to grips with Kotlin as quickly as possible. As it is an OOP language, this made learning it seem more natural than if I had chosen to code in something divorced from it. With a strict time limit for the project, I was therefore learning the language as I was building the program. Though not ideal, this had its benefits.

  I've found that when learning a new language it's common to do so via logic tests and coding exercises that sometimes have less in common with real-world applications than would be desirable. As Kotlin is syntactically similar to Java and I was on a strict schedule, I skipped these exercises and instead learnt on a real project, which I believe got me faster acquainted with the contents of the language. As a social media application, BandPal by nature deals with a lot of user data, so I needed clear data structures to collate this. Kotlin being an OOP language makes it perfect for this.

- **Android Application Development and Android Studio IDE**

  On the course we have developed a range of programs, from command line text games to client-server desktop applications that used JavaFX for their GUI and PostgreSQL for their database. However, we have not once touched on mobile development.

  Of course, I was able to adapt the OOP principles I had learnt on the MSc, but I quickly found myself picking up new ones due to programming for completely different hardware and OS. For example there was the concept of activities, where classes corresponded to a single screen of the application and connect to a myriad of different types of views. A form of this was present in the JavaFX desktop apps I'd previously developed, but significantly more complicated views were needed for BandPal, from ScrollViews to TableLayouts, going past what we'd covered in class. There was also .gradle files and the AndroidManifest to handle implementation, which involved a fair bit of trial and error to set up the necessary dependencies.

  The difference in interface between desktop and mobile is considerable, so I learnt to handle everything from thumb / slide based mechanics (as opposed to mouse and keyboard) to the built in Android back button (which I had to override at times). Then there was also GUI design differences, not just the smaller space to work with, but accounting for mobile's pop-up keyboard, the possibility of using multiple fingers at once and making the application function correctly on a myriad of devices, all with subtly different dimensions.

  Instead of using JetBrain's IntelliJ IDEA, I chose to develop the application in Android Studio[9], as this is the official integrated development environment for the Android OS, and is designed especially for it. Luckily, the IDE is also developed by JetBrains, so it didn't take too long for me to pick it up. However, there were some significant new challenges, such as using a split .xml and GUI layout designer at the same time, as in class we hadn't used JavaFX's accompanying designer, Scene Builder[10].

---

[9] https://developer.android.com/studio

[10] https://gluonhq.com/products/scene-builder/

- **Google Firebase**

  This acts as not just the database for the application, but also as the server too. PostgreSQL does not have built-in realtime capabilities, instead if you require this you need to use an aid like Pusher[11]. Google Firebase on the other hand is built around realtime database updates that you can watch as they come and go. Getting to grips with the format took me considerable time, as the database is NoSQL with data structures that take a parent / child format of lists, unlike SQL's table-based setup. I expand on my handling of these challenges in the **Implementation** section of this paper, sub-titled 'Database Design'.

High-level Architecture

In semester two's Software Workshop we learnt how to implement the client-server model, with both client and server being contained within a single block of client-side code. However as I had decided to use Google Firebase as the database for the application, I was intrigued to discover that Firebase offered a client-database model that bypassed the need for (almost any) server-side code. In light of all of the new skills I was learning and challenges I was facing I sorely needed a simple architecture. This model's benefits were not just limited to simplicity however. For example, the model allowed me to meet my proposal's preliminary aim of having Facebook and Google logins as well as just email. Firebase eliminated the need to write server-side authentication code. There is however, a small amount of server-side code, which was written in JavaScript and hosted on Google's Cloud Functions platform so that the app was able to handle device-to-device notifications. This is discussed further in the **Implementation** section under the *ViewMatchProfile* class analysis. For a full overview of each activity in the program, see **Appendix C - Class Diagram**.

[11] https://pusher.com/tutorials/postgresql-realtime/

Implementation

Class Descriptions and Activity Analysis

Here I will go into detail on all implementation and design decisions made for the most relevant classes in the program.

In Android mobile development each activity of an application (i.e. each individual 'screen') corresponds to one class (in the case of BandPal, one .kt class file). This primary activity class handles the code for all elements of the GUI, makes requests to the database and manages connections to other appropriate classes, such as data and handler classes, as well as API connections.

For each activity (and therefore each primary class) I will explain my process around their structure, connections and GUI layout via their accompanying .xml file. Every activity and class is sorted into one of five packages: com.sam.bandpal, influencesAPI, instrumentsAPI, userData and usernames. This helps to distinguish what each class is doing.

The names of each package should make it clear what the classes within are concerned with, but for avoidance of doubt: com.sam.bandpal[12] contains all classes that have GUI counterparts (thus, the core classes of the application), influencesAPI handles all LastFM Artist API connections, instrumentsAPI handles all MusicBrainz Instrument API connections, userData contains mostly data classes pertaining to user information and usernames has a single 'master' class used for keeping track of all unique user UIDs.

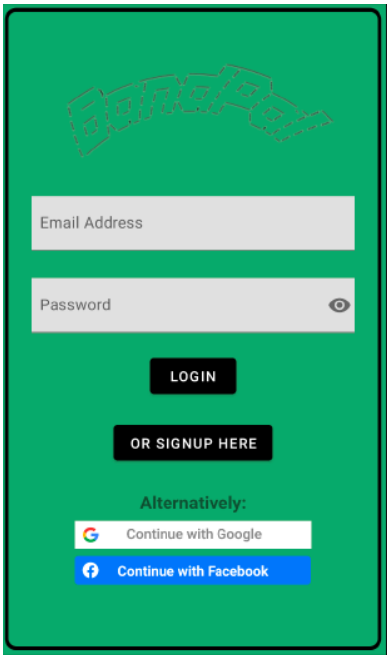
A full visual guide to BandPal's classes can be found in **Appendix C - Class Diagram**.

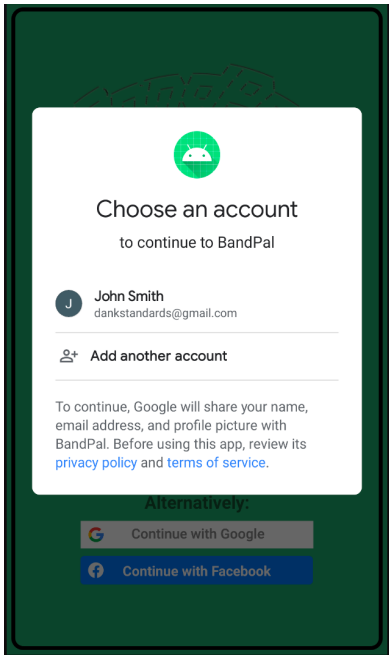
[12] Unfortunately this package is named 'com.sam.tippy' in the src files, as 'Tippy' was the old name for the application, and on refactoring it to change the name the application no longer ran. I spent considerable time trying to fix this but to no avail, so 'com.sam.tippy' remains.

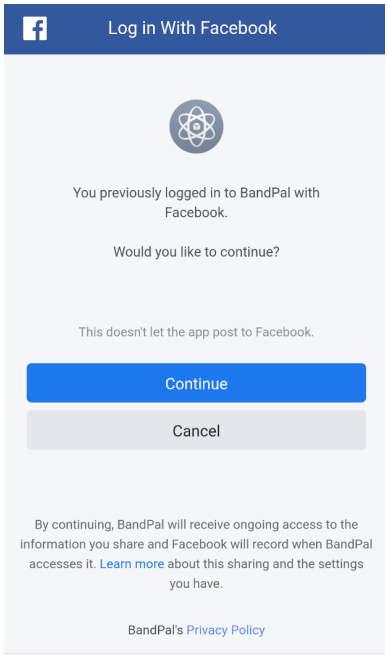
Login (com.sam.bandpal)

Functionality and Data Structures

The *Login* activity acts as the gateway to the application. It is the first screen that a new user sees when booting up the application. The main onCreate() method is overridden by onStart(), which acquires the user's UID (a standard identifier created whenever a user is authenticated via Firebase) and makes a call to the database to see if the user has data there. If they do then the user is forwarded onto the *ViewProfile* activity, if not then onCreate is begun. There is also an additional check to allow the user to stay on the *Login* activity if they have reached the activity via logging out of their account rather than booting up the application from scratch.







onCreate() is responsible for five main features, returning user login via their email and password, navigating new users to the signup activity, storing unique messaging-tokens for each user session and allowing both new and returning users to login using their pre-existing Google or Facebook accounts. The most important of these are covered below.

The decision was made early on to use view binding to save on lines of code and increase code-legibility. By employing this I didn't have to individually declare every widget used on each class' accompanying .xml layout file, a welcome change from JavaFX's long stream of declarations.

For the login screen of standard users, Google Firebase's signInWithEmailAndPassword() method was employed within simple if statements to check user details, forwarding onto either *ViewProfile* or *ChooseProfileType*, depending on whether user data was present.

It was a goal from my initial proposal to allow for users to login to the application via their pre-existing Google or Facebook accounts. To achieve this, code from Google's[13] and Facebook's[14] developer portals was adapted. Coordinating the two so that they both functioned properly and didn't interfere with each other proved quite a challenge. In particular, reconciling their onActivityResult() methods was initially confusing, until I realised that they could be combined into one method.

---

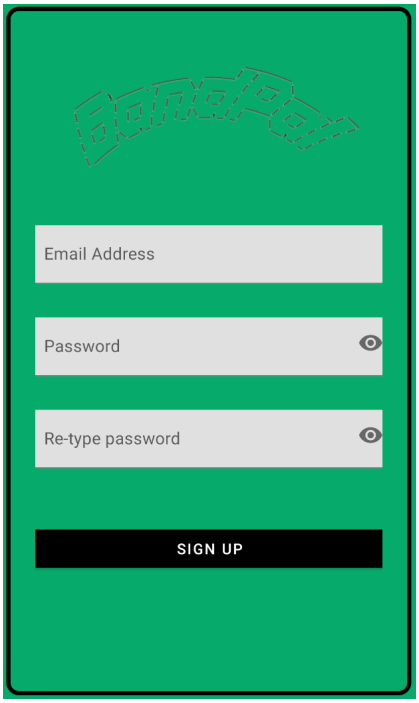
[13] https://developers.google.com/identity/sign-in/android/sign-in

[14] https://developers.facebook.com/docs/facebook-login/android/

GUI

The GUI was not a central focus of my project, as I opted to concentrate on developing features instead. However I still wanted a focused, minimal and clean GUI that didn't detract from the functionality of the application.

A striking green was chosen as the lead colour to catch the user's eye, with white and grey as secondary colours. The *Login* activity is one of three screens that are stamped with the application's logo, which is made of manipulated dashes that recall the ASCII art craze of the nineties, referencing the computer science impetus for the project.

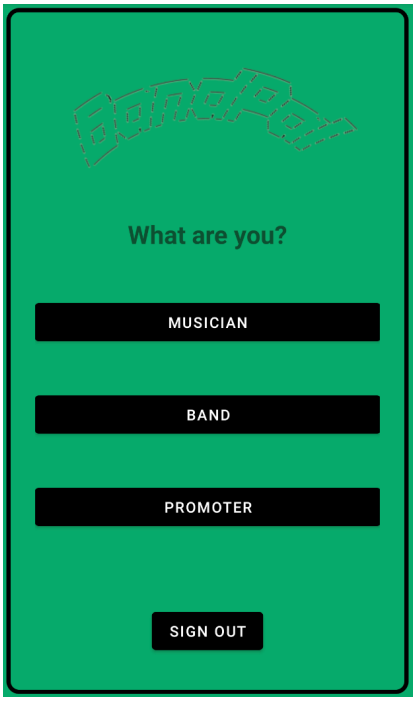


Signup (com.sam.bandpal)

Functionality and Data Structures

A simple class with a simple task. This activity allows the user to signup to the application with their pre-existing email address. As this class acts as another entrance point for the application there is a retrieveAndStoreToken() method that saves a unique String to the database for each individual session.

Despite being used to create users, there is no user object employed to save user data at this point. Instead, Google Firebase's authentication service is engaged via the createUserWithEmailAndPassword() method to give the user a UID. I found getting the authentication service setup with Android Studio and the application very difficult. There was a lot of trial and error with changes to the build.gradle files and searching through Google documentation for the correct implementations. This is difficult to qualify here, but I wanted to note it all the same.



GUI

A very similar GUI to *Login* to maintain consistency. Each password EditText has an eye-lock to aid privacy.

ChooseProfileType [YoureInActivity] (com.sam.bandpal)

Functionality and Data Structures

A very simple class where the user chooses (with finality) what form of user profile they want to have while using the application. Although this decision is final, the choice should be obvious as there is no crossover between solo artist and band. The third option of Promoter is visual only, as I didn't end up having time to implement this feature. I expected this

would be the case on my proposal. There is also a logout button that when pressed not only logs the user out and takes them back to *Login* but also deletes that session's token from the database.

GUI

I could have opted to include this choice of profile type in the *EditProfile* activity, but a design decision was made to do it as a separate activity for two reasons.

Firstly, I wanted to signal the fact that all three options change the user's experience of the application. By setting the decision as its own screen, this subconsciously indicates to the user that this is a key choice they have to make.

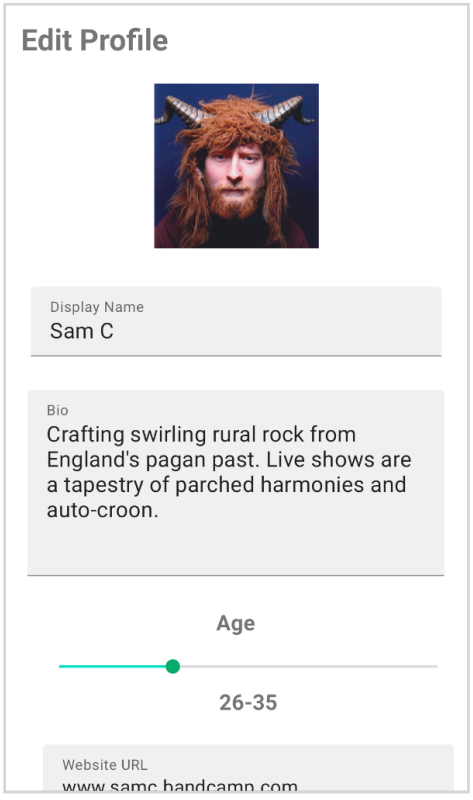
Secondly, if this choice were to be made as part of the *EditProfile* activity then it would be the only 'editable' piece of user data that was unmodifiable. This would clash with the very purpose of an edit screen, so it has its own screen to prevent user-confusion.

EditProfile (com.sam.bandpal)

Interacts with:

UserUIDs (usernames)
User (userData)
Instruments, Request (instrumentsAPI)
Artist, ArtistMatches, Request, Results (influencesAPI)

Functionality and Data Structures



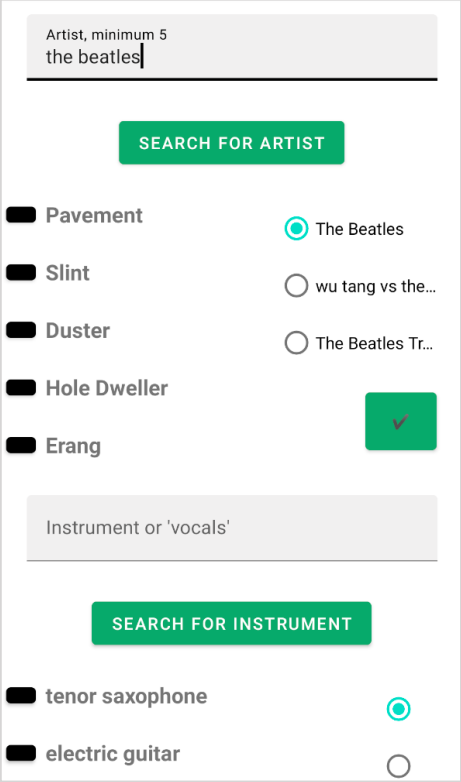
This was one of the most challenging activities to implement, with a myriad of engineering decisions to make and tasks to face as I went through. The most important questions were - What user data should be required? How should it be saved to and arranged in the database? And how should the user data be retrieved and updated?

Before any functions could be engineered for more significant information I wanted to work out what baseline user info was needed, and what data structure it should be temporarily stored in before heading on to the database. It is important to note here that regardless of whether the user selected their account to be a band or solo musician profile, *EditProfile* works nearly identically for them. The differences between band and solo feature much more prevalently in the *Search* and *SearchResults* classes, and will be covered in their accompanying sections.

Display name, a short bio, a profile picture and a website url were designated as this most basic of information. As Kotlin is an object-oriented language it made sense to create a *User* data class that would hold an individual user's data within an object. These *User* objects could then be saved to the database. All of this basic data could be saved as strings, with the image being taken from the user's camera roll, and converted to a url.toString() that lead to Firebase's image storage function.

Right from the start of the design and implementation process the most important question was: how can the application connect people of like-minded interests? I decided that a cohesive way to do this would be by matching strings of identical user data. The more matches, the better chance those two users had of wanting to connect. But this lead to another question: how would string consistency be maintained? There would be no point trying to reconcile five different spellings and capitalisations of the word 'guitar' with each other. Therefore for every piece of additional user data I needed to find a way to unify them.

Age was simple, instead of integers a SeekBar saves the bracket that the user falls into as a string that could be easily matched. This also aided privacy, as users do not have to give a specific age. I decided on location, influences (musicians, bands etc), instruments and genres as the remaining pieces of information, which would all prove tougher to implement.



With the aforementioned problem of matching opposing spelling and capitalisation, a user's influences needed to be able to be reconciled with each other, meaning that I needed to have access to a large database of artist names. For this, LastFM's Music Discovery API was employed. Each time a user searches for an artist via a text box the fetchLastFMData() method makes an API request of the user's searched string. The LastFM API then returns Json code that is filtered by the classes in the influencesAPI package, with the upUI() method displaying the top three results from this reply as intermediary radio buttons before the user confirms their entry. The user is required to save no-less than five influences via this process, to give the search algorithm the best chance of finding relevant potential matches. These influences are added to a MutableList of type string which is itself later added to the *User* object.

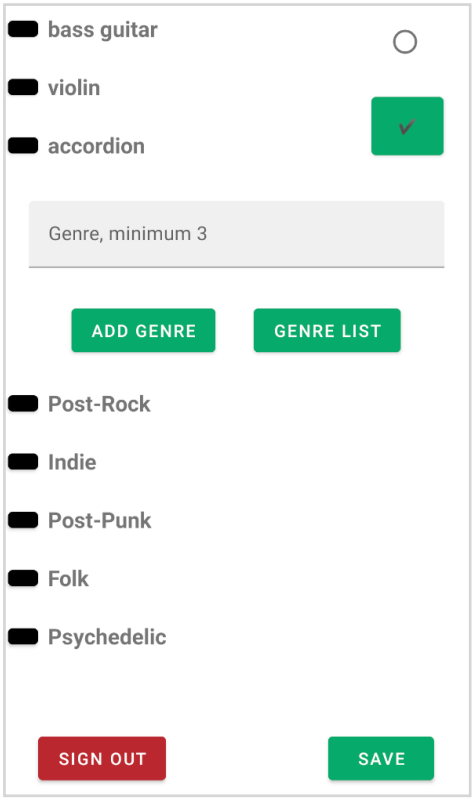
This process is similarly implemented for the user's instruments, except they're only required to put down one instrument minimum. MusicBrainz' Instrument API is used for this, following the same format as before. However the API database is missing the 'instrument' of human vocals, so it is hardcoded in as an option.

The process for music genres at first seemed like it was going to be very similar, but rather than go with an API database of genres I opted to create my own list in the database. This decision is central to the search algorithm relying on string matching. Unlike influences and instruments, genres all either sit within a dominant genre, for example Thrash being a sub-genre of Metal, or are a dominant genre themselves. With string matching it would massively decrease the likelihood of the algorithm connecting two users if one said they liked Dark Ambient and another said they liked Dungeon Synth. However these are both sub-genres of Ambient and very similar, so we can achieve far more matches by labelling them both as their parent genre. The fetchGenreData() method works much the same as its influence and instrument counterparts, but instead of calling an API it calls the database, returning the genre list as a TypedArray and looping through it to search for matches. Unlike influences and instruments, there is no intermediary step before the user confirms their choices, so a dialog box was added with the list of genres the user can choose from. The structure in the database of the genres is simple, with a heading (or reference) of 'GenreStore' and a child 'genres' containing a MutableList of string genres.

The last piece of user data to be added is their geolocation. With BandPal explicitly encouraging IRL contact above internet-relationships there needed to be a way for users to be findable within a local area. Before beginning the project I thought this might be one of the toughest areas to engineer, however it turned out to be relatively simple.

The getLocation() method is called via a button in the GUI, which checks to make sure the user has the necessary permissions allowed on their phone, then uses the fusedLocationProviderClient class to obtain the user's latitude and longitude, saved as doubles.

At the end of the activity the user can tap the save button which runs a loading wheel while the computations occur in the background. Checks are first made on whether the user is saving or updating details via a database UID lookup, then several reviews are run to make sure that the appropriate user information is present before calling the saveUserDetails() method to add each element to a *User* object, which is then itself added to the database. If successful, the user is then redirected to the *ViewProfile* activity.



I tried to keep the database structuring of user data as simple as possible. A benefit of Google Firebase as opposed to SQL-based structures is that the database can be viewed and edited without the need for queries by the database admin. Thus each user's UID is a sub-heading (or 'child' as Firebase calls it) of the Users heading, with the user's *User* object stored under their UID. For ease, every user's data is stored as strings or lists of strings, apart from their geolocation, which is two doubles, latitude and longitude.

Lastly we have the retrieval and updating of user data. The readUserDataEdit() method makes standard database requests on startup of the page. The only significant challenge here was how to turn the profile picture URL into a viewable image. To do this Picasso[15] was implemented to load images into the user ImageView.

The updating of user data surprisingly proved to be the biggest challenge of the whole activity. I spent multiple days trying to program a single method that would allow the user to either update their info but not their photo, or the other way around. This problem stemmed from the fact that if the image had been updated in that session by the user, then all other user info updates had to be included inside its listener. So if the photo had not been updated, then the other updates could not be reached. Unfortunately, the only way to fix this that I found was the duplication of variables, so I split the method into two, one for photo updates and another for without.

GUI

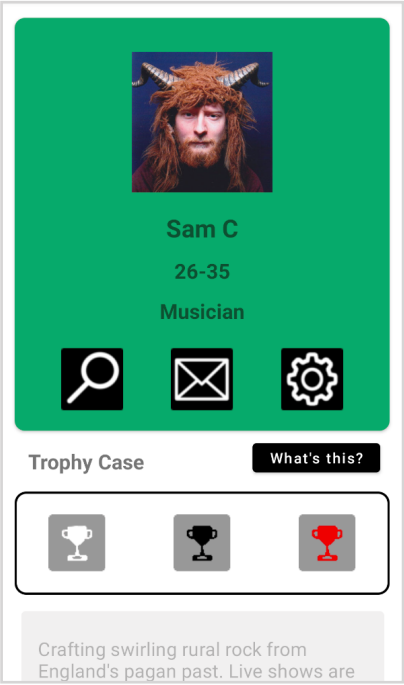
As is common in social media apps, the *EditProfile* screen is simple and clean. One design decision to make was how to fit the long and potentially interlocking strings for instrument, genre and influence names. A character limit was set, and all information was kept on the single activity, so that a user could see all of their entries and options at all times.

[15] https://square.github.io/picasso/

ViewProfile (com.sam.bandpal)

Functionality and Data Structures

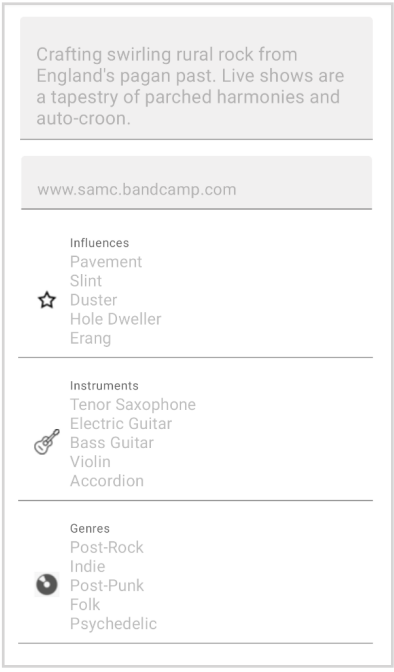
Unlike similar social media apps such as Instagram, Facebook or Reddit there is no content feed in BandPal, as its purpose is to facilitate IRL connection rather than purely online. Thus, despite *ViewProfile*'s main job being to clearly display the user's data, it also acts as a homepage for the user's experience. There they can navigate to all other key screens of the application, namely messages, search and edit profile.



Frustratingly, to display the user's data there was no way for me to receive the MutableList entries (influences, instruments, genres) in the database to the client side in list form. So these had to be downloaded as strings and then cut up into usable lists.

Despite *ViewProfile* ostensibly being an activity that displays information rather than edits or processes it, it is the first point at which the user experiences the trophy system, which plays a key role in the central algorithm of the application - the search function.

Trophies are rewards to users for using the application, the more trophies, the higher the user will appear in other user's searches, this is explained to users through a dialog box. However, this dialog box does not list the exact trophies that can be potentially attained, instead it encourages users to seek out other more seasoned users' profiles to discover how to attain more trophies. The trophies are:



1) First Match - Attained by connecting with at least one other user through the search function.
2) Maestro - Attained by adding five instruments to a user profile.
3) Genrehead - Attained by adding five genres to a user profile.

Rather than being their own elements to be saved to the database, it was decided to keep the trophies as totally dependent on their reasons for being given. This way there is no need to take up space in the database with trophy entries, as they're purely visual elements that appear or don't appear based on individual user's data.

GUI

The GUI for *ViewProfile* is arguably the most important visual element throughout the application. This is partly because it is the home screen for users, and thus seen the most, but mainly because it shows what other users of the application see when they go

to your profile. The goal of the application being to establish connections, an attractive profile will aid this. The application's main colour of green was employed to aid brand visibility, the trophy case is prominently on show, and all user data is clearly displayed.

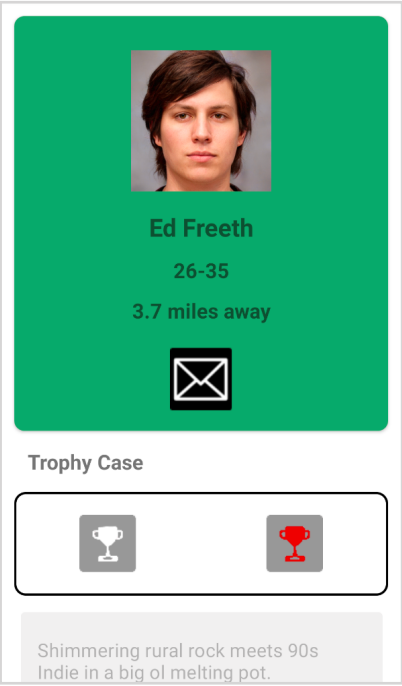
ViewMatchProfile (com.sam.bandpal)

Interacts with:

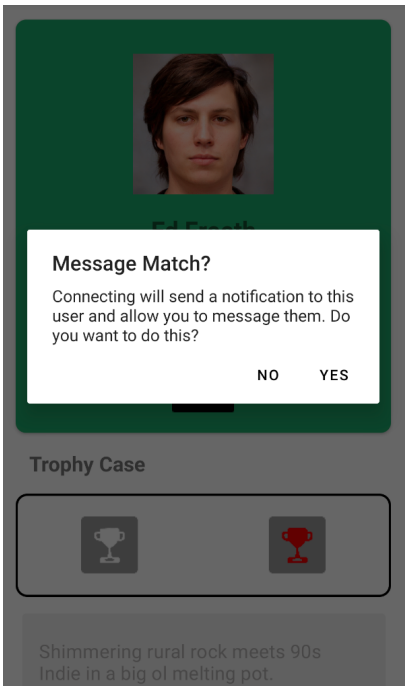
Notification (com.sam.bandpal)

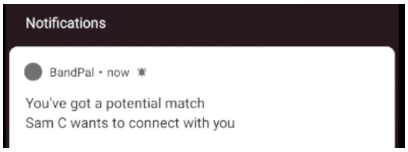
Functionality and Data Structures

Extremely similar to *ViewProfile*. *ViewMatchProfile* is a generic activity that displays a user's match's information. As the framework is so similar to *ViewProfile* all the activity needs to fill out its contents is the UID of the current user's potential match. The match's data is then requested from the database and displayed as it was for *ViewProfile* via the readMatchData() method. There are however a few significant features.



The message button opens a dialog box which (when confirmed) calls the messageMatch() method. This connects the two users by adding each to the other's connectedMatches list, which is itself connected to their UID's user data. The method then calls sendNotification(), which supplies a notification to the match, inviting them to return a message to the user. It is important to note, that due to limitations in time I was able to engineer notifications to be received by the match only when they are running their application in the background (or not at all) of their phone. If they're currently using their application when a notification request is sent then they would need to check their *MusicianMessages* to be able to see the new connection.





As covered in the High-Level Architecture section, there is very little code for BandPal that is not client-side, as Google Firebase acts as a replacement for a server, also containing the database. However it was necessary to write some JavaScript code for Google Firebase's server-less execution environment (Cloud Functions) to allow the sendNotification() method to process notifications. I used Node.js to connect my client-side code to Cloud Functions, which fires every time the method is triggered by the user pressing chatButtonNew and giving a positive response to the dialog box. This method, and system of notifications was based on code by Musab Nasr[16].

[16] https://youtu.be/O3R4jaR7QpA

GUI

Deliberately near identical to the layout of *ViewProfile* for consistency across user experience. The only significant differences are that the profile type section is replaced with the distance the current user has from the match, and the three buttons of search, user messages and edit profile are replaced with a button for messaging the profile of the match displayed, and a button for returning to the user's own profile.

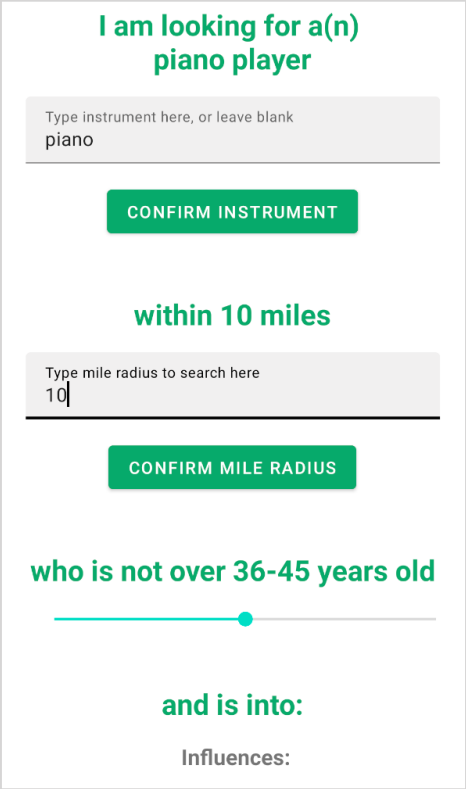
Search (com.sam.bandpal)

Interacts with:

UserSearchData (userData)
Instruments, Request (instrumentsAPI)

Functionality and Data Structures

The central algorithm of BandPal is its search and match function, whose core logic and code is found in the *SearchResults* class. Searching to match users based purely off their profile user data would be satisfactory, however I wanted to offer extra customisation.



The *Search* activity acts purely as a way of feeding extra information into the search and match algorithm. The extra information comprises of three parts: a searched instrument, a radius and an age bracket. This information is all collated into a *UserSearchData* object from the userData package.

The user can choose to search for a match who plays a particular instrument, or leave the search box blank. Once a term has been searched, the fetchInstrumentData() method calls the various instrumentsAPI classes in the same way it did it *EditProfile*, however in *Search* the decision was made to have no accompanying radio and confirm buttons, purely for ease of user experience. As 'vocals' and 'no option' weren't in the MusicBrainz instrument API, a second method, upUIBlank(), was made to handle those options. Next the user can input a mile radius, specifying how far they want to search from their current location.

Finally there is the seekBarSearchAge, which the user can slide along to the age bracket they would not like to search over. So for example an 18-25 year old may not want to match with anyone out of their own age bracket, therefore they would set the SeekBar to find a user 'who is not over 18-25'. Despite seeming quite simple to engineer, this SeekBar required considerable thought. There was the problem of security, which had been a central point of one of my prospective user interviews. I wanted to avoid a rather unsavoury use case where someone in a much older age bracket was repeatedly searching for those in the 18-25 category. Therefore, I needed to make a database request for the current user's information to prevent such a use case from being possible.

It would have saved on database space to pass the *UserSearchData* object (containing the searched instrument, radius and oldest age bracket) from the *Search* to the *SearchResults* activity, but it was decided to save it to the user's UID entry in the database instead. This was so that any analytics of highly searched terms could easily inform future improvements to the system. As opposed to that information being inaccessible by an admin.

GUI

The activity continues the general design theme of clean and minimal, but to aid usability changing visual queues are added. As each element of the search is confirmed, a sentence is formed, stating 'I am looking for a(n) ~insert instrument~ player, within ~insert number~ miles, who is not over ~insert age bracket~ and is into ~user's influences and genres~'. The non-user-changeable parts of the sentence swap from red to green as incrementally every search element is confirmed. This aids perceptibility of how far the user is through the search process and as the activity's logic relies on confirmation (rather than just stating) of user terms, the visuals make sure a user doesn't forget to put in one or more search elements.

SearchResults (com.sam.bandpal)

Interacts with:

PMatch (userData)

Functionality and Data Structures

The class of *SearchResults* holds the most important logic and functionality of the program: the search and match algorithm. First I will describe the high-level architecture of the algorithm, then its steps and my decision making process for its implementation. Some evaluation will be present here, but for the fullest analysis see the **Evaluation** section.

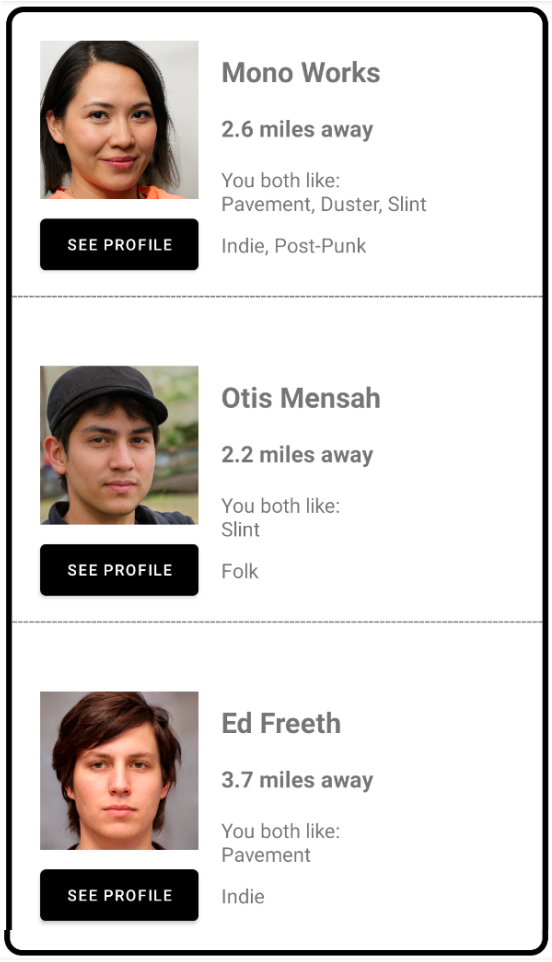
The high-level architecture of the algorithm naturally came before any attempts at implementation. Rather than try and come up with small elements of the code and then join them together, I instead laid out the entire algorithm in broad strokes, and moved through it slowly turning theory into practice.

If I were to sum up the algorithm in one term I would say that it is 'points based'. Each prominent section of the algorithm follows on from the previous via a series of methods that call each other in a chain i.e. pullUsernames() is called first, which then calls matchInstrMileAge() when it is complete and so on and so forth. This was one of the first decisions made, as it would separate the logic into manageable chunks of code and thus aid debugging. The easiest way to elucidate the algorithm is to go through it method by method. As the algorithm moves from method to method it passes an ever diminishing list of potential matches (pMatches) for the current user's search terms as it whittles away the worst matches until it is left with only the best. Each pMatch has an associated 'point-score' that increases if that pMatch's suitability is judged to be good (via string matching, geo-location and trophies) as the pMatch UID list is passed through the algorithm. At the end those pMatches with the highest score are outputted as GUI elements for the user to interact with and choose whether to connect with or not. The pMatch UID list takes the form of either a MutableList or Map depending on what data structure is required at the time. I will now go through the logic and implementation decisions made for each method of the activity.

pullUsernames()

As the algorithm is ostensibly search-based, the first method of pullUsernames() downloads all users to a MutableList so that it can be iterated through within the client-side code. It would be computationally more effective to not have to search through the entire database of user UIDs. But

I could find no way to only pull the most relevant from the database without iterating through them in some way. The only alternative way of doing this I theorised was having user UIDs pre-sorted within the database in categories by instruments they play or age groups they belong to so when it came to download and search through them, only a portion of users' UIDs needed to be requested. However I decided that this would've got messy fast, with lots of interweaving database categories, been hard to debug and required a lot of time I didn't have to implement. So in the interests of getting a solid working algorithm within the time available, I opted to download the whole user list to iterate through. As the list is passed from one method to another, it is gradually whittled down until only the most relevant potential matches are left. As the method is a simple download of an array, with no checks being performed on it, the Big O time complexity is O(n).



matchInstrMileAge()

Once pullUsernames() has completed it passes the UID list into matchInstrMileAge as a parameter. The current user's search preferences can be divided into two sets. The first of these criteria are 'hard', as in any potential match necessarily *must* meet them to appear as a search result. Whereas the second are soft, so potential matches don't certainly *have to* meet them. matchInstrMileAge() regards the hard of these sets, so it downloads the current user's search terms (previously stated by them in *Search*'s *UserSearchData* object) to match against the instruments, geolocation and age bracket of each potential match.

The method loops through the potential match (pMatch) UID list following a straightforward boolean system. Three booleans (set originally to false) correspond to each of the aforementioned method's 'hard' search terms. As the pMatch UID list is looped through, it downloads from the database that particular pMatch's instrument list, geolocation and age bracket. If one of the pMatch's instruments corresponds to the searched instrument (or there is no searched instrument) then the accompanying boolean is turned to true. Then Location.distanceBetween checks the distance between the pMatch's latitude and longitude and the user's latitude and longitude. If the distance is less than or equal to the searched radius then the accompanying boolean is turned to true. Lastly the age brackets are reconciled via an if statement, with the adjacent boolean turning to true if the searched top age bracket is higher than or equal to the pMatch's age bracket, or if the searched bracket is 60+, as there is not higher.

The end of the pMatch UID list loop is the end of the method. At the conclusion of every iteration if all three booleans are true, then the pMatch's UID is added to a list. Once every iteration is complete the matchInfluencesAndGenres() method is called, passing the new list as a parameter. With the list of pMatches being iterated through and each element checked, the Big O complexity is O(n).

matchInfluencesAndGenres()

Once the list is passed from *matchInstrMileAge()* all pMatches in it have cleared the three aforementioned 'hard' agreements of instrument, radius and age. *matchInfluencesAndGenres()* handles the 'soft' stipulations of influences and genres, but it is also where the gamification of the application interacts with the algorithm, with each pMatch's trophies playing an important role too. The method aligns most with my declaration of the algorithm as 'points-based', as for every match made between the user's genres and influences and the pMatch's, the pMatch gets a point. At the end of the class these 'point scores' are tallied and the five pMatches with the most points are available for the user to choose to interact with.

The method first downloads the current user's influences and genres to be matched against each pMatch's. It then loops through the pMatch UID list the same as done in *matchInstrMileAge()*. As each pMatch has already passed the 'hard' stipulations there is no need for booleans. Instead, a variable named 'point' is incremented by one for every match between the user and the pMatch's influences and genres. Frustratingly, I could not find a way to avoid nested for loops here, as the genres and influences are stored as arrays. However, these arrays cannot exceed a quantity of 5, which severely limits the number of iterations that would need to be made. A further analysis of this problem can be found in the **Evaluation** section of the report.

After the matching of influences and genres has been carried out the loop gets to the pMatch's trophies. As covered in the *ViewProfile* and *ViewMatchProfile* class descriptions the trophy system is a gamification element of the application whereby users are rewarded for using the application and filling out a full profile by featuring higher in other user's search results, thus making it more likely they can build connections. The trophies are Genrehead (if a user has five genres on their profile), Maestro (if a user has five instruments on their profile) and First Match (if a user has matched / connected with another user). Implementing the trophies was simple, if the pMatch's instruments and genres array sizes were five then they get a point for each, and if they have a *connectedMatches* entry in their database table then they also get a point (as it means they have a First Match trophy).

At the end of each loop the pMatch UID, display name, image URL, shared influences and genres (with the current user) and distance between that pMatch and the user are added to an object of the *PMatch* class. Here I encountered a problem. I needed to add these *PMatch* objects to a data structure that would allow them to be ordered by the amount of points each one had before passing them to the next method for displaying to the user. The obvious choice was a SortedMap, with the key being the pMatch's point score and the value being the accompanying *PMatch* object, all sequenced in descending order according to each pMatch's point score. But what if there were point ties? Maps by definition cannot have duplicate values as keys. To break these ties I decided to add a random double between 0.0 and 0.99 to each pMatch's point score, as if two matches had the same original point score, it wouldn't matter which of them came out on top.

The final logic of the method is to pass the SortedMap of pMatches as a parameter to the outputPMatches() method. However, since a SortedMap by definition cannot be iterated through, I opted to also pass a SortedSet containing the keys (themselves pMatch point scores) of the SortedMap in descending order. This was so that the SortedSet could be iterated through and the values within it (*PMatch* objects) could be accessed. The Big O complexity of this method is frustratingly higher than other parts of the algorithm, and will be covered in the **Evaluation** section.

outputPMatches()

The last method in the activity acts purely as a way of outputting the information found in the pMatch SortedMap to the GUI for user interaction. It is split into modules, which each account for a portion of the UI. The top five entries in the SortedMap are outputted.

GUI

The GUI is split into five modules, each dedicated to a pMatch of the current user's search. Only the most important information is displayed, being the matched genres and influences, along with basic pMatch information like their display name and photo. Once again Picasso is used for the profile pictures and a prominent button allows the user to navigate to that specific pMatch's profile. This is connected by intent calls to *ViewMatchProfile*.

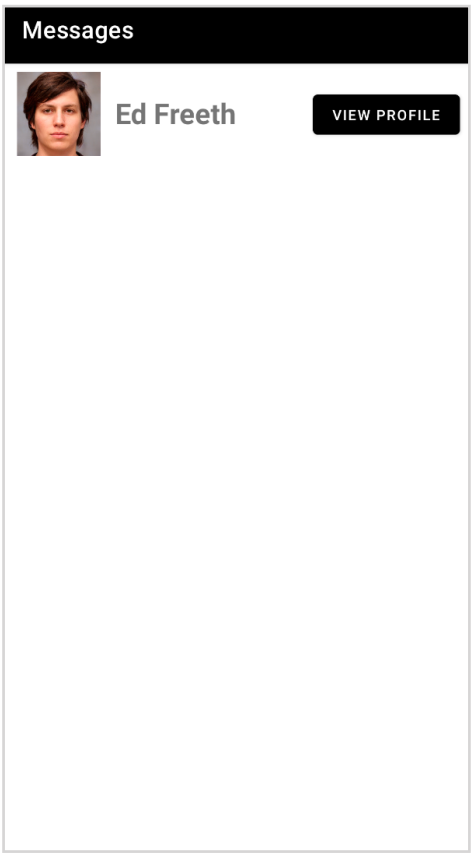
MusicianMessages (com.sam.bandpal)

Interacts with:

MessengerAdaptor (userData)
User (userData)

Functionality and Data Structures

This class displays for a user the accounts (matches) they have connected with either by the search and match feature or by being found by that feature by another user. Each match is displayed as part of a module in the GUI. A RecyclerView is implemented so that there can be an unlimited amount of matches displayed. The RecyclerView connects to an object of the *MessengerAdaptor* class which handles all match user data that needs to be forwarded to the connected classes of *Chat* and *ViewMatchProfile*. The GUI modules are filled by a database call for the current user's connectedMatches array (held as a child of the current user's UID in the Users table of the database). This array is then matched against a snapshot of the Users table and all matches are outputted to the GUI.



As *MusicianMessages* is accessible from multiple points in the application (*ViewProfile* and *ViewMatchProfile*), Android's inherent back button is overridden so that it always takes the user to *ViewProfile* and never back to *ViewMatchProfile* (even when they've arrived from that activity). This is to prevent the user needing to press the back button numerous times just to get back to the home screen of *ViewProfile* once they've matched with another user. I did consider that even after deciding to match with another user the current user may want to view their match's profile again, which is why buttons to navigate to *ViewMatchProfile* are present on every module, withdrawing the need for use of the back button for this task. This method uses elements of code developed by Abhay Maurya[17].

[17] https://youtu.be/8Pv96bvBJL4

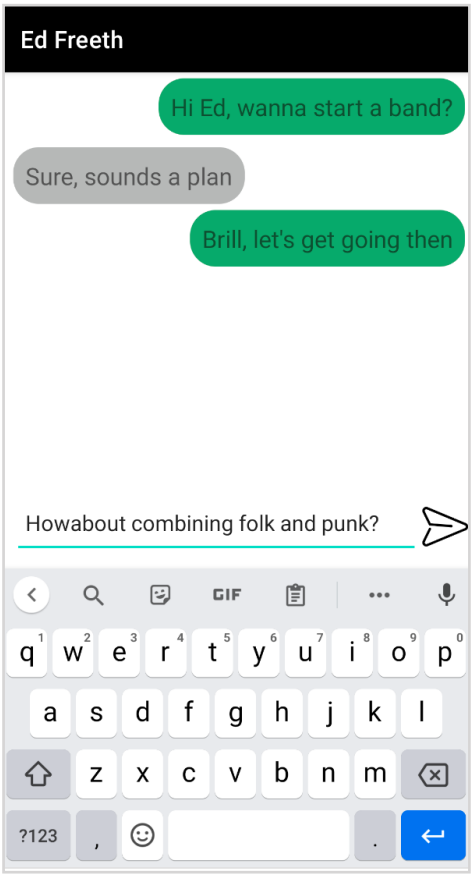
GUI

A RecyclerView allows for easy scrolling of the user's messages and matches. The activity is divided into modules, with each module afforded to one match of the user. Each module contains that match's display name, profile picture and a link to their full profile.

Chat (com.sam.bandpal)

Interacts with:

MessageAdaptor (userData)
Message (com.sam.bandpal)



Functionality and Data Structures

This activity is a messenger system, allowing users to directly communicate with each other. When *Chat* is navigated to from *MusicianMessages* it receives the UID and display name of the match the current user wants to message (via the *MessengerAdaptor* class). The UIDs of the match and current user are saved to constants of receiver and sender respectively and a 'room' of type string is declared for both. These unique rooms are supplied to the database as headers, for all messages to sit under as children. Both of these rooms sit under the main header in the database of allChats as children themselves. Individual messages saved in this manner take the form of the *Message* class, containing the message itself and the UID of the sender. These individual messages form part of the messageList ArrayList, which is fed to *MessageAdaptor* for handling. Like MusicianMessages, this method uses elements of code developed by Abhay Maurya.

GUI

This user interface was the last to be designed for the project, and is probably the simplest, as I believe a messenger should not feel cluttered, but display the information you need clearly and plainly. Following on the standard set by WhatsApp and other popular messenger services, sent messages are green and received are grey.

Database Design

BandPal uses Google Firebase as its database. Unlike the PostgreSQL databases we have used on the course, Firebase uses a NoSQL format that does not contain tables or records. Instead, data within the database is organised as a cloud-hosted JSON tree[18] full of data as JSON objects that grows downwards as data is added. When data is added it becomes a node (aka 'child') of whatever parent it is programmed to be under. Each piece of data has an associated key and value (like in hash structures).

I opted to have each crucial data element of the application heading up a parent-child structure of its own. In an SQL database these would be individual table names. I will now go through each of these six main headers (or 'parents') and discuss my design decisions for their implementation.

GenreStore

This header has one 'intermediary' child header of *genres* to allow easier access from the application (I did not find accessing a main header's contexts directly intuitive, through fault of either Firebase's design or my own coding shortcomings). Under the child *genres* sits the list of genres able to be added to a user's profile in the *EditProfile* activity.

Notification

Notifications in BandPal are used when one user matches with another via the *ViewMatchProfile* activity. They're automatically sent to the match's phone if they are using the application in the background (or not at all). Each child of *Notifications* is headed by a unique string, which contains children pertaining to the notification's receiver id, text and title.

Usernames

Much like *GenreStore, Usernames* is a simple list. It contains an easily accessible collection of all user UIDs that can added to and downloaded for iterating, key to the central search and match algorithm. The header has two intermediary children, *allUIDs* and *uidsOfUsers*.

Users

The most important part of the database, containing all user data. The structure is simple, with each child of the main header being a user's UID. This UID uniquely identifies the user (as it is a unique string) and its children are all strings, lists, doubles, integers or objects (containing the same variable types) pertaining to the information supplied in *EditProfile*, *Search* and *ViewMatchProfile*.

allChats

This section contains details of all messages exchanged between users. Every direct child of the main header is a unique string which is made up of the combined UIDs of the two users who have connected. The design of BandPal's chat function supposes that for each chat (i.e. each series of messages between two users) both users act as both senders and receivers (depending on which perspective you're looking from). This means that rather than having one child per chat, in which all data for that chat sits, there is instead two copies of the chat data, one for one user and one for another. Each individual message sits under its own unique identifying string, with children being the id of the sender and the string of the message itself.

[18] https://firebase.google.com/docs/database/web/structure-data

tokens

The last main header of the database is a list of unique tokens, all corresponding to currently open sessions (i.e. a list of who is currently using the application). Aside from being potentially useful for monitoring user metrics, the tokens stored in *tokens* are used to identify matches a user wishes to send a notification to.

Testing

Introduction

In testing my application I decided early on that as many of the tests as possible should directly correspond to BandPal's functional system requirements. I took a systematic approach to this task, combing through each activity and making sure all crucial functions were represented.

Objectives

- To ensure that the application works in the way it is intended, and free from any bugs or errors.
- To demonstrate to all potential stakeholders app functionality.
- To prove that the functional and non-functional requirements of the application have been met.

Strategy

I adopted manual testing as the primary testing method. All of these listed below were done (and any modifications necessary made) after core functionality of the application was complete. Additional to this formal testing however was my constant use of Android Studio's debugging tool (allowing me to test code in one line increments) and console logging (using print statements to show what is currently inside important variables and methods line by line).

Cases

Login

Corresponding Requirement ID	Description	Data	Expected	Actual
1	Display login screen on first time opening of app		*Login* screen is shown	**CORRECT**
2.2	Signup with pre-existing Google account	Username: John Smith Email: dankstandards@gmail.com	Google API is connected to, account verified, added to Authentication portal on Google Firebase and user navigated to *ChooseProfileType* screen	**CORRECT**
2.3	Signup with pre-existing Facebook account	Username: Solomon Frank Email: samuelcommander94@hotmail.co.uk	Facebook API is connected to, account verified, added to Authentication portal on Google Firebase and user navigated to *ChooseProfileType* screen	**CORRECT**

Corresponding Requirement ID	Description	Data	Expected	Actual
2.1	Login with pre-existing BandPal account	Email: aparanoidking@gmail.com Password: afalseknight9	If account has been authenticated but no profile has previously been created: User navigated to *ChooseProfileType* screen If account has been authenticated and profile has been created: User navigated to *ViewProfile* screen	**CORRECT**
3	Navigate to *Signup* screen		User is navigated to *Signup* screen	**CORRECT**

Signup

Corresponding Requirement ID	Description	Data	Expected	Actual
4	Signup with email and password	Email: testing1@gmail.com Password: testpass1!	Account created and added to Authentication portal on Google Firebase, user navigated to *ChooseProfileType* screen	**CORRECT**
5	Inability to signup if no password supplied	Email: testing2@gmail.com	Account unable to be created and appropriate message displayed	**CORRECT**
6	Inability to signup if passwords do not match	Email: testing2@gmail.com Password 1: testpass2! Password 2: testpass3!	Account unable to be created and appropriate message displayed	**CORRECT**

ChooseProfileType

Corresponding Requirement ID	Description	Data	Expected	Actual
9	Choose ‘Musician’ type and navigate to *EditProfile* screen		Navigated to *EditProfile* and intent of ‘Musician’ passed	**CORRECT**
9	Choose ‘Band’ type and navigate to *EditProfile* screen		Navigated to *EditProfile* and intent of ‘Band’ passed	**CORRECT**

EditProfile

Corresponding Requirement ID	Description	Data	Expected	Actual
13	Attempt to create a profile with profile type as ‘Musician’ and all details filled except influences, instruments and genres by pressing the ‘Save’ button	Profile photo: image in png format Display name: Testing 1 Bio: A bio. Age-bracket: 18-25 Website URL: www.emergency-ration-ensemble.bandcamp.com/ Location (auto-collected by app): 52.4833626 -1.94956726	Error message displayed, profile not created	**CORRECT**

Corresponding Requirement ID	Description	Data	Expected	Actual
13	Create a profile with all elements filled by pressing the ‘Save’ button, note the data contains one instrument and five genres	Profile photo: image in png format Display name: Testing 1 Bio: A bio. Age-bracket: 18-25 Website URL: www. emergency-ration-ensemble.bandcamp.com/ Location (auto-collected by app): 52.4833626 -1.94956726 (Rotton Park) Influences: Hole Dweller Fen Walker Spellbound Mire Deep Gnome Lost Fellow Instruments: Synthesizer Genres: Electronic Ambient Metal Grime Lo-fi	Profile successfully created, user data saved to database under *Users* -> ~User UID~ and user navigated to *ViewProfile*	**CORRECT**
16	Sign out of app by pressing the ‘Sign out’ button		User is navigated to *Login* screen	**CORRECT**

ViewProfile

Corresponding Requirement ID	Description	Data	Expected	Actual
18	View user profile information	Profile photo: image in png format Display name: Testing 1 Bio: A bio. Age-bracket: 18-25 Website URL: www. emergency-ration-ensemble.bandcamp.com/ Influences: Hole Dweller Fen Walker Spellbound Mire Deep Gnome Lost Fellow Instruments: Synthesizer Genres: Electronic Ambient Metal Grime Lo-fi	On navigating to *ViewProfile* all user data must be displayed, including the trophy Genrehead	**CORRECT**
22	View ‘Trophy Case’ dialog box by pressing ‘What’s this?’ button		Dialog box opens to explain the trophy system	**CORRECT**
19	Navigate to *Search* screen by selecting the search icon		Navigates the user to the *Search* screen	**CORRECT**
20	Navigate to *MusicianMessages* screen by selecting the messages icon		Navigates the user to the *MusicianMessages* screen	**CORRECT**
N/A	Navigate to *EditProfile* screen by selecting the cogwheel icon		Navigates the user to the *EditProfile* screen	**CORRECT**

Search

Corresponding Requirement ID	Description	Data	Expected	Actual
25	View user influences and genres	Logged in as: Sam C (located in Rotton Park)	On navigating to *Search* the user's influences and genres must be displayed. In this case they are: Influences: Pavement Slint Duster Hole Dweller Erang Genres: Post-Rock Indie Post-Punk Folk Psychedelic	**CORRECT**
23, 24	Conduct a 'Search for a BandPal' by supplying details and selecting the 'Search for a BandPal' button	Logged in as: Sam C Instrument: Piano Mile radius: 10 Maximum age: 26-35	Navigates the user to the *SearchResults* screen with the search terms taken into account	**CORRECT**

SearchResults

Corresponding Requirement ID	Description	Data	Expected	Actual
26, 27, 28, 29	Display results from 'Search for a BandPal' test made in *Search*	Logged in as: Sam C Instrument: Piano Mile radius: 10 Maximum age: 26-35	Displays the profile information of the most fittings matches, in this case Mono Works and Ed Freeth	**CORRECT**

Corresponding Requirement ID	Description	Data	Expected	Actual
26, 27, 28, 29	Display results for a second ‘Search for a BandPal’	Logged in as: Sam C Instrument: ‘Anything’ Mile radius: 10 Maximum age: 26-35	Displays the profile information of the most fittings matches, in this case Mono Works, Ed Freeth and Otis Mensah	**CORRECT**
26, 27, 28, 29	Display results for a third ‘Search for a BandPal’	Logged in as: Sam C Instrument: ‘Anything’ Mile radius: 30 Maximum age: 36-45	Displays the profile information of the most fittings matches, in this case Mono Works, Ed Freeth, Otis Mensah and Symbol Soup	**CORRECT**
30	Navigate user to other user Mono Works’ profile on selection of ‘View Profile’ button		Navigates the user to the *ViewMatchProfile* of user / potential match Mono Works	**CORRECT**

ViewMatchProfile

Corresponding Requirement ID	Description	Data	Expected	Actual
31, 33	View user profile information of user Mono Works	Logged in as: Sam C User's profile being viewed: Mono Works Profile photo: image in png format Display name: Mono Works Age-bracket: 26-35 Miles away: 2.9 miles away Trophies: Genrehead Bio: Weaving melodic guitar loops, sampled beats and reflective lyrics to create emotive sound maps. Website URL: www.monoworks.bandcamp.com Influences: Pavement Hole Duster Slint Rodan Instruments: Electric Guitar Piano Genres: Electronic Indie Post-Punk Dub Techno	On navigation to screen, user profile of user Mono Works is viewable	**CORRECT**

Corresponding Requirement ID	Description	Data	Expected	Actual
N/A	Navigate to *ViewProfile* screen by selecting the 'Own Profile' button		Navigates the user to the View*Profile* screen, i.e. back to their own profile / homepage	**CORRECT**
N/A	View description of trophy 'Genrehead' by selecting icon		Displays a toast message detailing what the user Mono Works was awarded the trophy Genrehead for	**CORRECT**
N/A	Connect with user by selecting message icon and verifying from the popup dialog box to go ahead		Sends a notification from the user to the pMatch's phone inviting them to connect, as well as adding both users to each other's connected matches, thus allowing them to see the other in their *MusicianMessages* screen	**CORRECT**

MusicianMessages

Corresponding Requirement ID	Description	Data	Expected	Actual
35	On navigation to the screen, display the user's connected matches	Logged in as: Sam C Connected matches: Ed Freeth	Display connected match Ed Freeth's profile picture and display name	**CORRECT**
36	On selection of 'View Profile' button next to connected match Ed Freeth's profile picture navigate user to their *ViewMatchProfile*	Logged in as: Sam C Connected matches: Ed Freeth	Navigates the user to the *ViewMatchProfile* screen with user Ed Freeth's details displayed	**CORRECT**
37	On selection of connected match Ed Freeth's profile picture navigate user to the *Chat* screen with Ed Freeth	Logged in as: Sam C Connected matches: Ed Freeth	Navigates the user to the *Chat* screen with user Ed Freeth as the user to be messaged	**CORRECT**

Corresponding Requirement ID	Description	Data	Expected	Actual
N/A	On navigation to the screen from *ViewMatchProfile* when back button selected, take user to *ViewProfile*, not back to *ViewMatchProfile*		Navigate user to *ViewProfile*	**CORRECT**

Chat

Corresponding Requirement ID	Description	Data	Expected	Actual
38, 39	Send message to connected match Ed Freeth	Logged in as: Sam C Connected match to message: Ed Freeth Message: Hi Ed, wanna start a band?	Message is sent from user Sam C to user Ed Freeth, being displayed on both their *Chat* screens	**CORRECT**
38, 39	Send reply message to connected match Sam C	Logged in as: Ed Freeth Connected match to message: Sam C Message: Sure, sounds a plan	Message is sent from user Ed Freeth to user Sam C, being displayed on both their *Chat* screens	**CORRECT**

Evaluation

Search and Match Algorithm

The search and match algorithm makes up the most important functionality of the application. It was the single largest and hardest engineering challenge of the project. I am extremely pleased that it works in the way intended, however it is not without its faults, which I will evaluate here.

General Achievements

As stated, the very fact the algorithm functions as I set it out to in my original proposal is a considerable achievement in itself. However, to analyse further I believe that the accomplishments of the algorithm are in its clear and well-defined structure, accuracy and reliability.

1) Modularisation

   The dissection of the overall algorithm into manageable methods that all chain together by way of parameters makes for a clean and simple architecture. This is further improved by the way that, as a search algorithm, the parameter passed is always a list of pMatch UIDs that gradually gets whittled down until only the top five options (or less) are remaining. Each method has a clearly defined role and there is very little repeated functionality between them.

2) Points-based structure

   With the algorithm adopting a points-based system that moves through methods concurrently (as described in **Implementation's** *SearchResults* section) I believe it showed good engineering foresight to order the methods by 'hard' requirements and 'soft' requirements. This means that as the pMatch UID list moves through the methods, it meets first those requirements which a pMatch *necessarily must* meet if it is to be considered further. These are mile radius, instrument played and age bracket, which are specified by the user in the *Search* activity previously. This saves a decent chunk of potential extra code, as it's opposed to doing all of the processing for soft and hard together, creating a messy and potentially inconsistent system.

3) Accuracy

   The algorithm is very accurate. If two users have some matching influences and genres, then they are guaranteed to be considered as a pairing by the algorithm, only being looked over if enough better pairings are present. This achievement is less down to the design of the algorithm itself, and more about the way that the data being taken in by it is made cohesive with itself. My decision to unify all spellings and capitalisations of instruments, genres and artists via API libraries is responsible for this accuracy.

General Shortcomings

1) Nuanced matching

   As the algorithm relies on direct String matching to pair users, any deviation from this will not result in a potential pairing gaining any 'points' to aid its standing in the results. This is a problem. For example, if a user is particularly into nineties Indie Rock music then their five artists might be Pavement, Modest Mouse, Built to Spill, Rodan and Pixies. A pMatch in their local area may also be into this style, but instead lists the artists of Guided by Voices, Sparklehorse, Dinosaur Jr., Slowdive and R.E.M. on their profile. Now, despite the huge

amount of crossover between these ten bands, not one single point will be gained for the likelihood of the pairing to be a good one, despite the fact that to any human observer it clearly is. Furthermore, if another pMatch had a list of artists that contained only one match with the user and the rest were of totally different styles then this pairing of users would appear higher in the results than the previously mentioned 'nineties Indie Rock' example despite being (all things considered) an inferior one. In a way the matching of genres is a balm to this problem, as they're far more general (and there are less options), but the problem still stands. To see a potential fix for this, see the **Potential improvements** section below.

2) Scalability

The following pseudocode helped me to analyse the Big O time complexity of the algorithm and assess its scalability. To prevent an overabundance of unnecessary detail I have removed most declarations. Variables should be named clearly enough that their purpose and type is obvious. It's also important to note that my perception of each method's time complexity are estimations.

Pseudocode - pullUsernames()

```
pullUsernames()
{
~Downloads list of user UIDs and passes them to next method~
}
```

Time complexity = O(n)

Pseudocode - matchinstrMileAge(uidList)

```
matchinstrMileAge(uidList)
{
        ~Boolean declarations of instrMatch, distanceMatch and ageMatch~

        ~Downloading user data to variables~

        ~Declaring of passedList which contains all pMatch UIDs that pass boolean checks~

        // Loops list of user UIDs
        for(pMatchUID in uidList){

                ~Downloading pMatch user data to variables~

                // Boolean true if instrument match or no instrument searched
                for(instrument in pMatchInstruments){
                if (instrument == searchedInstr || searchedInstr == "anything") {
                        instrMatch = true
                    }
                }

                /*
                Boolean true if distance between user and pMatch
                less than or equal to searched radius limit
                */
                val distance = distanceBetween(userLat, userLong, pMatchLat, pMatchLong)
                if (distance <= searchedRadius){
                    distanceMatch = true
                }

                // Boolean true if ages align
                if (searchedAgeBracket == pMatchAge ||
                    searchAgeBracket == "60+" ||
                    searchedAgeBracket >= pMatchAge){
                    ageMatch = true
                }

                /*
                If booleans are true, UIDs are not identical and pMatch is not a band profile
                pMatchUID is added to passedList
                */
                if (pMatchUID != userUID
                    && pMatchProfileType != "Band"
                    && ~all booleans == true~){
                    passedList.add(pMatchUID)
                }

                // Call matchInfluencesAndGenres on loop finished, pass passedList
                if(~loop complete~){
                  matchInfluencesAndGenres(passedList)
                }
        }
}
```

Time complexity = $O(n^2)$

Pseudocode - matchInfluencesAndGenres(passedList)

```
matchInfluencesAndGenres(passedList)
{
        // General declarations
        var pMatchScore = 0.0
        val pMatchAndScore = hashMapOf<Double, PMatch>
        val indexesOfMap: MutableList<Double>
        ~Downloading user data to variables~

        // Loop through passedList of pMatchUIDs
        if (passedList != null){
            for (pMatchUID in passedList){

                ~Downloading pMatch user data to variables~
                ~Declaring MutableLists of matchedInfluences and matchedGenres~

                // For every artist match, pMatch gains a point
                ~nested for loop of pMatchInfluences against userInfluences~{
                  if (~match found~){
                      pMatchScore++
                  }
                }

                // For every genre match, pMatch gains a point
                ~nested for loop of pMatchGenres against userGenres~{
                  if (~match found~){
                      pMatchScore++
                  }
                }

                /* TROPHIES
                ~If pMatch has Maestro trophy (plays 5 instruments) pMatchScore++~
                ~If pMatch has Genrehead trophy (5 genres on profile) pMatchScore++~
                ~If pMatch has First Match trophy pMatchScore++~

                // Random double below 1 to break ties
                pMatchScore += rand ~rand = Random.nextDouble(0.00001, 0.99999)~

                // Adding final pMatchScore to list
                indexesOfMap.add(pMatchScore)

                /*
                Making object of pMatch's data and adding to HashMap
                with pMatchScore as key
                */
                pMatchAndScore.add(key = pMatchScore, value = PMatch(~data~))

                // Resetting pMatchScore for next loop iteration
                pMatchScore = 0.0

                // On end of loop, converting pMatchAndScore HashMap
                // to SortedMap by descending key value and doing the
                // same with indexesOfMap
                if (~loop finished~){
                val sortedPMatchAndScore = pMatchAndScore.sorted
                val sortedIndexesOfMap: SortedSet = indexesOfMap.sorted
                outputPMatches(sortedPMatchAndScore, sortedIndexesOfMap)
                }
                }
        }
}
```

```
data class PMatch(
    val uid: String,
    var matchedInfluences: MutableList<String>,
    var matchedGenres: MutableList<String>,
    val distanceToPMatch: Double,
    val imageUrl: String,
    val displayName: String
)
```

Time complexity = not good, potentially higher than $O(n^2)$

Pseudocode - outputPMatches(sortedPMatchAndScore, sortedIndexesOfMap)

```
outputPMatches(sortedPMatchAndScore, sortedIndexesOfMap)
{
        ~outputs pMatches to GUI by iterating sortedIndexesOfMap which
         gives keys to get sortedPMatchAndScore's values~
}
```

Time complexity = O(n)

Ultimately, with a time complexity of $O(n^2)$ or potentially higher the algorithm is (arguably) disqualified from being able to scale. This is due to the nested loops in **matchinstrMileAge()** and **matchInfluencesAndGenres()** that first iterate a list of pMatch UIDs, and then do further iterations to find influence and genre matches within that list loop. Despite this, the algorithm works just fine for small batches, and for the purpose of the project fulfils the requirements set out in my proposal.

Potential improvements

1) Nuanced matching

   The problem with the rigid string matching system is caused by the denial of influence or genre matches that are not 100% accurate. This could potentially be fixed by putting a labelling system in place, whereby each influence and genre is part of a data structure that maps links between similar artists and styles. There could be degrees of this, with the similarity between two artists being on a scale. So for example if a user likes The Beatles and a pMatch likes the Rolling Stones then despite these not being the same artist, the algorithm would detect their similarity and would give, say, half a point for the pairing (as opposed to a full one for if both user and pMatch liked The Beatles). Similarly if the user liked The Beatles and the pMatch liked Grime rapper D Double E then there would be no point added, because their music is very different.

   To achieve this improved matching the application would need access to some kind of labelling system which would require the interaction with a lot more data than the project had access too, though this could potentially be achieved with a commercially available API.

2) Scalability

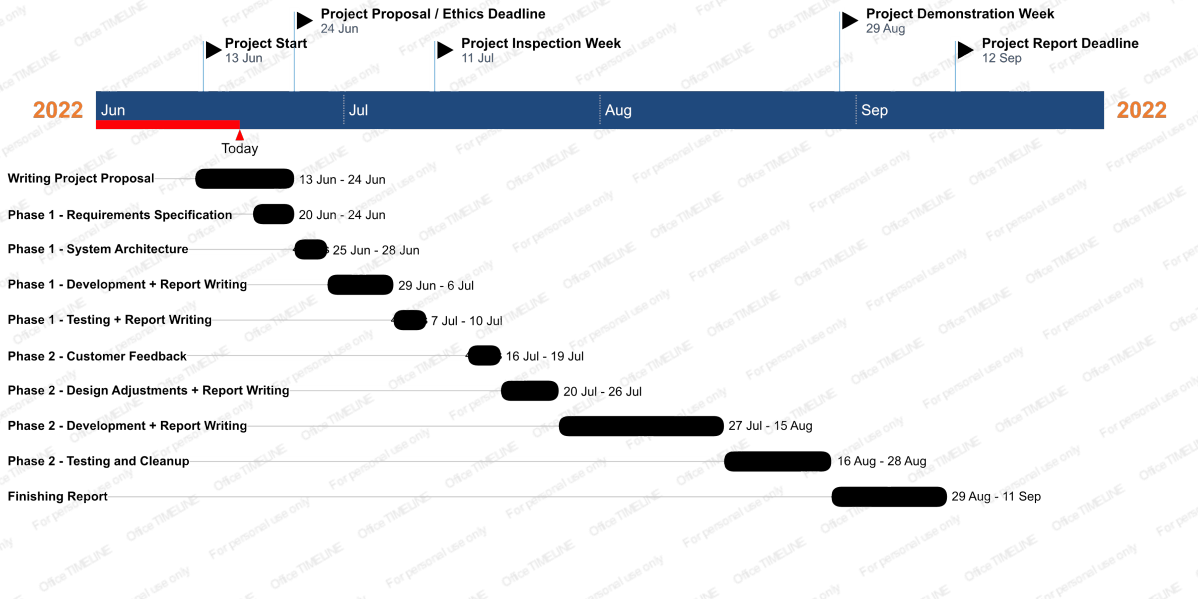
   As mentioned, the time complexity of the algorithm falters when it encounters nested loops which iterate artists and genres against each other, looking for matches. A way to improve this would be to feed into the nested loops lists of influences and genres that are alphabetised or contain accompanying numbers corresponding to specific elements, so that an O(log n) binary search could speed up the search time. However this would mean labelling the artist and genre data (i.e. re-indexing it), and since the artists in particular come from an API the modification of this data would require a significant new section of code within the app and potentially a way to store this newly indexed data as the API would not immediately contain it.

   Another way the time complexity falters is with the need to download and iterate through the entire app's list of UIDs for each run of the search and match algorithm. A way to improve this would be via database management. Instead of having one big list of user's UIDs, the database could store users based on their last known location, this way when the search algorithm is run, only those pMatches closest to the user would be downloaded into the original pullUsernames() method. This wouldn't improve the Big O complexity of O(n),

but it would improve the overall time to complete, as a local UID list is far smaller than the entire app's one.

Project Management

Originally (and in my proposal) I adopted an agile approach to my project's management. As you can see from the diagram below, my plan was to build the application not in modules (like say: login, signup and choose profile type in one module) but to build rudimentary versions of the entire app, in sequence, which gradually get more and more features added to them.



However I quickly realised that I had a pretty clear idea of what pages I wanted the app to have, and what features those pages needed to implement, despite not knowing exactly how each individual page's functionality would be structured. This led me to do a bit more research on the agile approach, and realised it is more suited to engineers working with stakeholders who do not know exactly what form they want their app to take, or what functionality it will have, they just know how the app should *feel* or the purpose it should fulfil. This was near opposite to my own circumstances, where I knew from the start what pages and functions my application needed to implement.

Therefore, I decided to instead adopt a waterfall style approach to the engineering, whereby I began at the beginning and went on till I got to the end. The order of the screens I built was natural, as they all followed on from one another in the user's journey. I didn't begin a new screen until the one that led to it was complete (but not fully tested), or at least very nearly complete. Once my potential stakeholder interviews were done, and the project was mapped out at a high-level, I started with the *Login*, *Signup*, *ChooseProfileType* and *EditProfile* screens, then moved onto *ViewProfile*, the search and match algorithm etc. finishing with the messaging system. Although

some tests were made incrementally as I went along, the bulk of them occurred once the app was nearing completion, as laid out in the **Testing** section of this report.

That isn't to say that there were no features that were retroactively applied. For example the trophy system was added after the *ViewProfile* section had already been thought complete, and the notification capabilities were added after the messaging system was already in place.

Future Plans

Although I enormously enjoyed creating BandPal and learning all the new skills necessary to make it a success, I won't be scaling it into a publicly available app. This is mainly due to time constraints, as I'm moving on from the MSc to be a Systems and Software Engineer professionally. However, if I were to continue the project further down the line, these are a few briefly outlined features I would like to include:

- A pitching feature
  This was mentioned as a possible function in my project proposal. It would build on the community aspects of the application by allowing those using the application under the 'Band' profile type to pitch their music to a raft of local venues.

- Expanding the trophy system
  The awards mechanism implemented to both gamify the user's experience and aid their interactions via the search and match algorithm was a later addition to the project. It was not mentioned in the proposal, but when conceived it quickly became a core part of BandPal's functionality. It would be great to expand the amount of trophies on offer. The most obvious would be a 'globetrotter' trophy for searching / matching in multiple locations, then there could be a 'fasthand' award for particularly quick responders to their connections. There's also the potential for a trophy for consistent usage of the application, though this may be hard to gauge.

- Adding galleries and videos
  Some of my original stakeholder feedback was that of making Bandpal an expansive media platform. In its current form, there is only functionality for one profile photo per user (which can be changed) and no videos. However, some kind of media showcase on *ViewProfile* and *ViewMatchProfile* involving galleries of live photos, live videos, audition tapes and studio sessions could really add not just a cosmetic but tangible boost to the app's user experience.

- Quality of life changes
  The app doesn't contain a way to change your account password, authenticate your account via email or force you to have a password that conforms to particular security standards. These would be nice 'quality of life' changes to make in order to boost user experience.

Conclusion

The ultimate problem BandPal was trying to solve was this: Had the application been available during my younger years, would I and my friend (having similar musical interests and location) been able to speed up our meeting? I think the answer is yes. The application meets and exceeds this core purpose laid out in my proposal, with new features and functionality like the trophy system and phone-to-phone notifications being added too. There is also plenty of room for expansion, as laid out in the **Future Plans** section.

However, as discussed in **Evaluation** I do not believe in its current form that the app would be able to scale, due to its infrequent but necessary use of nested loops, bringing a higher time complexity to the central search and match algorithm than desired. Fortunately though, I believe this is just evidence of my relative youth as a programmer, and not an indication that the whole idea is unscalable. As I've laid out previously in a high-level manner, there are fixes I could pursue from the engineering side to make the app possible for wider use.

Ultimately, I am extremely proud of what I have managed to accomplish in the limited time frame available. Building an application from scratch with no prior knowledge of the main language, database system or mobile environment has buoyed me on for the future, and proven to myself that I can learn new skills quickly, while implementing them in a meaningful project.

Bibliography

1. DataReportal – Global Digital Insights (n.d.) *60% of the World's Population Is Now Online* [online] Available at: https://datareportal.com/reports/6-in-10-people-around-the-world-now-use-the-internet#:~:text=As%20we%20revealed%20in%20our [Accessed 6 Sep. 2022].

2. Market Business News. (n.d.) *What is social media? The effects of social media* [online] Available at: https://marketbusinessnews.com/financial-glossary/social-media-definition-meaning/ [Accessed 6 Sep. 2022].

3. Reed, N.I. (2019) *Is developing for desktop dead?* [online] Available at: https://productcoalition.com/is-developing-for-desktop-dead-f8d7ba9fd180 [Accessed 6 Sep. 2022].

4. Smart Insights (2018) *The exodus from desktop to mobile: Why it matters and what to do about it* [online] Available at: https://www.smartinsights.com/mobile-marketing/mobile-marketing-strategy/from-desktop-mobile-why-matters [Accessed 6 Sep. 2022]

5. Enge, E. (2021) *Mobile vs. Desktop Usage in 2020* [online] Available at: https://www.perficient.com/insights/research-hub/mobile-vs-desktop-usage#:~:text=Globally%2C%2068.1%25%20of%20all%20website [Accessed 6 Sep. 2022].

6. StatCounter (2021) *Mobile Operating System Market Share Worldwide* [online] Available at: https://gs.statcounter.com/os-market-share/mobile/worldwide [Accessed 6 Sep. 2022].

7. Manchanda, A. (2022) *The 7 Best Programming Languages to Write & Develop Native Android Apps* [online] Available at: https://www.netsolutions.com/insights/best-programming-languages-to-write-develop-android-apps/#:~:text=Frequently%20Asked%20Questions- [Accessed 6 Sep. 2022].

8. Lardinois, F. (2019) *Kotlin is now Google's preferred language for Android app development* [online] Available at: https://tcrn.ch/2vJMMO2 [Accessed 6 Sep. 2022].

10. Android Studio (2019) *Download Android Studio and SDK tools* [online] Available at: https://developer.android.com/studio [Accessed 6 Sep. 2022].

11. Gluon (n.d.). *Scene Builder* [online] Available at: https://gluonhq.com/products/scene-builder/ [Accessed 6 Sep. 2022].

12. Pusher (n.d.) *Making PostgreSQL database realtime with Pusher and Node* [online] Available at: https://pusher.com/tutorials/postgresql-realtime/ [Accessed 6 Sep. 2022].

14. Google Developers (n.d.) *Integrating Google Sign-In into Your Android App | Google Sign-In for Android* [online] Available at: https://developers.google.com/identity/sign-in/android/sign-in [Accessed 6 Sep. 2022].

15. Facebook for Developers (n.d.) *Android - Facebook Login - Documentation* [online] Available at: https://developers.facebook.com/docs/facebook-login/android/ [Accessed 6 Sep. 2022].

16. Picasso (2013) *Picasso* [online] Available at: https://square.github.io/picasso/ [Accessed 6 Sep. 2022].

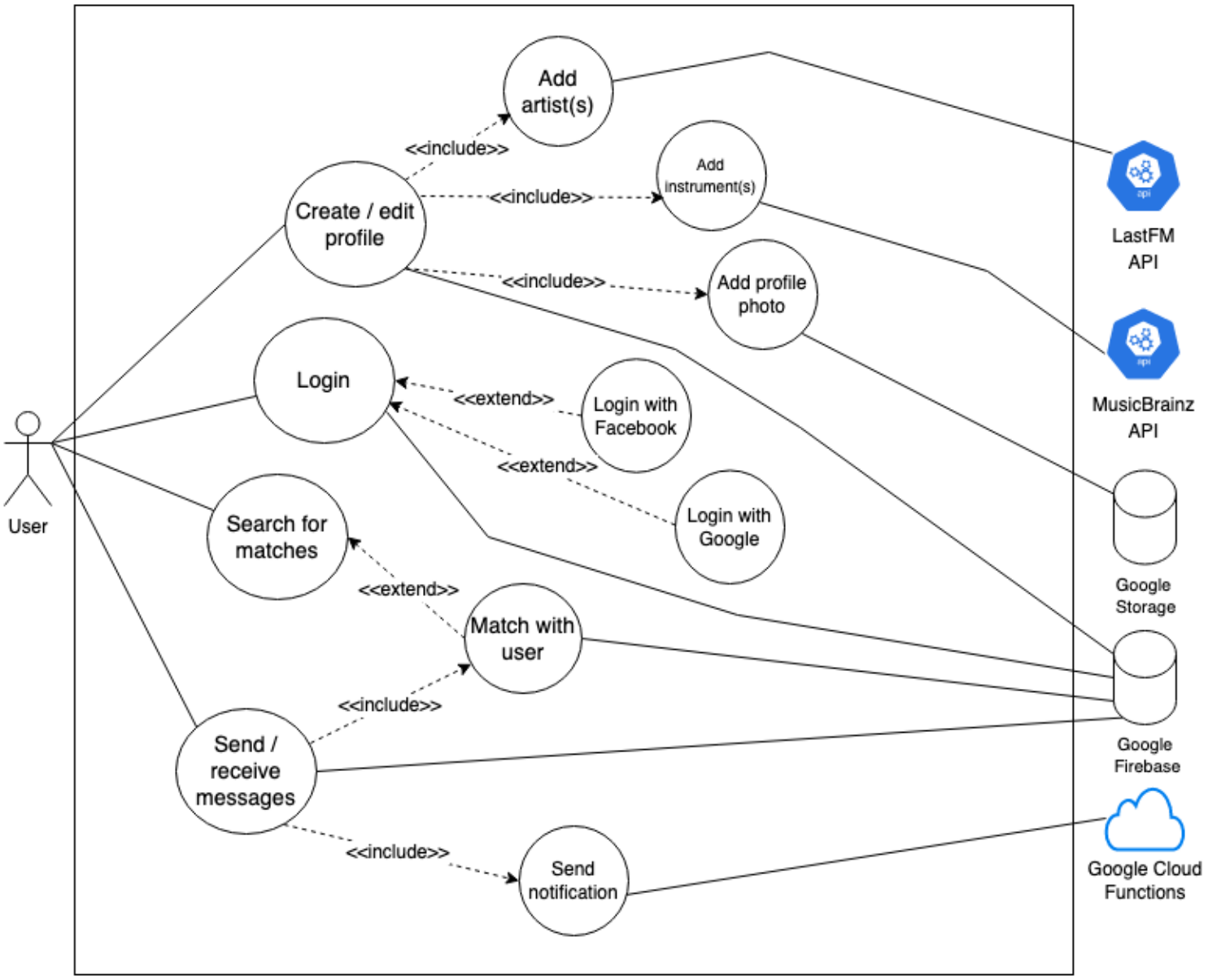
17. Nasr, M. (2021) *Device to Device Notification Firebase Android* [online] Available at: https://youtu.be/O3R4jaR7QpA [Accessed 6 Sep. 2022].

18. GeeksforGeeks (2021) *Creating Chatting Application in Android Studio Using Kotlin* [online] Available at: https://youtu.be/8Pv96bvBJL4 [Accessed 6 Sep. 2022].

19. Firebase (2019) *Structure Your Database* [online] Available at: https://firebase.google.com/docs/database/web/structure-data.

Appendix

Appendix A - Use Case Diagram



Appendix B - System Requirements

Functional

Login Screen

The first screen a new user will see, and a way to login to pre-existing accounts.

1. The application must display the landing page when the user opens the application for the first time.
2. Must display a visual menu which allows the user to sign into their account in multiple ways:

    2.1. Must allow the user to sign-in with their account's email address and password.
    2.2. Must allow the user to signup or sign-in with a pre-existing Google account.
    2.3. Must allow the user to signup or sign-in with a pre-existing Facebook account.

3. Must allow the user to navigate to the Signup page if they do not have an account.

Signup Screen

A signup screen for account creation.

4. Must allow the user to signup to BandPal.

    4.1. Must allow the user to input an email address.
    4.2. Must allow the user to input a password in two separate text boxes.

5. Must display a message to the user stating that their signup attempt has been unsuccessful and to try again if any of the boxes are blank.
6. Must display a message to the user stating that their signup attempt has been unsuccessful and to try again if the two password text boxes do not match.
7. Must allow the user to toggle the password boxes so that their prospective password can be hidden or visible while typing.
8. Must include a save / sign-in button that when tapped communicates with the Google Firebase server to save the new account's data.
9. Must allow the user to navigate to the Account Type Selection screen.

Account Type Selection Screen

Lets new users choose between account types. Only seen once on signup.

10. Must allow the user to select from three different types of account, being Musician, Band or Promoter[19].
11. Must include a save button that when pressed communicates with the Google Firebase server to save the account's data and navigates the user to the Edit Profile Screen.
12. Must contain a sign out button that navigates the user to the Login Screen.

Edit Profile Screen

Allows users to edit their public profile.

[19] The Promoter option is purely cosmetic. The only true options in this form of the application are Musician and Band.

13. Must allow users to modify their public profile and include a save button. Once the save button is tapped all modified elements are saved to Firebase and the user is navigated to the View Own Profile Page. Elements are:

    Profile photo
    Display name
    Bio (limited to 160 characters)
    Age-bracket (e.g. 18-25)
    Website URL
    Geolocation (this is obscured on public display)
    Musical influences (limited to 5 artists)
    Instruments played (limited to 5 instruments)
    Genres (limited to 5 genres)

14. The 'musical influences' text box must be connected to LastFM's artist search API, so that all artist names are formatted identically between user profiles.
15. The 'instruments played' text box must be connected to MusicBrainz's instruments API, so that all instrument names are formatted identically between user profiles.
16. Must include a sign out button.

View Own Profile Screen

Acts as the home page of the application.

17. If logged in user has accessed the application before and already created a profile, then this is the first screen they will see once tapping the application icon.
18. Must layout the user's public profile information in an informative and visually attractive GUI.
19. Must contain a search button that when tapped navigates the user to the Search Page.
20. Must contain a button that when tapped navigates the user to their messages.
21. Must contain a 'trophy case' that contains the user's trophies, these are:

    First Match - Awarded for making a first connection to another user
    Genrehead - Awarded for adding five genres to your profile
    Maestro - Awarded for adding five instruments to your profile

22. Must contain a button to launch a dialog box that explains the trophy system.

Search Screen

The page that lets users begin to interact with each other.

23. Must allow the user to 'search for a BandPal' by putting in their search requirements. These are:

    The instrument they want any prospective bandmate to play (supplied by MusicBrainz Instrument API). 'No instrument' must also be an option.
    How close the p.bandmate is to a user's own location. This will be shown as 'within 5 miles' etc.
    The oldest age bracket they want to be included in the search.

24. Must contain a 'search' button which once tapped communicates with Firebase to pull the necessary results and take the user to the results page.
25. Must display the user's influences and genres.

Search Results Screen

Displays the results of the search page in an easy to comprehend format.

26. The search result corresponds to a list of p.bandmates, ordered by best fit to the user's search requirements.
27. Each p.bandmate's profile must be summarised in a simple format with their details that match the user's requirements displayed clearly.
28. Each p.bandmate must have text next to them that shows how far away they are from the user, e.g. '~display name~ is ~n~ miles away'.
29. p.bandmates are ordered by how well they match the user's search terms and profile information.
30. Each p.bandmate can be tapped on to navigate the user to their public profile.

View Other User Profile Screen

Only accessible by a user from the Chat screen (if they have contacted the p.bandmate before) or via conducting a 'search for a BandPal'.

31. Must follow the same format as the View Own Profile Page (display, elements etc).
32. Must have a chat button that navigates the user to the Chat Page with the p.bandmate.
33. If the user navigates from a search and has not previously connected with the p.bandmate then their profile should display how far they are from the user in miles.
34. If the user navigates from the Chat screen then there should be a message showing that the p.bandmate is a match.

Messages Screen

A display of all the user's connected matches.

35. Matches' display name and profile picture must be displayed.
36. The user must be able to navigate to the View Other User Profile screen of any particular match.
37. The user must be able to navigate to the Chat screen with the match they selected.

Chat Screen

A rudimentary messaging function.

38. Must allow users to chat with each other.
39. Must work with Firebase to save and retrieve chat history.

Non-Functional

Usability

40. The system must allow the user to buttons or gestures to navigate between each of the tabs.
41. The system must clearly identify each tab and signpost all functionality with a visual icon and textual description.

42. The system must have a simple yet stylish graphical user interface that is easy to understand, use, and navigate.

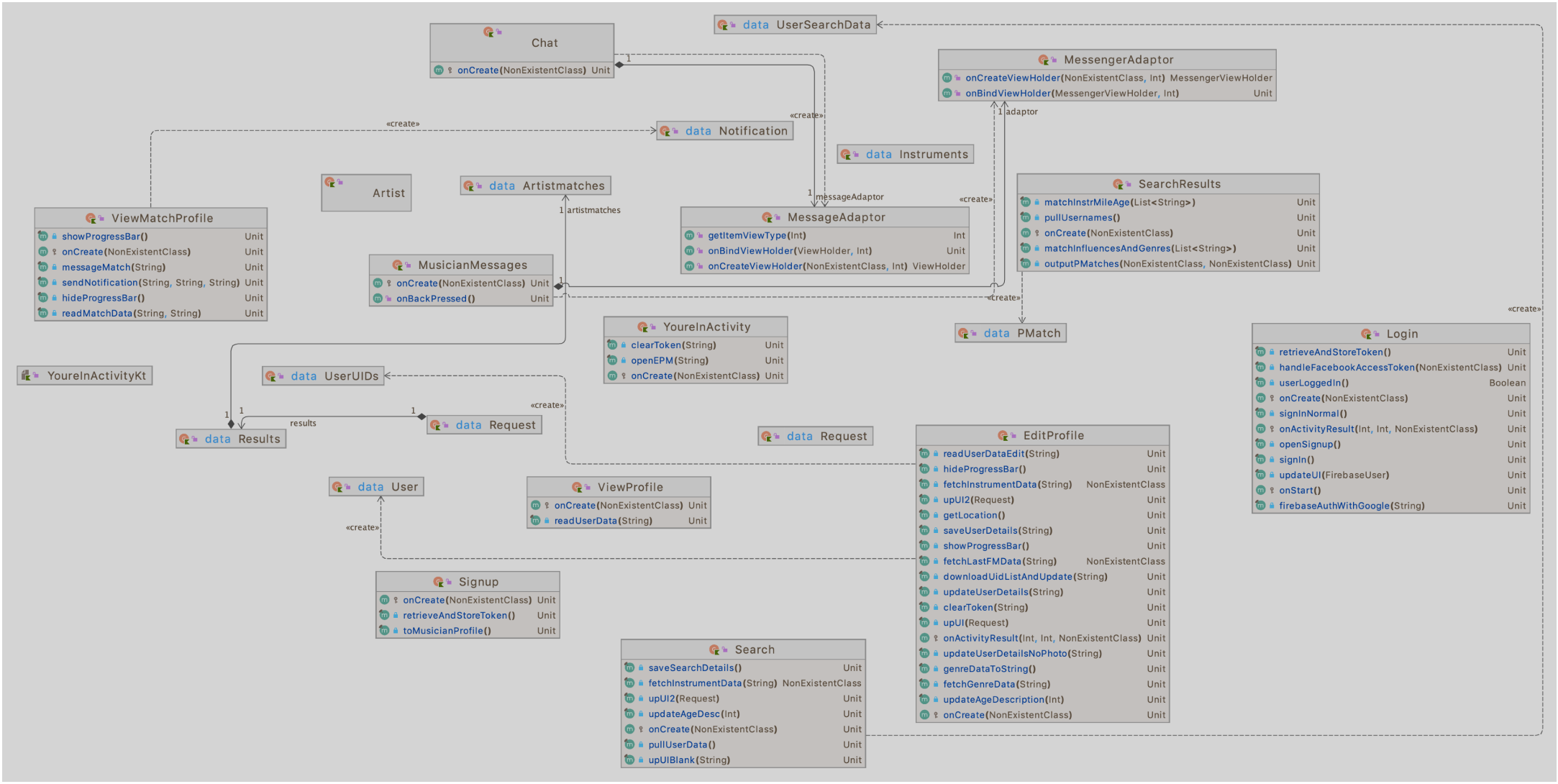
Efficiency (Space and Performance)

43. The application must have the necessary memory in the database to store user's profile information, password and email.
44. The application must have a download size of no more than 40 MB.
45. The application should display a 'connection lost' message to indicate there is an issue retrieving data, provided the user does not have a Wi-Fi or data connection or poor signal.

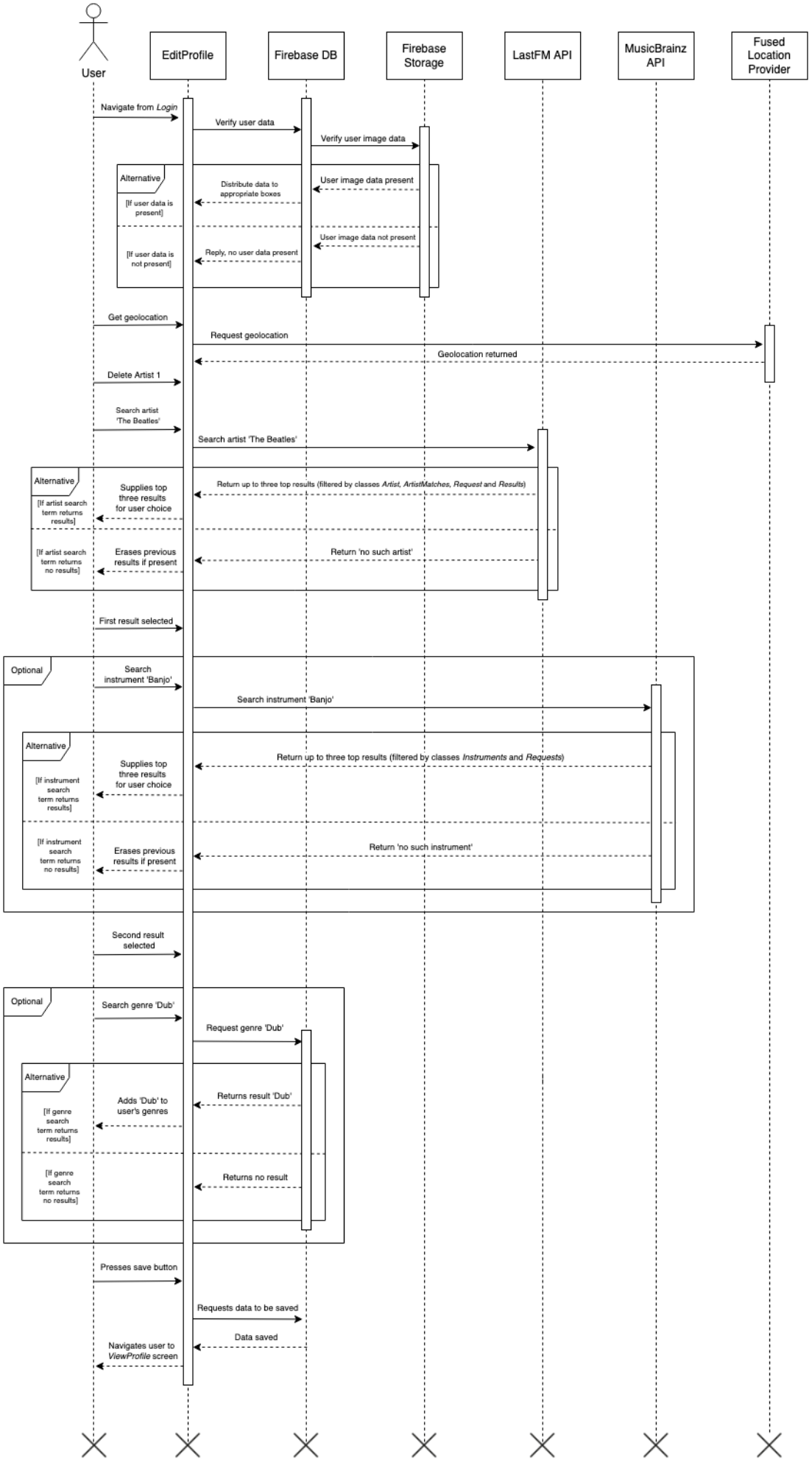
Security

46. The application must abide by UK data protection law for complete security and privacy of user data.
47. The application must abide by UK GDPR (General Data Protection Regulation) regulations.
48. The application must lock a user's account if a password is entered incorrectly more than four times.
49. The application must allow a user to delete their information, subsequently deleting the data from all databases.
50. The application must ensure the user's private or personal data is secure.

Appendix C - Class Diagram



Appendix D - Sequence Diagram



Appendix E - User's Guide to BandPal

Git repository location address:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2021/sxc1368

Git repository structure:

The git repository contains only the project files, no additional files are present.

How to run BandPal:

Clone the contents of the git repository into the Android Studio IDE and run the code on an Android mobile phone emulator (which can be downloaded within Android Studio).

For the best experience, I would like to request that the program is run on a Google Pixel 4 that *must* have Google Play enabled on the device. If Google Play is not on the emulator then certain features (like login via Facebook) will not work.

To assist with inspection of the application I have created several profiles (displayed on the next page) that I would recommend searching for via the central search algorithm (found via activities *Search* and *SearchResults*). These users accounts are all based in either Birmingham or the wider West Midlands, and can be matched with from any account created by a user that shares some match-worthy attributes with them. It is important to note that they are not the only user accounts in the database based in Birmingham.

Name, Age, Type	Location	Influences	Instruments	Genres	Trophies
Mono Works 26-35 Musician	Moseley	Pavement Hole Duster Slint Rodan	Electric Guitar Piano	Electronic Indie Post-Punk Dub Techno	Genrehead
Ed Freeth 26-35 Musician	King's Heath	The Cure Nap Eyes Shellac Alex G Pavement	Acoustic Guitar Piano	Electronic Indie Pop Blues Jazz	Genrehead First Match
Otis Mensah 36-45 Musician	Rotton Park	Milo Scallops Hotel Nas Four Tet Slint	Clarinet Piano Synthesizer Vocals Electric Guitar	Folk Electronic Trance Dub Reggae	Genrehead First Match Maestro
Symbol Soup 36-45 Musician	Telford, Shropshire	Black Sabbath Sparklehorse Wilco Rodan The Feelies	Accordion Dia Harmonica Piano	Dub Reggae Ska Trance	

Name, Age, Type	Location	Influences	Instruments	Genres	Trophies
Rodney Smith 46-59 Musician	Bearwood	Hole Dweller Pavement Steve Gunn Prince Princess Nokia	Electric Guitar Cello Drums Bass Guitar Piano	Rock Indie Folk	First Match
Aaberg 46-59 Musician	Dudley	Codeine Aphex Twin Lomelda Dinamarca Mapache	Piano Alto Saxophone	Folk Electronic Disco	