



Music Bingo

Jamie Power
C17316471

Submitted in partial fulfilment of the requirements for the degree
of
BSc in Business Computing

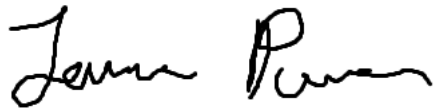
Technological University Dublin (City Campus)
May 2021

Supervisor: Farrah Higgins

Declaration

I hereby declare that the work submitted is entirely my own and that ideas or extracts from other sources are appropriately acknowledged and referenced.

Signed: Jamie Power

A handwritten signature in black ink, reading "Jamie Power". The signature is written in a cursive style with a large initial 'J' and 'P'.

Date: 30/04/21

Acknowledgements

I would like to take an opportunity to thank all of those who supported and encouraged me to stay motivated and to reach the end of this journey.

I would like to acknowledge the fantastic continuous support from my supervisor Farrah throughout the development of this project. Farrah consistently offered guidance and set targets for me, which offered an enjoyable sense of achievement and progress. I am very grateful for Farrah's input and I wish to thank her for all of her help.

I also want to offer a huge thanks to Jenny, my second reader, for pushing me to maintain a high standard within this project and offering a unique perspective that helped me reach my potential.

I would like to thank both Jenny and Farrah for their enthusiasm for my project and initial encouragement, which gave me the confidence to pursue something out of my comfort zone.

I also wish to thank my friends and colleagues within the college who assisted and supported me and motivated me to stay organised and on top of my work.

I would also like to thank friends and family of mine who consistently supported me over the last four years and helping me achieve my goals.

Finally, I would like to thank my girlfriend Zara, who gave me the inspiration for this project. Her input and support was a massive help throughout my four years in college.

Abstract

Music bingo is an adaption of the traditional bingo game, which utilises broadcasting randomly played songs instead of numbers where players would mark off a card containing song names again as opposed to numbers. The most common implementation of the game involves a third-party or manual set up of the game, which involves creating playlists, cards, and providing pens, a location to host, and an audio source to broadcast songs.

Music Bingo is an out-of-box multiplayer solution aiming to give the traditional bingo experience and enjoyment within an android mobile app. Within this report, I will discuss the business application of this solution as well as the development process, the underlying technology and an explanation of how the game operates.

Table of contents

Cover Page	1
Declaration.....	2
Acknowledgments	3
Abstract	4
Introduction	7
Objective	8
Business Case.....	9
Technology.....	10
Supporting Technologies.....	13
Requirements Analysis	14
Functional Requirements	14
Non-Functional Requirements	16
High-Level Use-Case	17
In-Game Use Case.....	18
System Design	19
Technologies.....	20
Database Structure.....	22
Software Design	25
User Interface Design	25
Implementation	26
Login and Registration	26
Homepage	27
Hosting	27
Connection	29
Playing Songs	30
Claiming Lines.....	31
False Claims	32
Text Chat.....	33
Reactions.....	35
Played Songs.....	35
Leader board.....	36
End Game	36

App Insights	37
Playlist Store	38
Profile	39
Music Catalog	40
Issues and Resolutions	41
Test plan	42
Summary	45
Conclusion.....	45
References.....	46

Introduction

The advent of COVID-19 was an incredibly disruptive force on the world, but It also highlighted how imperative technology was to connect people and keep the economy running. When lockdowns were implemented across the globe, it forced many people to devise clever alternatives for social interaction. Quizzes, bingo, and online gaming facilitated a massive demand for entertainment as all hospitality and events were suspended. Along with lockdowns came guidelines regarding social distancing placing a stronger emphasis on games being played remotely.

Prior to the global pandemic, music bingo was hosted by bars and nightclubs, there were also other events which played a variation known as "Bingo Loco" which gained immense popularity. Event management companies offer music bingo services which involves their own proprietary solutions, these companies often work in conjunction with publicans to host 'bingo nights' where they allocate cards and pens, create their own playlists as well as displaying songs and results on TV's or monitors within the establishment the game is hosted. The event-management companies delegate rewards to the clients which can often be in the form of cash prizes or drinks. The concept for the game is relatively new and has gained a lot of traction since its conception although it has yet to be implement in the form of a web application or mobile solution. Many management companies have turned to hosting the bingo events over video conferencing tools such as Zoom although these packages often carry a substantial hosting fee and players have limited influence over the specifications of the game and how it is played.

I decided to create music bingo as a convenient multiplayer solution to allow players to create games and to play them together in the same local network to avoid the need to buy online packages, wait for events to play or having the control of the game abstracted. Hosts can define the genre they want to play and start a game within seconds. The wireless connectivity allows players to enjoy a game together in a large room or outdoor setting while being able to maintain social distance. My anecdotal experience of these games has proven to be very enjoyable, and I had personally worked in an established which hosted them where I saw the success and popularity of these games. I believe this application resides in a niche of its own and its creation has potential to be successful.

Objective

The primary goal of Music Bingo was to create a concrete solution to adapt the traditional bingo variation game to a mobile application. I aimed to provide the playlist and music content by utilising **Spotify's** massive library and expansive library to offer high-quality and reliable audio streaming. I also wanted to provide another alternative method for social interaction as opposed to traditional online solutions, which are limited and oversaturated. I also wanted to provide users with more control and convenience when planning and hosting games with minimal setup and unrestricted playing, which would be played and operated over a socket connection (**TCP/IP**) created by the user who hosts the game. I also wanted to offer much more interactivity within the application as well as a superficial scoring system to add a competitive nature to the games. Another goal I set out to achieve was to accommodate the restrictions of social distancing attributed to the COVID-19 pandemic. While also defining the fundamental goals of the project, there were numerous features I set out to offer users as an extension to the core functionalities of the application.

Users of the application are capable of

- Users can register and log in via their Spotify Account.
- Users can host games.
- Hosts can specify the genre and subgenre of the games and select from
- Users can join a session being hosted by another user.
- Users can interact with various in-game features:
 - Message opponents using the chat feature.
 - React to songs being played.
 - Examine the in-game leader board.
 - View songs already played.
 - Validate Card with the "Bingo!" button.
- View details about their profile, score and lines achieved.
- View their playing history.
- Read a tutorial within the application explaining how the rules.
- Explore the most up to date catalogue of songs within a playlist.
- Examine statistical insights about games played.
- Explore and sort the player leader board:
 - By Score
 - By lines achieved
- Purchase expanded playlist packages with alternative subgenres.

Business Case

Music Bingo has demonstrated its potential to be a revenue-generating and revenue enabling solution. What it was generally missing beforehand was a monetisation model which capitalised on its popularity. Traditional bingo events required a pay-in to receive a card and be able to play, although my experience of music bingo was that it encouraged sales within public houses and nightclubs when customers would buy drinks or food as a side effect while playing the game. This model is a potential application for the solution, although it is not feasible in the current pandemic climate.

Players have access to the mainstream playlists of the six primary genres, which are Rock, Pop, Electronic, Country, Rap and Festive. Each genre has a set of expanded sub-genres associated with them, which are genres used in-game. The application contains a paywall that block's the expanded sub-genre playlists such as Alternative rock (Rock) or Techno (Electronic). The expanded genres come in packages that can be purchased through **Googles In-App billing API** who have a provider/Developer revenue ratio of 30/70 accordingly. Each playlist package costs €2 and is a once-off payment that authenticates the user to be able to use those playlists when hosting games. Players who join other sessions are not required to own the playlist package to join a game. This makes purchasing packages more attractive as the cost can be split amongst all players, and each player can benefit. Adding new playlists and genres can be done with ease, so the revenue model could pivot to being subscription-based under a premium model, which would entitle users to full access to all playlists and extended features. The application could also form the basis of a solution that could be tailored to an event management company who could utilise it as a service within their company, paying fees for use or, again, possibly offering a subscription to their service.

Another potential avenue of income associated with the application is the potential for betting. While betting can occur in person through cash, collaboration with a payment system could facilitate the transaction of funds between players into a collective pot which would be allocated to the winner following the end of the game. This could also be achieved possibly via Fiat currency or cryptocurrency. By facilitating the game as well as handling the transactions, the application would be justified in incurring a fee for the process creating another potential revenue stream.

Technology

A brief overview of the technology stack used to develop the application.

Kotlin



I decided to use kotlin to develop my application. The language was developed by Google who wish to push it as the flagship language for android development as it completely interoperable with Java.

This project was my first introduction to the language, and I really enjoyed the process of pushing myself to learn new concepts and syntax. I feel confident in my understanding of the language having learned it throughout the development of the application.

Android Studio



Android Studio is the IDE I used to develop the application. The built-in emulators and port forwarding capabilities using telnet were very useful when testing the multiplayer functionality. Android Studio also has native firebase capabilities and utilises Gradle for build automation allowing for easier dependency management.

Sockets (TCP/IP)



The TCP/IP protocol is the underlying technology powering the multiplayer capabilities of the application by facilitating end-to-end communication between clients and host. The host opens a connection on a specific port on the same network which the client players join and allow them to play wirelessly.

The sockets communicate via print writers and readers for simplicity using JSON objects for sending and receiving kotlin data classes. The socket connection is used for broadcasting song information, chat, reactions, and leader board updates.

Spotify API



My application utilises the Spotify SDK to authenticate users which grants them a bearer token allowing the device to communicate with Spotify's API. The API returns Song, Album, User and Playlist data in the form of JSON responses.

These objects contain album covers, mp3 samples as well as artist information used within the application.

Firebase



Firebase is a Google platform which offers software development tools. I utilised firebases Realtime-database which a NoSQL cloud database to store user information, game logs as well as playlist keys to retrieve playlist objects from the Spotify API. Firebase is also used to generate the insights and analytics for the application as well as for authenticating player permissions for playlist packages bought the play store.

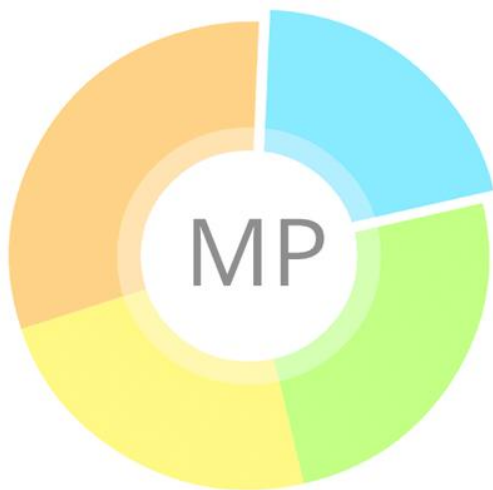
Google Play Services



Google Play services offer various tools, SDK's, and APIs for native app development in Android Studio. I utilised googles In-App Billing library (IAB) to position a pay wall for expanded subgenre packages as well as facilitating payments for said packages.

Playlist authorisation is implemented in collaboration with firebase as when a user purchases a new playlist, the permission is amended in their user account.

MPAndroidChart



MPAndroidChart is an open-source library used for developing charts and data visualisation views in android applications. This library was very convenient when developing the analytical insights based on historical game data.

There is wide variety of visualisations all of which are highly customisable and interactive, and the library was convenient to use. The library was used with firebase to pull game logs from the database.

Supporting Technologies

GSON



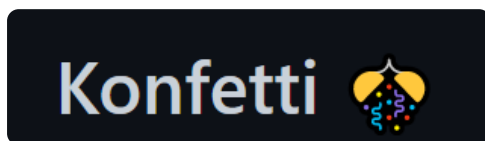
GSON is Google library used to parse, generate and map Kotlin/Java objects to JavaScript Object Notation (JSON). This was particularly useful for mapping Song and playlist objects to JSON to send over print writers rather than using object streams.

Picasso



Picasso is a library used for image handling library for android which is useful for resource loading which I needed for retrieving album covers from the Spotify API. The library also offers tools to transform and adapt images.

Konfetti



Konfetti is a lightweight confetti particle system which I use for victory dialogs within the game. The graphics are also highly customisable allowing me to adapt the confetti to my application colour scheme.

Kotlin Coroutines



The Kotlin coroutine library offers asynchronous programming and non-blocking routines in kotlin which was very useful when making HTTP requests to Spotify as well as delegating my client handlers into threads.

Git / GitHub



I used Git as a means of version control, the repository for the project was submitted to github where my commits reflected major changes and new features. Git was useful for logging the progress of development as well as rolling back any undesired major changes.

Requirements Analysis

Functional Requirements

The functional requirements outline the essential features for the system to perform as outline in the project objective. Each feature serves a significant purpose, and some features are dependent on others. The requirements outlined follow a generic template of mobile application requirement analyse, as well as an example case study of a requirement analysis of an android application tailored to fishermen.

Feature	Requirement	Purpose
User Register	Add user to the database and create an account	Allows the user to access the application and to be authenticated with Spotify.
User Login	Authenticate user with Spotify	Grants the application a bearer token needed to interact with API
Create Game	Allow a user to assume the role of a host and create a session	A session must be created for other players to join and play, which also allows the host to define the genre.
Join Game	Allow a user to attempt to join a session	Allows a user to access hosted session and for the game to begin.
Validate Card	Validate a user's Card to examine whether lines have been achieved	Necessary for the game to progress, players can claim lines with the bingo function, allowing them to claim lines and potentially win
Text Chat	Facilitate text chat between players in-game	Allows players to communicate and respond to events in-game.
Song Reaction	Enable users to react to the currently playing song	Encourages engagement and interactivity.

Feature	Requirement	Purpose
In-Game Leader board	Allow users to examine the current standings within a game	Users can examine their current standing compared to their opponents and check the next objective
View Played Songs	Display the songs already played so far	Reminds users of what songs have been played in case one is missed in-game
Browse Catalogue	Display an interface of all available playlists and their content	Users can examine the most up to date playlists as well as browse playlists for potential purchase.
Generate Insights	Provide a dashboard of data visualisations and insights regarding information about user activity and games	Provides an insight into the most popular genres, sub-genres as well as a global leader board of users based on score or lines
User Profile	Allows users to examine their profile and details	Displays user information, including location, country, score and lines, as well as a list of previously played games.
Game Tutorial	Reveals a helpful guide explaining the rules of the game	Ensures new or unfamiliar players are accustomed to how to play the game
Genre Store	Facilitates the purchase of subgenre packages	Allows users to buy expanded subgenre packages that can be used when hosting games and for authorising the use of subgenre packages.

Non-Functional Requirements

Usability

An existing Spotify account is required to access the application. Spotify premium subscription is not mandatory. A tutorial is provided within the application to explain the process of hosting and join as well as the rules of the game. Access to an internet connection is required to utilise the application.

Legal Requirements

All use of licensed audio regarding royalties and permissions as well as handling of user data is covered under Spotify's terms of service as well as their branding guidelines.

Reliability

The reliability of the system depends on the availability of both firebase and Spotify. Google claim in their service level agreement (SLA), they will make reasonable efforts to ensure an uptime of 99.95%. Spotify does not retain an SLA regarding their platform and reserve the right to perform maintenance and enforce downtime as they wish.

Performance

The system's performance relies heavily on the strength of the user's network as there are certain activities and features which require Spotify or Firebase queries, but there are dialogues to reflect the wait time. Hosting of games is seamless as the setup and parsing of playlist objects will happen asynchronously. There are a total of 5 return communications between a client and host occurring around 2-3 times an iteration amounting to potentially 10-15 transmissions a second over a TCP connection allowing for instantaneous messaging, reacting and validation.

Compatibility

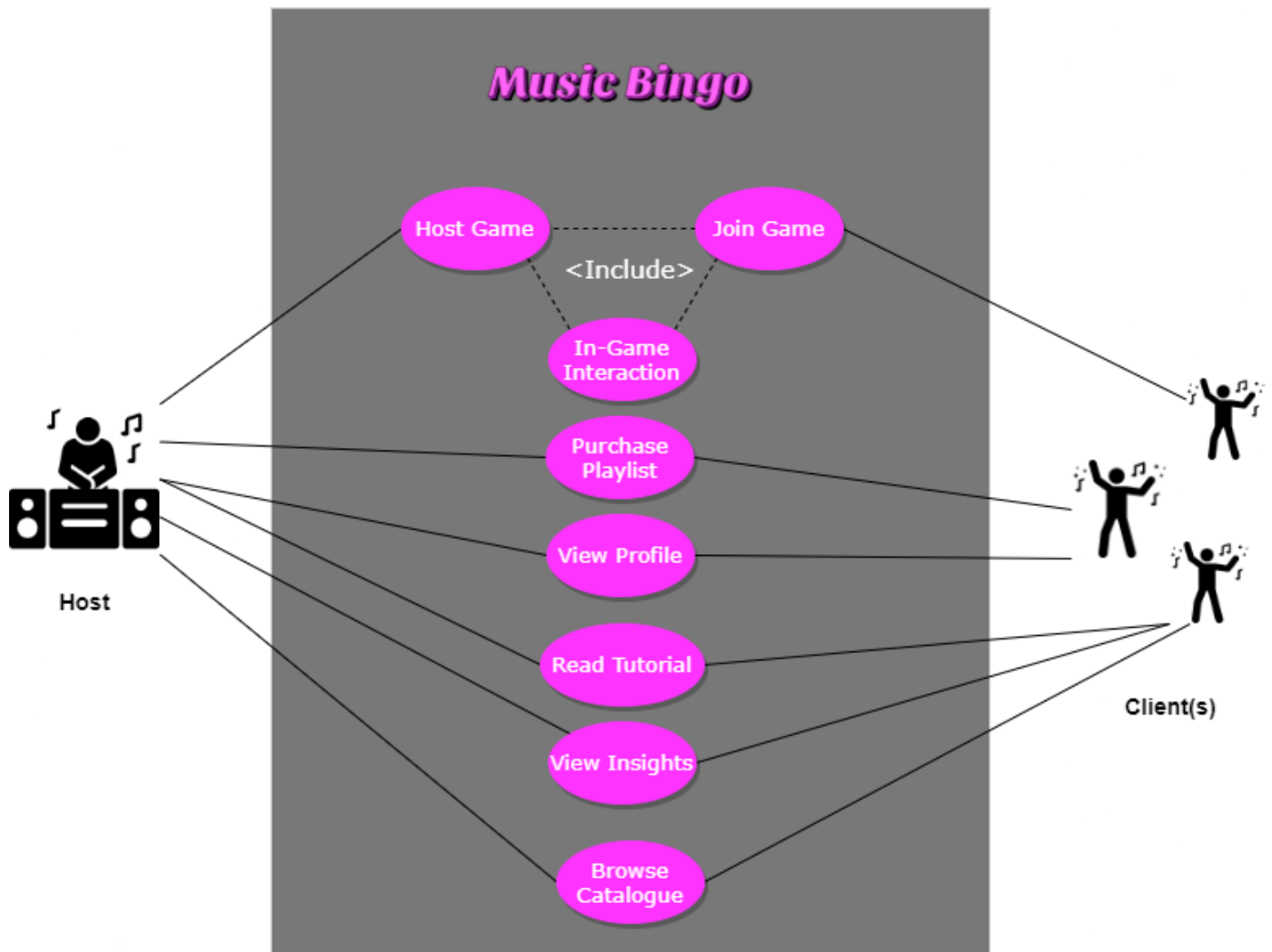
Available across all mobile android devices supporting Android operating system Lollipop 5.0(API 21) or above

Scalability

Firebase is prepared to scale linearly, while Spotify's API is equipped to handle massive amounts of queries meaning the system is highly scalable.

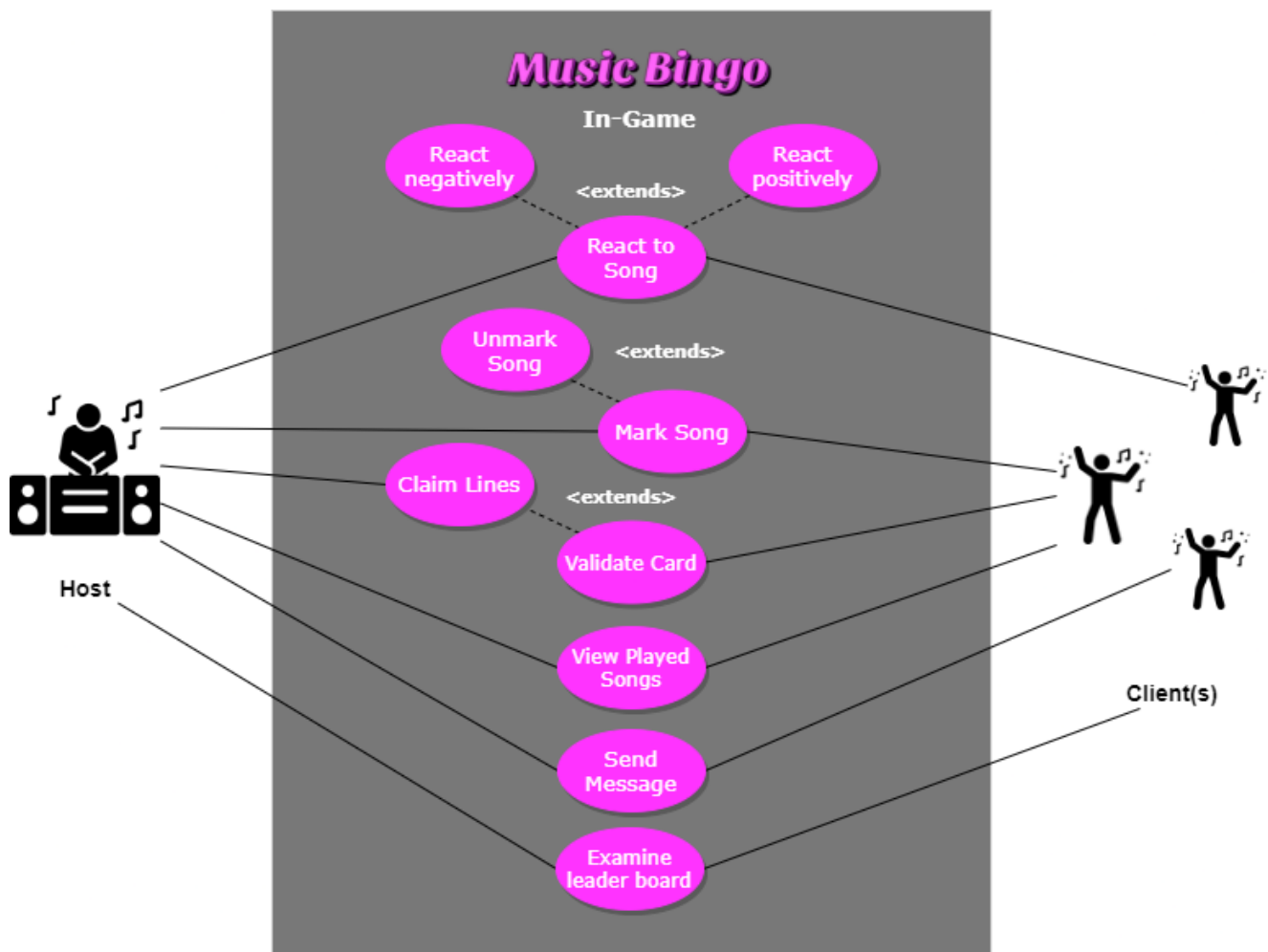
High-Level Use-Case

This use-case highlights the actors and interactions throughout the system, including the extraneous features of the application.



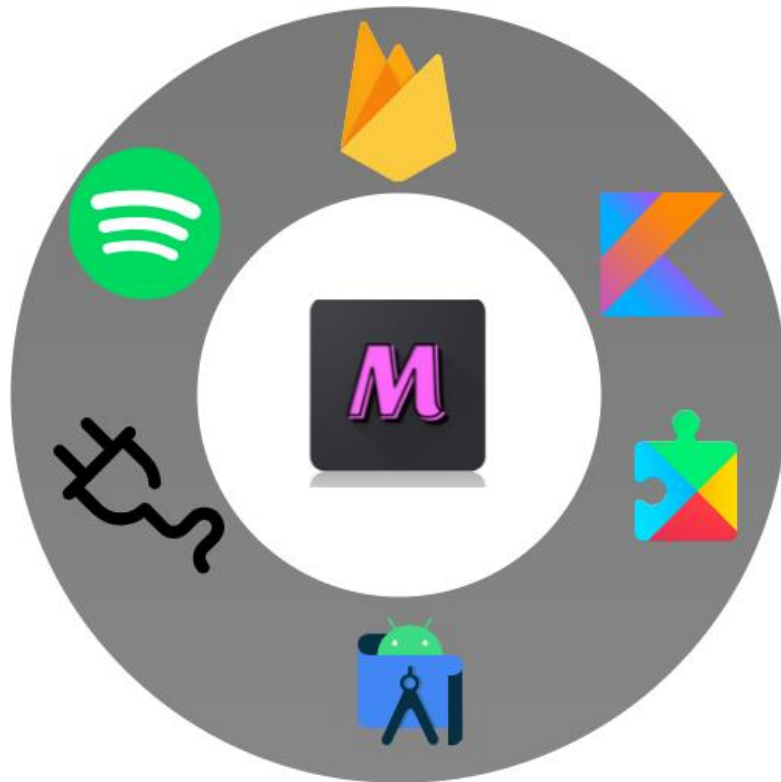
In-Game Use Case

This use-case demonstrates the actor interaction with features within the context of a session. Both host and clients can interact with the same features, the only difference being that one player must assume the role of host in order to create the session.

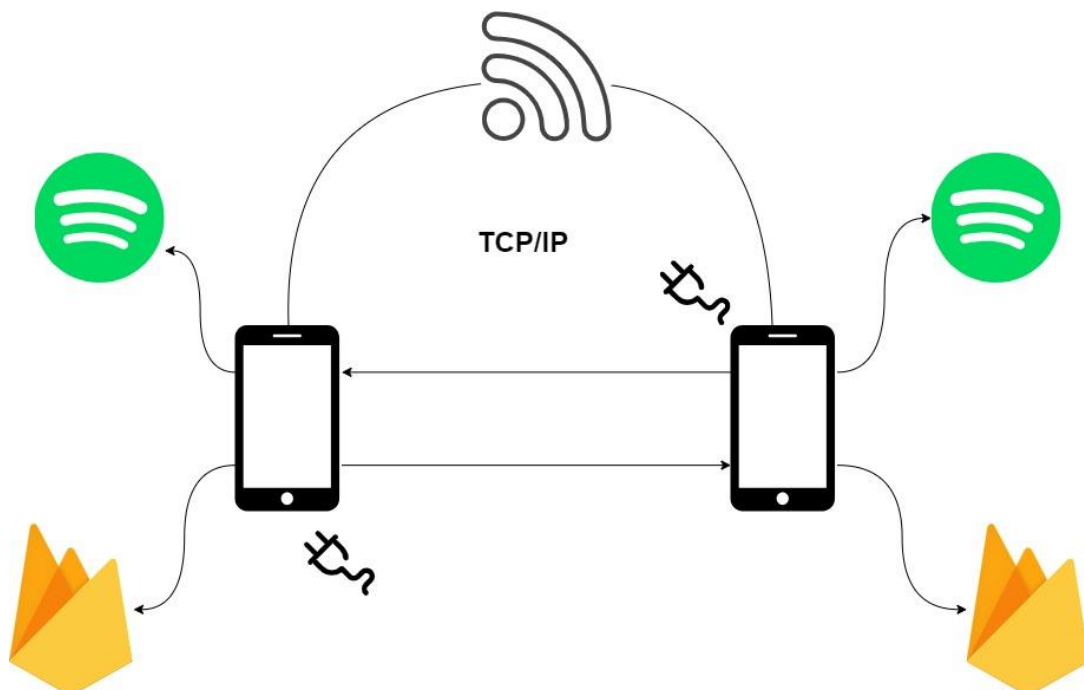


System Design

Core technology stack



In-game functionality stack detailing the TCP functionality. The playlist object is received from the Spotify API, and game details are logged to firebase.



Technologies

Kotlin

Given that this project was my first introduction to the Kotlin programming language, and it made the initial development process very slow and arduous, but in time I came to really enjoy the language and its features. The 'lateinit' variable declarations, coroutines and built-in functions proved very useful throughout the creation of the app, and I would never turn back to developing android applications in Java. I feel a sense of achievement in having engaged with new technology and building proficiency and understanding in it.

Spotify

I had decided to use the Spotify API to remove the need for accumulating and processing hundreds of MP3 files to be used in the app, as well as avoiding any ethical or legal barrier associated with utilising licensed music in this way. Spotify provided a vast library of tools, including an authentication handler which I used for my login and registration. There was also a remote player who would appear in a user's notification drawer. Spotify provides very comprehensive documentation, which unfortunately has no Kotlin alternatives, but the translation was not too difficult. Early in the development process, I had decided to avoid the remote player as it would allow players to control the audio player. Instead, I chose to use the Android media player, which can stream the mp3 song sample directly via the URL received from the Spotify API.

TCP/IP

The TCP/IP protocol is the underlying technology I decided to use to implement the multiplayer functionality within the application. Kotlin has access to all of Java's native libraries, and my understanding of the fundamental concepts of Sockets was essential to building the appropriate streams as well as validating inputs and formatting outputs. Building the necessary channels for communication and tying in it with the UI and flow of the game was the most difficult aspect of the project. To test this functionality, I needed to implement port forwarding within the emulator to pair the devices as it contains its own address. Running the two emulators is also very demanding on my machine, and testing proved to be very slow and tedious. The functionality of the app is heavily reliant on constant communication between the sockets facilitating the Chat feature, reactions, game updates and validating of cards. The slow development of the Sockets was worthwhile. I feel as the connections are fast and frequent, offering quick updates and reflections in-app.

Google In-App Billing

Google's play services API offered a comprehensive library that facilitated in-app payments and allowed me to enforce a pay wall on extended content in my app in the form of playlist keys for specific sub-genre packages and adding a monetisation element to the project. The API was well documented and easy to use, but the implementation was slow as to avail of google play services. The application was required to be submitted to the developer console, where it underwent investigation to ensure the application complied with ethical and content guidelines. This process took around 3-4 weeks. Once the application was approved, I was able to create products on the developer console and define the price and attributes surrounding them. Once all the products were defined, they could be retrieved as Stock keeping unit (SKU) objects from the API, which would be given as parameters to the billing client which handled the purchasing process. Once complete, the user's profile was updated in the database to reflect that they now had access to the product they had purchased.

Firebase

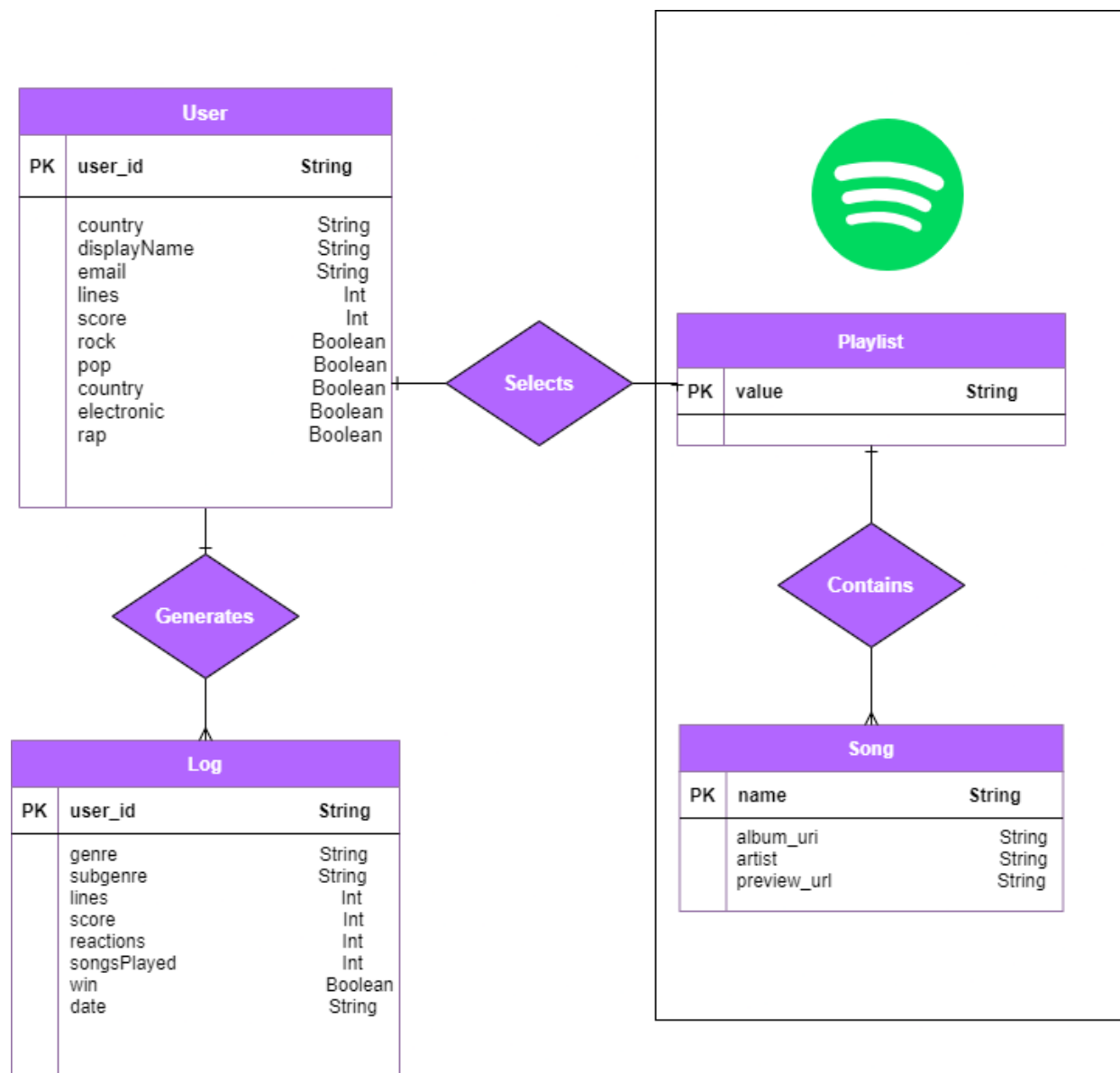
Firebase's real-time database was the immediate choice for this project as I felt comfortable writing queries and manipulating data on the platform. The reliability and convenience of the cloud allows all instances of the application to make seamless and reliable connections to firebase, removing the need to host a backend. The JSON structure of the database also eliminates the need to create a static schema, and referential integrity is much more flexible. Firebase is used to store User information, playlist keys needed to retrieve objects from Spotify, as well as historical data related to the games and their resulting scores, played songs and genres. Firebase also proved very in collaboration with Spotify, the user was authenticated by Spotify, and in turn, their details are stored on firebase. Firebase was essential in controlling the authorisation of players to use specific playlist packages.

MPAndroidChart

MPAndroidChart adds chart and data visualisation capabilities to android applications. I included an insights presentation within the application which displays observations based on the most up to date information within firebase. Each game generates a historical log which I used as a metric to generate pie charts and distributions based on genre and subgenres played, as well as calculating score and line averages.

Database Structure

Firebase real-time database is a NoSQL database meaning it does not enforce a relational schema and does not use tabular entities. While there are fewer restrictions and dependencies when managing firebase objects, it is essential to implement an artificial framework to manage the interactions between objects and for indexing purposes. The entities are represented as Kotlin Data classes before being converted to JSON as they are persisted in the database.



User

The primary descriptive attributes of a user, which are the ID, name and country, are received through a user-based query to the Spotify API. Users are created with a line and score value of 0, which is accumulated through gameplay. The genre entitled Booleans each indicated a respective authorisation for a particular user to access the monetised extended playlists.

user_id	Spotify User ID
displayName	Username associated with Spotify account
email	E-mail associated with Spotify account
lines	The total number of lines achieved in-game
score	The total score accumulated in-game
rock	Rock sub-genre package authorisation
pop	Pop sub-genre package authorisation
country	Country sub-genre package authorisation
rap	Rap sub-genre package authorisation
electronic	Electronic sub-genre package authorisation

Playlist

To access the details of a specific playlist in the Spotify API, the request needs a playlist URI parameter. All the playlist URI codes are stored in the database, and the necessary code is retrieved when a game is created. The URI's are stored in a key/value format, the name of the subgenre being the key and the URI being the value.

playlist_uri	Code needed to retrieve Spotify Playlist object
---------------------	---

Song

The API returns the playlist object in the form of a JSON response. The playlist contains a JSON Array of Song objects with various attributes and nested objects. For the purpose of my application, I mapped only the song name, album cover URL, MP3 preview URL and artist to a Kotlin Data class. Each song object is added to Array List, and this list is used in numerous different in-game operations such as generating cards, printing cards, passing songs over I/O streams and validating, displaying album covers and streaming audio.

user_id	User ID associated with this game
genre	The primary genre of the game
subgenre	The sub-genre of the game
lines	The lines achieved by the player in this game

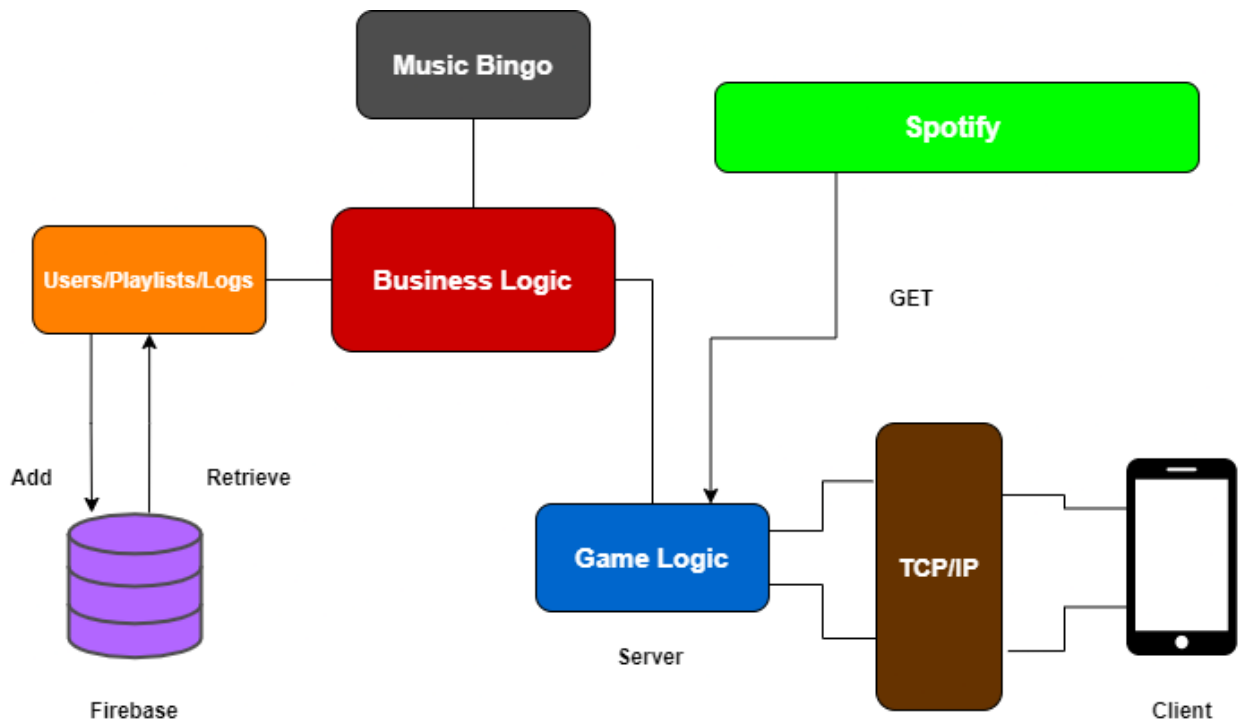
Log

After a game wraps up, the details of the session are persisted as a game log. The details of these games are used for insights as well as an activity log for individual users. Attributes such as genre and subgenre are used to create distributions to measure the behaviour and preferences of players.

user_id	User ID associated with this game
genre	The primary genre of the game
subgenre	The sub-genre of the game
lines	The lines achieved by the player in this game
score	The score achieved by the player in this game
reactions	The number of reactions the player made
songsPlayed	The number of songs played during the game
win	Indicates whether the player won the game
date	The date on which the game occurred

Software Design

A brief overview of the system architecture.



User Interface Design

UI Colour Scheme



- Rounded buttons, text fields or layouts are from open-source XML resources.
- All dialogues are custom-designed following the application colour scheme.
- An open-source GIF was utilised for the loading dialogue.
- An open-source video was utilised as the landing page for the application.
- The home-screen backdrop is also an open-source image.
- All icons and SVG's were sourced online.
- The Logo was created using a font generator.
- The card layout and button gradients were also custom made.

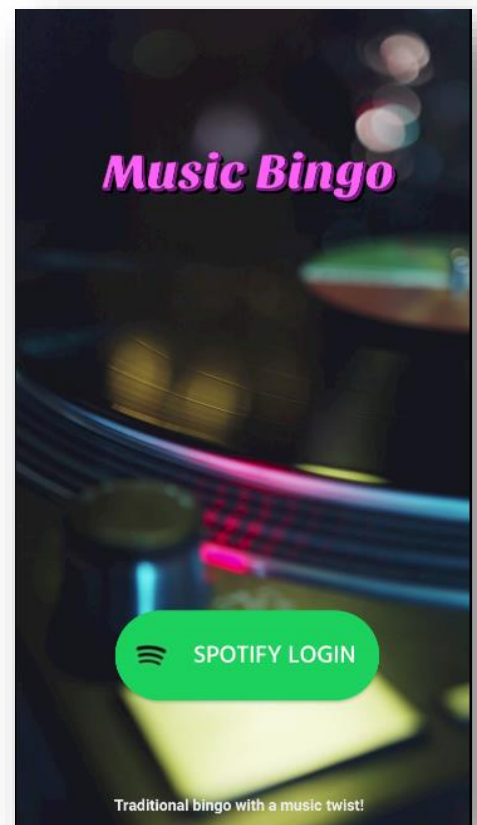
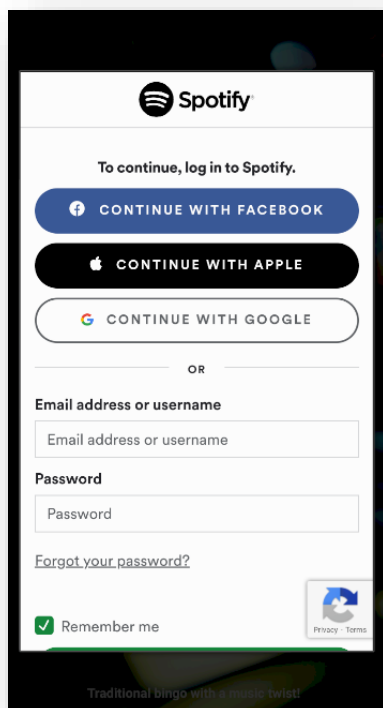
Implementation

In this section, I will highlight the features and fundamental aspects of the system as well as the direction and decisions made when implementing these features.

Login and Registration

MainActivity.kt

This is the opening activity of the application, the background of the activity is a repeating sequence of a spinning vinyl record. To enter the platform, the user must prompt the Spotify Authentication Client and define a scope of permissions. The permissions outline the various types of data that the user is granting the application access to such as user data, personal activity, playlist information.

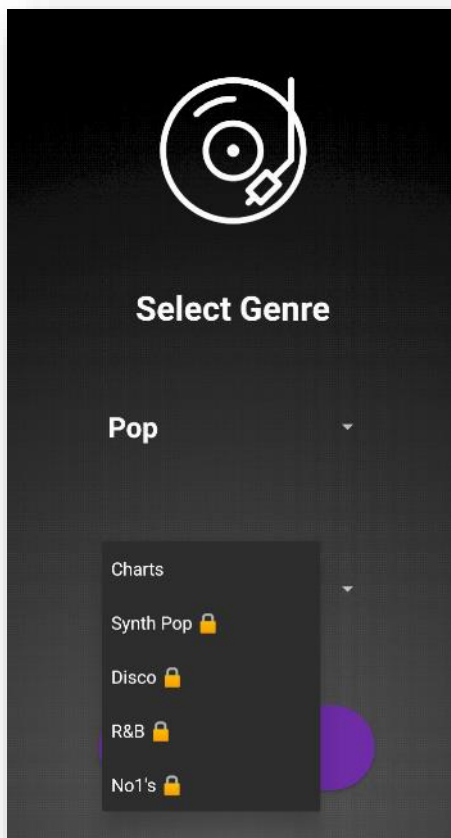
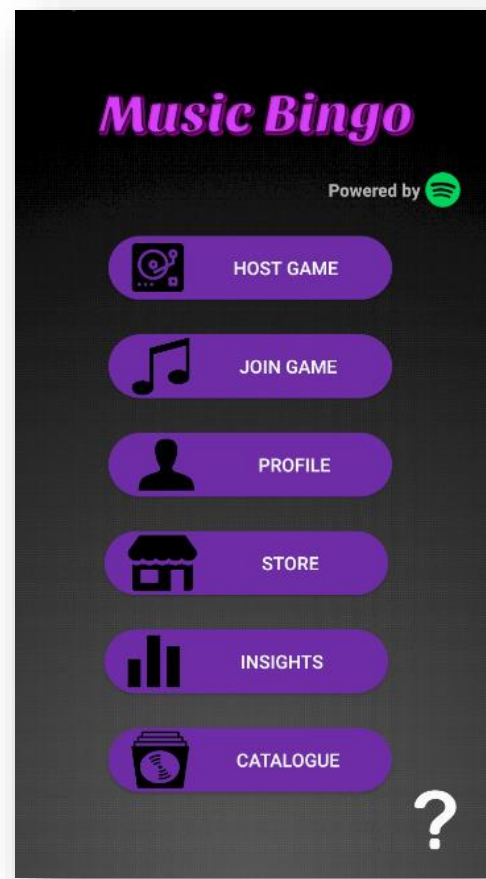


The benefit of the Spotify authentication client is that it offers a wide variety of login options. Once a user is logged in to the application their details are remembered, and it removes the need to login again each time they launch the app. Once the login is authenticated, the user's presence is examined in the database and if not detected the user is persisted. The application receives a bearer token from the client's response which is stored in **sharedPreferences** for convenience.

Homepage

OptionsActivity.kt

This activity is the directory to all other features within the application, the user can access their profile, buy playlists in the store, host and join games and examine the application insights. The question mark in the bottom right corner brings the player to the app tutorial explaining the rules and in-game functionality.



Hosting

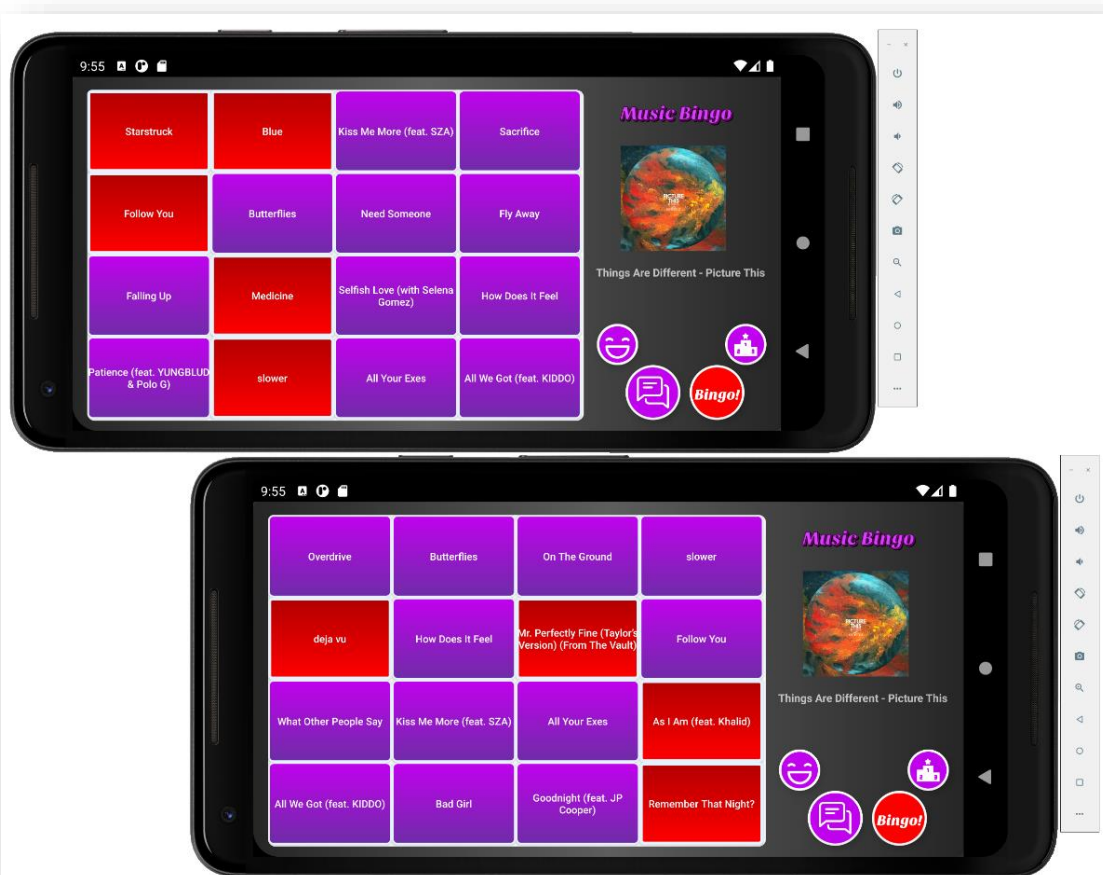
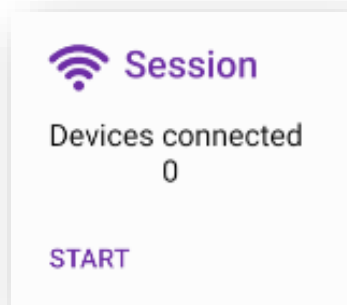
HostActivity.kt

To create a session a player must assume the role of host, select a genre, and create a game, this user opens a connection with a server socket listening for clients. Notice that the extended subgenres are locked as this user has not purchased the package yet. When the genre is selected firebase is queried to retrieve the playlist URI which is then passed to Spotify to retrieve the playlist object.

Gameplay

Game.kt / GameClient.kt

The host waits and listens for clients to join the session. The game cannot begin unless there is at least one player connected. The host can hand up to a maximum of 3 clients. When the game commences, players must aim to match four songs vertically to claim a line, after that two lines of the objective and finally the whole Card. The host user controls the creation of cards using a self-implemented algorithm which adds wild cards in the form of imitation tracks and slices and shuffles the playlist based on the size and sends it to each client.



Connection

Once a client requests service from the server socket, this function delegates a thread to handle the inbound and outbound communication while also tracking the number of players in the session.

```
private fun handleClient() {
    GlobalScope.launch(Dispatchers.IO) { this: CoroutineScope
        users = ArrayList()
        server = ServerSocket( port: 6000)
        while(true)
        {
            if(user_count != 3)
            {
                try{
                    val sock = server.accept()
                    thread(start = true) { connect(sock) }
                    users.add(sock)
                    user_count ++
                    card.players = user_count
                    runOnUiThread { session.setMessage("Devices connected\n
                                                                $user_count") }
                }catch (e : SocketException){}
```

The connect function defines the input and output streams of the client's socket. The client is notified of the genre and subgenre of the game and given a card to print. Once the connection is established, the host can start the game triggering the first song to play. The function is very long and contains more channels and sections for validation, it is the core of the gameplay, and all transmissions pass through this function, dictating texts, reactions, card validation, playing songs and updates to the standings. All these communications are mirrored on the client-side with slight differences.

```
private fun connect(client: Socket)
{
    GlobalScope.launch(Dispatchers.IO) { this: CoroutineScope
        try {
            val output = PrintWriter(client.getOutputStream(), autoFlush: true)
            val input = BufferedReader(InputStreamReader(client.inputStream))
            val jsong = Gson().toJson(card.generateCard(songs, sub_genre))

            output.println("$genre-$sub_genre")
            output.println(jsong.toString())

            var status = "STATUS_NEUTRAL"
            var change = "NO_CHANGE"

            while(true)
            {
                if(next and start)
                {
                    next = false // Resets the next boolean so another song will be played
                    if(song_index < songs.size)
                    {
                        val n_song = Gson().toJson(songs[song_index])
                        status = "SONG$n_song"
                        playsong(songs[song_index])
                    }
                }
            }
        }
    }
}
```

Playing Songs

The song index and next Boolean handles the timing and changing of songs. This playsong function receives a song object where it prints the title, loads the image using Picasso and streams the song using the preview URL.

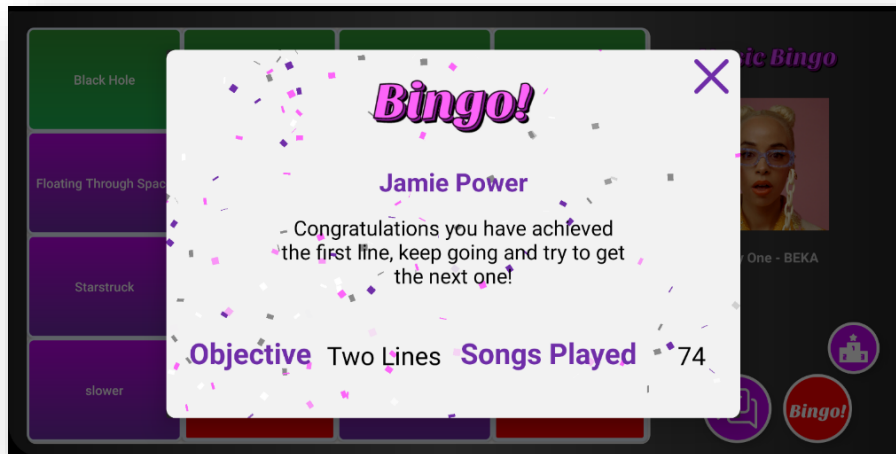
```
private fun playsong(song: Song) {  
  
    played_songs.add(song.name)  
    now_playing.text = "${song.name} - ${song.artists}"  
    react = ""  
    runOnUiThread { react_button.count = 0 }  
    val image = song.album_url  
  
    GlobalScope.launch { this: CoroutineScope  
        this@Game.runOnUiThread {  
            Picasso.get()  
                .load(image)  
                .into(album_cover) }  
  
        reactions.clear()  
        mediaPlayer = MediaPlayer().apply { this: MediaPlayer  
            setAudioAttributes(  
                AudioAttributes.Builder()  
                    .setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)  
                    .setUsage(AudioAttributes.USAGE_MEDIA)  
                    .build())  
            setDataSource(song.preview_url)  
            prepare() // might take long(for buffering)  
            start()  
            delay( timeMillis: 30000)  
            song_index ++  
            next = true  
        }  
    }
```

Once the song is complete, the next Boolean resets, the index moves to the next song, and this function is called.



Claiming Lines

Once a player believes they have matched a line, they will click the bingo button to validate their Card. Songs are added to a list as they are played on both the server and client-side. This list is used for validation to indicate to a user whether they have indeed claimed a line. The line is turned green to demonstrate that its been claimed, and the buttons are locked.



When a user clicks the bingo button, the Card calls a function called ValidateCard. This function is a long cascading conditional verifying each possible scenario where lines can be called based on whether the first line or two lines have been achieved. When a conditional is satisfied, the function returns the response, it plays a jingle to indicate a victor and locks the songs achieved turns the buttons green.

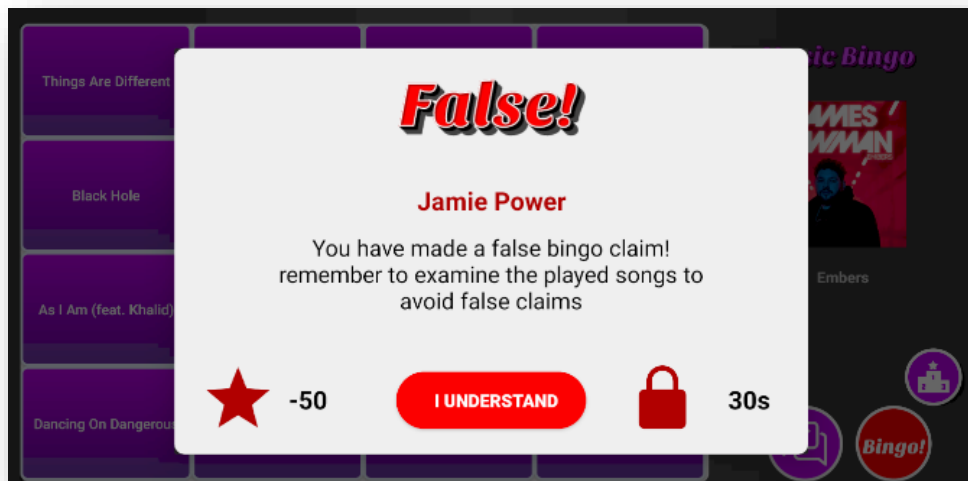
```
fun validateCard(played_songs : ArrayList<String>) : String
{
    var response = "None"

    if (!ONE_LINE) {
        if (played_songs.contains(song1.text) && played_songs.contains(song2.text)
            && played_songs.contains(song3.text) && played_songs.contains(song4.text)
            && (song1_click && song2_click && song3_click && song4_click) ) {

            ONE_LINE = true
            response = "ONE"
            song1.background = activity.resources.getDrawable(R.drawable.green_border_button, activity.resources.newTheme())
            song2.background = activity.resources.getDrawable(R.drawable.green_border_button, activity.resources.newTheme())
            song3.background = activity.resources.getDrawable(R.drawable.green_border_button, activity.resources.newTheme())
            song4.background = activity.resources.getDrawable(R.drawable.green_border_button, activity.resources.newTheme())
            song1.lock = true
            song2.lock = true
            song3.lock = true
            song4.lock = true
            song1_click = true
            song2_click = true
            song3_click = true
            song4_click = true
            val mediaPlayer = MediaPlayer.create(
                context,
                R.raw.success_sound)
            mediaPlayer.start()
        }
    }
}
```

False Claims

Traditionally in bingo, making a false claim is seen as either embarrassing or an annoyance to other players, and it should not be without consequence. The same goes for music bingo. Making a false claim will cause a player to lose 50 points and have their bingo button locked for 30 seconds as a result.



The **validateCard()** function is called in a switch statement from either the host or client. We see that when it receives a "None" response indicating a false call. We see in this instance a vinyl scratch sound effects plays in the background, a counter is reset, and a thread is activated counting down from 30, preventing the player from making further validations in the elapsed time.

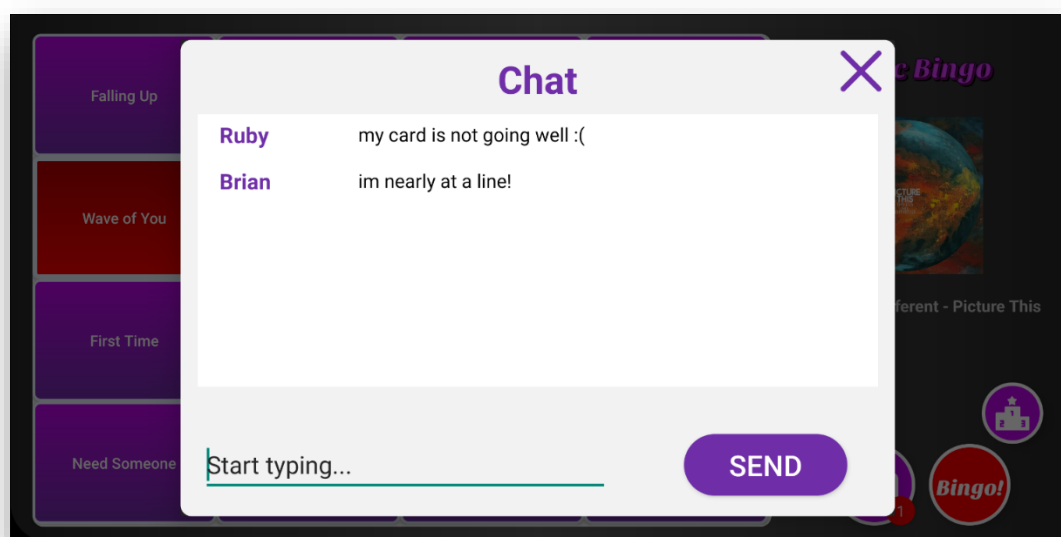
```
"None" -> {
    val mediaPlayer = MediaPlayer.create(
        context: this,
        R.raw.vinyscratch)
    mediaPlayer.start()
    lockCounter = 30
    val dialog = Dialog( context: this)
    dialog.setContentView(R.layout.false_dialog)
    dialog.findViewById<TextView>(R.id.fail_name).text = username
    dialog.findViewById<Button>(R.id.fail_button).setOnClickListener { it: View!
        dialog.dismiss()
    }
    dialog.show()
    bingoLock = true
    thread {
        while(lockCounter != 0)
        {
            lockCounter --
            if (lockCounter == 1)
                bingoLock = false

            Thread.sleep( millis: 1000)
        }
    }
}
```


Text Chat

The chat functionality is also self-implemented. I had decided early in the development of the app to use a **PrintWriter** output and **BufferedReader** input for the sake of simplicity. Any objects which were required to be sent over the streams were transformed to JSON. The chat works using an Array List of message objects to monitor the last message submitted and a list for the most up to the version of the chat log.

A message object contains a username, content, and date in the form of a long data type. Each player is constantly sending and receiving the latest message that was sent. When a user receives a message not in their log, it is added to the list, and the change is reflected in real-time.



The messages are adapted to a recyclerView which is contained inside a custom dialogue. The chat button is an open-source Android library that has a counter notifying users of new or unread messages similar to app icon notifications seen in other mobile applications



The chat function has its own input and output streams dedicated to handling message communications. Each iteration, the server sends each client its latest message and receives each client's latest message and vice versa. We can see the messages are converted and mapped to and from JSON objects using the Gson library. When the chat box is displayed, messages are sorted in order of most recently sent.

```
var lastMessage = ""
if(chatDialog.myLog.isNotEmpty())
{
    lastMessage = Gson().toJson(chatDialog.myLog[chatDialog.myLog.size - 1])
}
output.println(lastMessage)

val messagesString = input.readLine()
if(messagesString != "")
{
    val message = Gson().fromJson(messagesString, Message::class.java)
    if (!chatDialog.messages.contains(message))
    {
        if(!chatBox.isShowing)
        {
            runOnUiThread { messageButton.increase() }
        }
        runOnUiThread {
            chatDialog.notifyChange()
        }
        chatDialog.messages.add(message)
    }
}
```

The chat dialogue has its own class and set of methods for initialisation, setting adapter and filtering by date. This object is used by both server and client activities.

```
fun build(context : Context) : Dialog
{
    dialog = Dialog(context)
    dialog.setCancelable(true)
    dialog setContentView(R.layout.chat_dialog)
    recycler = dialog.findViewById(R.id.played_songs_cycler)
    sendButton = dialog.findViewById(R.id.send_button)
    val closeButton = dialog.findViewById<ImageView>(R.id.win_dialog_close)
    val mLayoutManager: RecyclerView.LayoutManager = LinearLayoutManager(context.applicationContext)
    recycler.layoutManager = mLayoutManager
    closeButton.setOnClickListener { dialog.dismiss() }
    text = dialog.findViewById(R.id.message_box)
    val adapter = ChatAdapter(displayChat())
    setAdapter(adapter)
    return dialog
}

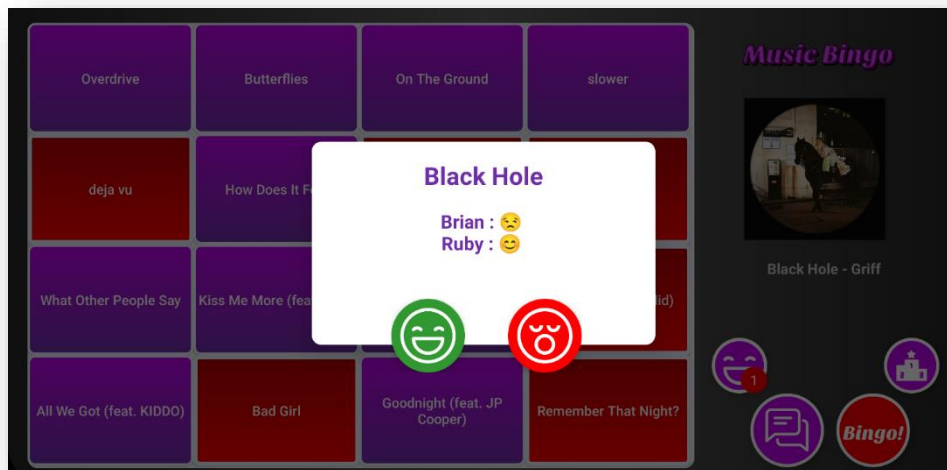
private fun setAdapter(adapter: ChatAdapter)
{
    recycler.adapter = adapter
}

private fun displayChat() : ArrayList<Message>
{
    messages.sortBy { it.time }
    return messages
}

fun notifyChange()
{
    if(this::recycler.isInitialized)
    {
        recycler.adapter?.notifyDataSetChanged()
    }
}
```

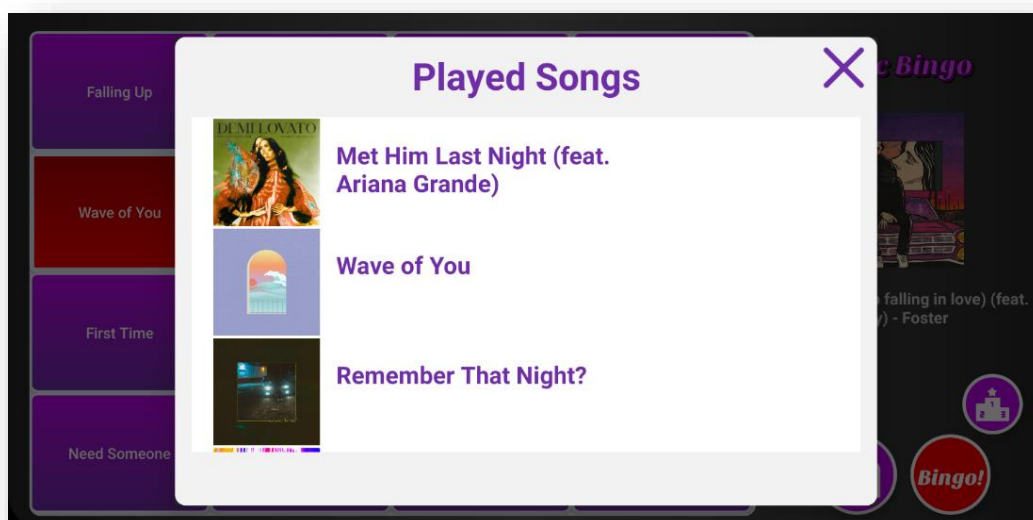
Reactions

The reaction feature adds another element of interactivity to the game allowing users to express their feelings to the currently playing song. Reactions also contribute a small amount to a player's score. Like the text chat, the feature is implemented using a hash map with the player being the key and reaction being the value. The reactions function also uses the counter button to notify players of new reactions. The reactions are reset after each song.



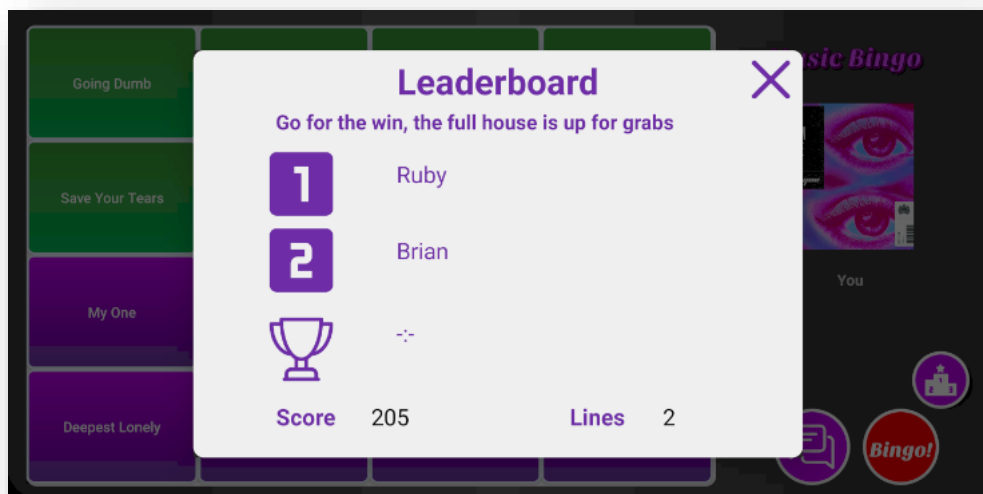
Played Songs

When the user clicks the album cover, it displays a recyclerview list of the already played songs as a reminder of what has already been played to avoid falsely claiming a line. As each song is played on both sides, it is added to this list.



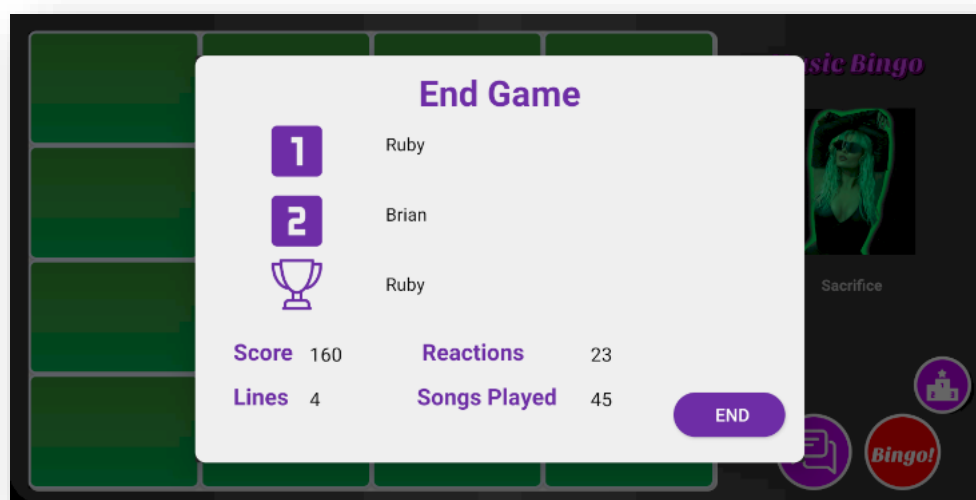
Leader board

The leader board allows users to examine the current standings of the game as well as examine their score and lines achieved. The leader board is also a convenient reminder of what the following line to play for is.



End Game

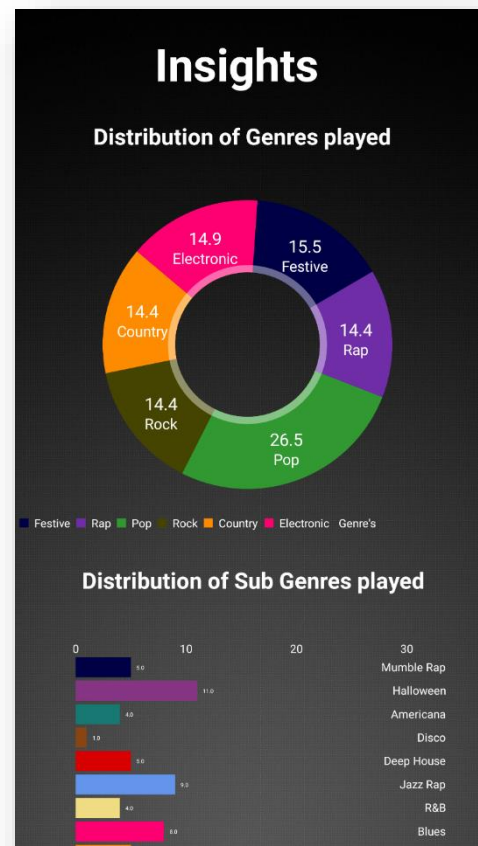
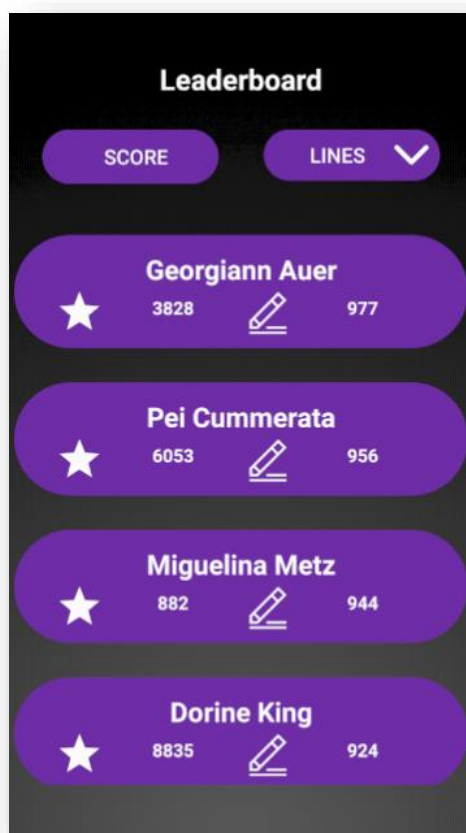
Once all lines are achieved by a player, the game ends, the player's profiles are updated to reflect scores and lines and the details about the game are logged. The user can only click the end button, which will return them to the home page, thus concluding the session.



App Insights

Insights.kt

The insights activity is a combination of firebase and MPAndroid Chart. The Insights feature a breakdown of genres which pulls all logs from the database and calculates the sum of each genre to generate a distribution as percentages. The graph below is a bar chart displaying a breakdown of the most popular subgenres based on the number of times they have been played.



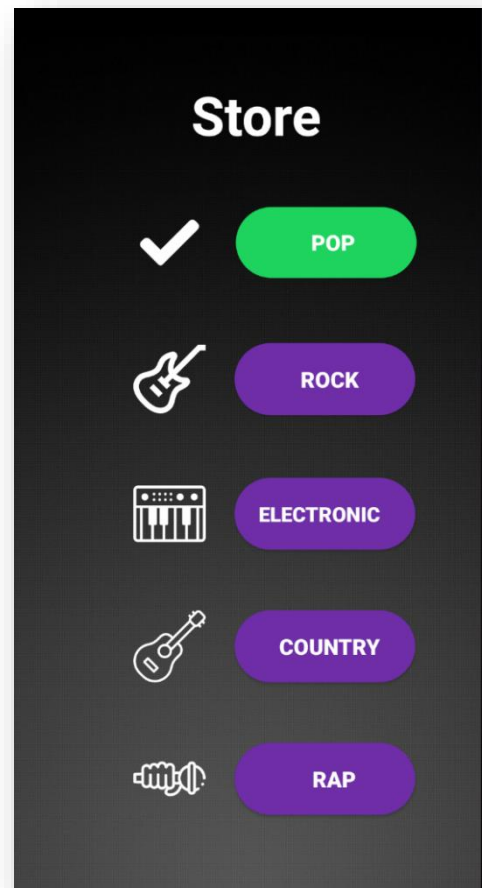
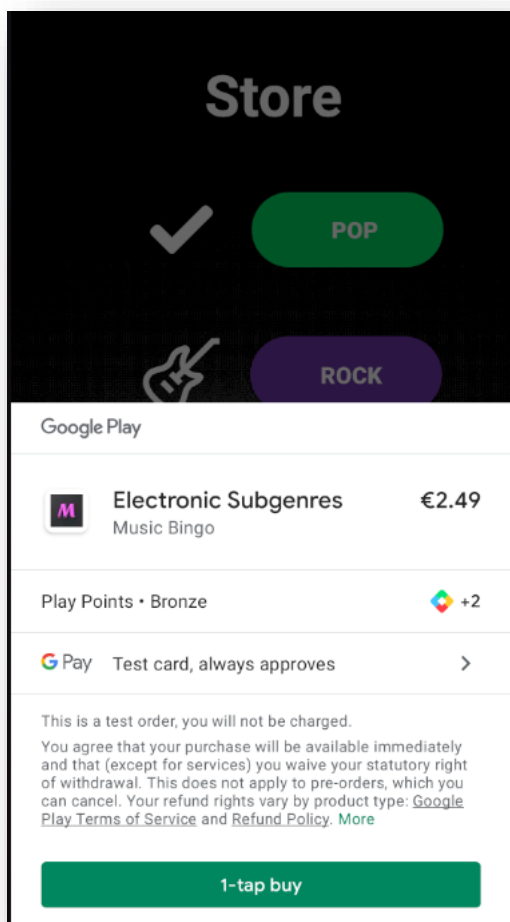
Below the graphics are series of indicators such as total score, total lines and average songs played per game. Further down in the insights activity is the leader board displaying the current standings of all players based on score and lines achieved. The leader board can be sorted in ascending and descending order using Kotlin Collections library to sort by attribute.

Playlist Store

Store.kt

The store activity is when a user can purchase the extended playlists by genre. The green button and tick icon indicates a package which has already been purchased and the purple

available to buy.

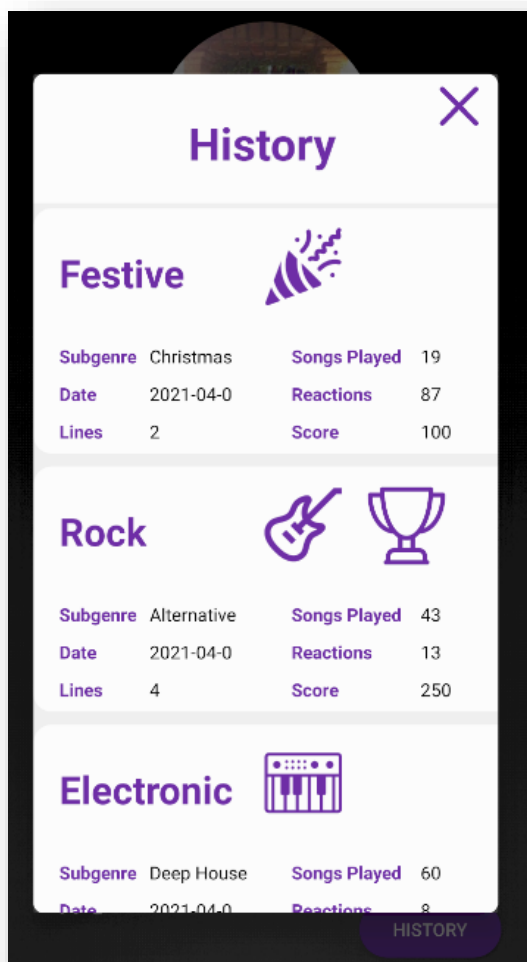
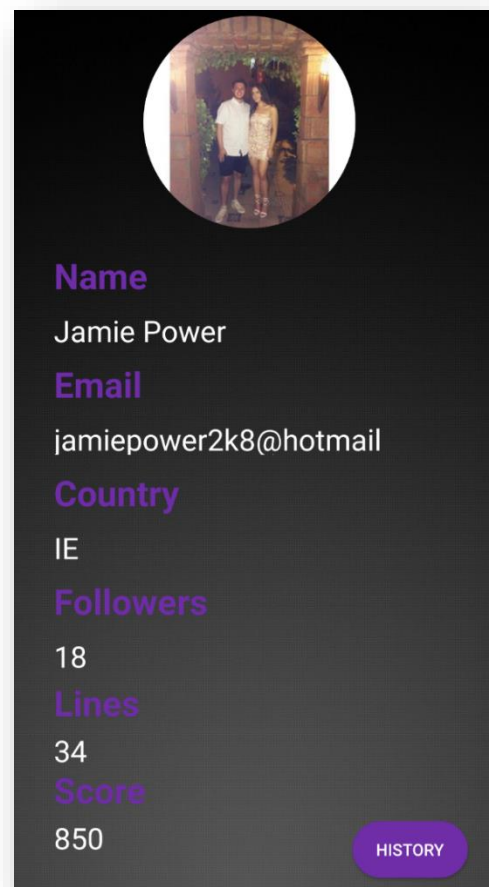


Clicking the desired genre opens the Google Play in app-billing client which outlines the product, application and price depending on the region the package is being purchased from, this client will handle things such as payment, verification as well as linking debit , credit or pay pal payment options available from the users google account.

Profile

Profile.kt

A user's profile is where they can view their details which are retrieved from Spotify such as the email linked to the account, their location as well as their total score and lines achieved through playing games.

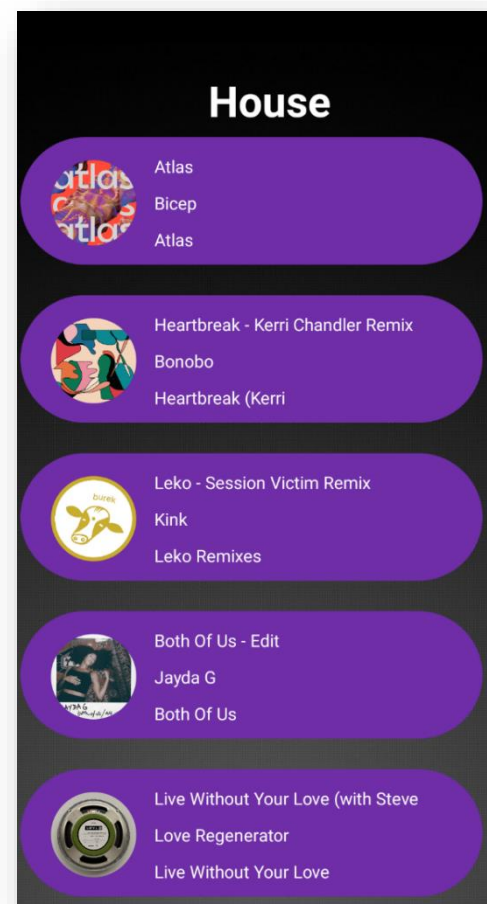
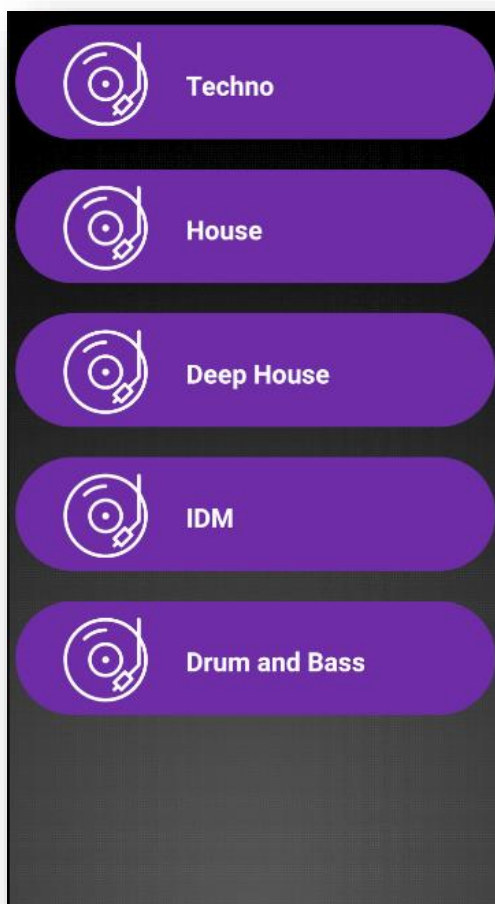
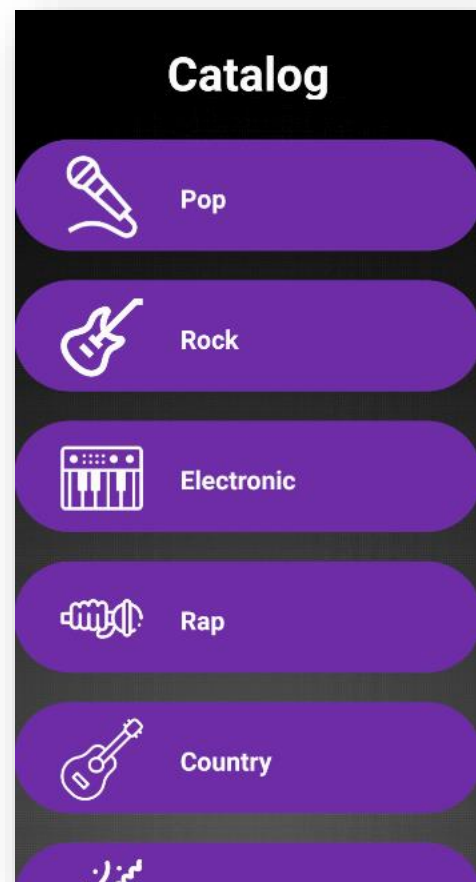


The history button opens a dialog displaying the users gaming activity. The list displays all the games a user has participated in as well as the genres, score, lines, number of reactions as well as the date the game occurred. The trophy icon indicates whether the player has won that game.

Music Catalog

Catalog.kt

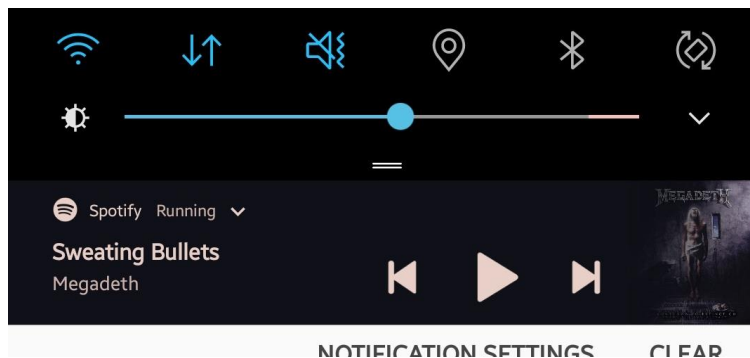
The catalog allows users to examine all of the available playlists and the songs within them. This feature serves to main purposes, the API is pulling real existing playlists from the Spotify platform which are subject to change and updates, so it allows players to view the latest tracks. It also allows users to browse the playlists so that they may examine the content before a possible purchase.



Issues and Resolutions

Spotify App remote library and Audio player

The core functionality of the application relied on seamless audio streaming and a wide variety of music to choose from. Spotify seemed like the obvious choice. The SDK offered a remote library allowing devices to connect and play an entire song with extra control of seeking and shuffling. I realised soon enough that streaming songs this way generated the audio player in the dropdown box.



This was highly problematic as it allowed all players complete control over the flow of the playlist, making the game easy to manipulate or break. I recalled from using the platform that there were songs demos that played for around 20 or 30 seconds and investigated whether I could use them. I discovered the song object contained a 30 second MP3 sample URL which adapted to the Android **MediaPlayer** object, which was also conveniently was the length of a song played during live music bingo games.

Port Forwarding and Telnet

The socket connectivity of the application was slow to come to fruition as developing a secure connection was difficult to achieve through the emulators. I initially failed to connect two devices from the same machine and explored numerous possible causes. After a long process of researching and investigating, I discovered it was the devices that were the problem. Each instance of the emulator runs behind a virtual router/firewall service that isolates it from your development machine's network interfaces and settings and from the internet. Emulator networking is scarcely documented before I discovered that the solution was to set up network redirections on the virtual router.

The server listens on port number 6000

```
server = ServerSocket(6000)
```

The client opens on port 5000 , on the devices gateway address

```
socket = Socket("10.0.2.2", 5000)
```

The device can be connected via Telnet which an application protocol on the Android Debug Bridge(ADB). The following commands allow the redirect

```
telnet localhost 5554
redir add tcp:5000:6000
```

By defining the redirect, I was able to connect the devices without any issues.

Test plan

Case	Expected	Actual	Pass/Fail
MainActivity.kt			
Register	User should be able to register an account through Spotify		
Login	User should have their login remembered and being to log in without authentication		
Game.kt / GameClient.kt			
Create Game	A user should be assuming the role of a host and create a game		
Join Game	A user should be able to look for a connection to join		
Claim a line	A user should be able to validate their Card		
Make a false call	A user should have their bingo function locked by making a false call		
Send a message	A user should be able to send a message visible by all other players		
Receive a message	Other users should be notified of a new message		
Submit a reaction	Allow a user to react to a song		
Receiving a reaction	Other users should be notified of a new reaction		

Case	Expected	Actual	Pass/Fail
Marking a song	When a user clicks a song, it should turn red, indicating its marked		
Locking buttons	lines that are claimed should turn green and be locked		
Examine leader board	A user should see the most up to date standings in the game		
Examine played songs	A user should be able to see an up-to-date list of played songs		
Dismiss notifications	When a reaction and text dialogues are opened, notifications are reset		
Notifying of wins	When a player claims a line, other players should be notified		
End Game	When a player finishes their Card, the game should conclude, and they should be brought to the home page		
Insights.kt			
Generate Insights	A user should be able to examine insights related to the app		
Interact with Graphs	The distributions should provide a small level of interactivity		
Sort Leader board	Sort leader board in ascending and		

	descending order by score and by lines achieved		
Case	Expected	Actual	Pass/Fail
Profile.kt			
Examine Profile	A user should be able to examine their profile		
View History	A viewer should be able to view their gaming activity		
Store.kt/OptionsActivity.kt			
Detect purchased items	When a user enters a store, the system should detect packages they have already bought		
Trigger a purchase flow	Clicking a product should prompt the billing client		
Handle purchase	Allow a user to purchase a product with multiple payment methods		
Reflect purchase	The store reflects the package that has been purchased and the playlists available when hosting a game		
Catalog.kt			
Select Genre	Display a list of subgenres based on genres		
Select Playlist	Display the songs within a specific playlist		

Summary

Music bingo is a mobile solution for a readapted bingo variant. The application is powered by Spotify, and It allows multiplayer connectivity over a local network allowing users to play together. The game is interactive, and the functionality mimics a real music bingo session. The application also offers insights about the platform, playlist catalogues and an in-app store allowing users to purchase more playlists, adding a monetisation dynamic.

The app is available exclusively to the android platform and requires network connectivity. In this report, I outlined the structure of the system, the business case for the application as well as the process of implementing the functionality as well as the technology used to support it. I also outlined a test plan to identify possible errors in my implementation and to ensure the integrity of the system.

Conclusion

Developing this application was a gratifying and satisfying experience for me. The depth and scope of technologies used within this project are more than I have ever used before, and learning a new language on top of that made the journey of the development a mix of highs and lows, but overall, I feel much more confident in my skills in programming, and I have gained proficiency in new technologies.

I feel I also gained valuable project management skills through the process of assessing the scope and objective of the project early on and adapting the changes and progress throughout the development. If I were to begin another project of this scale, I feel as though I would have the confidence to use a wider variety of API's and libraries as well as explore new concepts I would not be comfortable with. The added dynamic of COVID-19 made the experience very strange, but I feel I was resilient enough to stay motivated and on top of the deadlines I faced.

Leaving college with the skills I have acquired has left me feeling excited to start my career in computing and to push myself into new ventures.

References

Akanmu, S., Osman, W., 2013. Functional Requirements of Mobile Application for fishermen.

Flaticon, the largest database of free vector icons [WWW Document], 2021. Flaticon. URL <https://www.flaticon.com/> (accessed 4.27.21).

Functional vs Non-functional Requirements: List & Examples [WWW Document], 2021. URL <https://theappsolutions.com/blog/development/functional-vs-non-functional-requirements/> (accessed 4.27.21).

Resources

Text Generator

Cool Texts Generator in SVG / PNG with 30+ Effects x 800+ Fonts · maketext.io [WWW Document], 2021. URL <https://maketext.io/> (accessed 4.28.21).

Socket Tutorials

Wei, J., 2010. Incorporating Socket Programming into your Applications. Think Android. URL <https://thinkandroid.wordpress.com/2010/03/27/incorporating-socket-programming-into-your-applications/> (accessed 4.28.21).

Kotlin tcp with java sockets , 2020. . Sylhare's blog. URL <https://sylhare.github.io/2020/04/07/Kotlin-tcp-socket-example.html> (accessed 4.29.21).

Spotify Tutorials

Velazquez, N., 2019. Spotify's SDK in Kotlin. URL <https://tolkiana.com/how-to-use-spotifys-sdk-in-kotlin/> (accessed 4.29.21).

Android SDK Authentication Guide | Spotify for Developers [WWW Document], 2018. URL <https://developer.spotify.com/documentation/android/guides/android-authentication/> (accessed 4.29.21).

MPAndroidChart

Yeung, C.L., 2020. Using MPAndroidChart for Android Application — PieChart. Medium. URL <https://medium.com/@clyeung0714/using-mpandroidchart-for-android-application-piechart-123d62d4ddc0> (accessed 4.29.21).

Diagrams and Charts

Diagram Software and Flowchart Maker [WWW Document], 2021. URL <https://www.diagrams.net/> (accessed 4.29.21).